1. 01_Arrays_&_Hashing/01_Contains_Duplicate/0217-contains-duplicate.py

```
"""
Problem: LeetCode 217 - Contains Duplicate

Key Idea:
To check if there are any duplicates in the given list, we can use a hash set (set in Python) to store the unique elements as we traverse the list. For each element, we check if it is already present in the set. If it is, then we have found a duplicate, and we return True. If we traverse the entire list without finding any duplicates, we return False.

Time Complexity:
The time complexity of this approach is O(n), where n is the size of the input list. In the worst case, we might need to traverse the entire list once to find a duplicate.

Space Complexity:
The space complexity is O(n), as the hash set can potentially store all elements of the input list if they are all distinct.
"""


class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        hashset = set()

        for n in nums:
            if n in hashset:
                return True
            hashset.add(n)
        return False
```

2. 01_Arrays_&_Hashing/02_Valid_Anagram/0242-valid-anagram.py

```
"""
Problem: LeetCode 242 - Valid Anagram

Key Idea:
To determine if two given strings are anagrams of each other, we can compare their character frequencies. An anagram of a string contains the same characters with the same frequency, just arranged differently. We can use a hash map (dictionary in Python) to keep track of the character frequencies for each string. If the character frequencies of both strings match, then they are anagrams.

Time Complexity:
The time complexity of this approach is O(n), where n is the length of the input strings. We need to traverse both strings once to build the character frequency maps.

Space Complexity:
The space complexity is O(1) because the hash map will have at most 26 entries (one for each lowercase English letter) regardless of the input string lengths. Therefore, the space complexity is constant.
"""


class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False

        char_frequency = {}

        # Build character frequency map for string s
        for char in s:
            char_frequency[char] = char_frequency.get(char, 0) + 1

        # Compare with string t
        for char in t:
            if char not in char_frequency or char_frequency[char] == 0:
                return False
            char_frequency[char] -= 1

        return True
```

3. 01_Arrays_&_Hashing/03_Two_Sum/0001-two-sum.py

```
"""
Problem: LeetCode 1 - Two Sum

Key Idea:
The key idea to solve this problem efficiently is by using a hash map (dictionary in Python) to keep track of the elements we have traversed so far. For each element in the input list, we calculate the difference between the target and the current element. If this difference exists in the hash map, then we have found the pair that sums to the target, and we return their indices. Otherwise, we add the current element to the hash map and continue with the next element.

Time Complexity:
The time complexity of this solution is O(n), where n is the size of the input array 'nums'. In the worst case, we might need to traverse the entire array once to find the pair.

Space Complexity:
The space complexity is O(n), as the hash map can potentially store all elements of the input array 'nums' if they are all unique. In the best case, where the target sum is achieved with the first two elements, the space complexity would be O(1). However, in the worst case, if all elements are distinct, the hash map will store all elements, leading to O(n) space complexity.
"""


class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        prevMap = {}  # val -> index

        for i, n in enumerate(nums):
            diff = target - n
            if diff in prevMap:
                return [prevMap[diff], i]
            prevMap[n] = i
```

4. 01_Arrays_&_Hashing/04_Group_Anagrams/0049-group-anagrams.py

```
"""
Problem: LeetCode 49 - Group Anagrams

Key Idea:
To group anagrams together, we can use a hash map (dictionary in Python) where the key is a sorted version of each word, and the value is a list of words that are anagrams of each other. By iterating through the list of words, we can group them into the hash map based on their sorted versions.

Time Complexity:
The time complexity of this approach depends on the number of words (n) and the maximum length of a word (m). Sorting each word takes O(m*log(m)) time, and we do this for n words. Therefore, the overall time complexity is O(n * m * log(m)).

Space Complexity:
The space complexity is O(n * m), where n is the number of words, and m is the maximum length of a word. In the worst case, all words are unique, and the hash map will contain n entries, each with a list of words of length m.
"""


class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        anagrams_map = {}

        for word in strs:
            sorted_word = "".join(sorted(word))
            if sorted_word in anagrams_map:
                anagrams_map[sorted_word].append(word)
            else:
                anagrams_map[sorted_word] = [word]

        return list(anagrams_map.values())
```

5. 01_Arrays_&_Hashing/05_Top_K_Frequent_Elements/0347-top-k-frequent-elements.py

```
"""
Problem: LeetCode 347 - Top K Frequent Elements

Key Idea:
To find the k most frequent elements in the given list, we can use a hash map (dictionary in Python) to keep track of the frequency of each element. We then use a min-heap (priority queue) to keep the k most frequent elements at the top. We traverse the list once to build the frequency map, and then we traverse the map to keep the k most frequent elements in the min-heap.

Time Complexity:
The time complexity of this approach is O(n + k*log(n)), where n is the number of elements in the input list. Building the frequency map takes O(n) time, and inserting k elements into the min-heap takes O(k*log(n)) time.

Space Complexity:
The space complexity is O(n) because we need to store the frequency map of all elements in the input list. Additionally, the min-heap will also have a space complexity of O(n) in the worst case if all elements are unique.
"""

import heapq


class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        frequency_map = {}
        for num in nums:
            frequency_map[num] = frequency_map.get(num, 0) + 1

        min_heap = []
        for num, frequency in frequency_map.items():
            heapq.heappush(min_heap, (frequency, num))
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        return [num for frequency, num in min_heap]
```

6. 01_Arrays_&_Hashing/06_Product_of_Array_Except_Self/0238-product-of-array-except-self.py

```
"""
Problem: LeetCode 238 - Product of Array Except Self

Key Idea:
To solve this problem, we can first calculate the product of all elements to the left of each index and store it in a list. Then, we calculate the product of all elements to the right of each index and update the result list accordingly by multiplying it with the previously calculated left product. In this way, each element in the result list will contain the product of all elements except the one at that index.

Time Complexity:
The time complexity of this approach is O(n), where n is the number of elements in the input list. We perform two passes over the list to calculate the left and right products, and each pass takes O(n) time.

Space Complexity:
The space complexity is O(1) for the output list since we are allowed to return the result in the same list. Therefore, the space complexity is constant, as it does not depend on the size of the input list.
"""


class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        result = [1] * n

        # Calculate the left product of each element
        left_product = 1
        for i in range(n):
            result[i] *= left_product
            left_product *= nums[i]

        # Calculate the right product of each element and update the result list
        right_product = 1
        for i in range(n - 1, -1, -1):
            result[i] *= right_product
            right_product *= nums[i]

        return result
```

7. 01_Arrays_&_Hashing/07_Valid_Sudoku/0036-valid-sudoku.py

```
"""
Problem: LeetCode 36 - Valid Sudoku

Key Idea:
To determine if a given Sudoku board is valid, we need to check three conditions:
1. Each row must have distinct digits from 1 to 9.
2. Each column must have distinct digits from 1 to 9.
3. Each 3x3 sub-grid must have distinct digits from 1 to 9.

We can use three nested loops to traverse the entire board and use sets to keep track of digits seen in each row, column, and sub-grid.

Time Complexity:
The time complexity of this approach is O(1), as the Sudoku board is always a fixed 9x9 grid. We traverse each cell of the grid once, and the number of cells is constant.

Space Complexity:
The space complexity is O(1) as well because we are using a fixed amount of additional space (sets) that does not depend on the size of the input grid.
"""


class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        seen = set()

        for i in range(9):
            for j in range(9):
                if board[i][j] != ".":
                    num = board[i][j]
                    if (
                        (i, num) in seen
                        or (num, j) in seen
                        or (i // 3, j // 3, num) in seen
                    ):
                        return False
                    seen.add((i, num))
                    seen.add((num, j))
                    seen.add((i // 3, j // 3, num))

        return True
```

8. 01_Arrays_&_Hashing/08_Encode_and_Decode_Strings/0271-encode-and-decode-strings.py

```
"""
Problem: LeetCode 271 - Encode and Decode Strings
"""


class Codec:
    def encode(self, strs: List[str]) -> str:
        encoded = ""
        for s in strs:
            encoded += str(len(s)) + "#" + s
        return encoded

    def decode(self, s: str) -> List[str]:
        decoded = []
        i = 0
        while i < len(s):
            delimiter_pos = s.find("#", i)
            size = int(s[i:delimiter_pos])
            start_pos = delimiter_pos + 1
            end_pos = start_pos + size
            decoded.append(s[start_pos:end_pos])
            i = end_pos
        return decoded
```

9. 01_Arrays_&_Hashing/09_Longest_Consecutive_Sequence/0128-longest-consecutive-sequence.py

```
"""
Problem: LeetCode 128 -Longest Consecutive Sequence

Key Idea:
To find the longest consecutive subsequence, we first create a set of all the elements in the input array 'nums'. Then, for each element in the array, we check if it is the starting element of a consecutive subsequence. To do this, we check if the element before the current element (i.e., nums[i] - 1) exists in the set. If it doesn't, it means nums[i] is the starting element of a consecutive subsequence. From here, we keep incrementing the current element until the consecutive subsequence ends (i.e., the next element does not exist in the set). We keep track of the length of the consecutive subsequence and update the maximum length found so far.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'nums'. The set operations (addition and lookup) take O(1) time on average, and we perform these operations for each element once.

Space Complexity:
The space complexity is O(n), where n is the number of elements in the input array 'nums'. In the worst case, the set will store all elements from the input array if they are all distinct.
"""


class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        num_set = set(nums)
        max_length = 0

        for num in num_set:
            if num - 1 not in num_set:
                current_num = num
                current_length = 1

                while current_num + 1 in num_set:
                    current_num += 1
                    current_length += 1

                max_length = max(max_length, current_length)

        return max_length
```

10. 02_Two_Pointers/01_Valid_Palindrome/0125-valid-palindrome.py

```
"""
Problem: LeetCode 125 - Valid Palindrome

Key Idea:
To determine if a given string is a valid palindrome, we can use two pointers approach. We initialize two pointers, one at the beginning of the string (left) and the other at the end of the string (right). We then compare characters at these two pointers. If they are both alphanumeric characters and equal in value (ignoring case), we move both pointers towards the center of the string. If they are not equal, we know the string is not a palindrome. We continue this process until the two pointers meet or cross each other.

Time Complexity:
The time complexity of this solution is O(n), where n is the length of the input string 's'. In the worst case, we might need to traverse the entire string once to determine if it is a valid palindrome.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the two pointers and other variables.
"""


class Solution:
    def isPalindrome(self, s: str) -> bool:
        left, right = 0, len(s) - 1

        while left < right:
            while left < right and not s[left].isalnum():
                left += 1
            while left < right and not s[right].isalnum():
                right -= 1

            if s[left].lower() != s[right].lower():
                return False

            left += 1
            right -= 1

        return True
```

11. 02_Two_Pointers/02_Two_Sum_II_-_Input_Array_Is_Sorted/0167-two-sum-ii-input-array-is-sorted.py

```
"""
Problem: LeetCode 167 - Two Sum II

Key Idea:
The input array 'numbers' is already sorted in non-decreasing order. To find the two numbers that add up to the target, we can use a two-pointer approach. We initialize two pointers, one at the beginning of the array (left) and the other at the end of the array (right). We then check the sum of the elements at these two pointers. If the sum is equal to the target, we have found the pair. If the sum is less than the target, it means we need to increase the sum, so we move the left pointer one step to the right. If the sum is greater than the target, it means we need to decrease the sum, so we move the right pointer one step to the left. We continue this process until we find the pair or the two pointers meet or cross each other.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'numbers'. The two pointers approach iterates through the array once, and at each step, we move at least one of the pointers, so we do not revisit any element.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the two pointers and other variables.
"""


class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        left, right = 0, len(numbers) - 1

        while left < right:
            current_sum = numbers[left] + numbers[right]

            if current_sum == target:
                return [left + 1, right + 1]
            elif current_sum < target:
                left += 1
            else:
                right -= 1

        # No solution found
        return [-1, -1]
```

12. 02_Two_Pointers/03_3Sum/0015-3sum.py

```
"""
Problem: LeetCode 15 - 3Sum

Key Idea:
To find all unique triplets that sum to zero, we can use a three-pointer approach. First, we sort the input array 'nums' in non-decreasing order. Then, we iterate through the array with a fixed first element (i). For each fixed first element, we use two pointers (left and right) to find the other two elements that sum to the negation of the fixed first element. As the array is sorted, we can move these two pointers towards each other to efficiently find all possible triplets.

Time Complexity:
The time complexity of this solution is O(n^2), where n is the number of elements in the input array 'nums'. Sorting the array takes O(nlogn) time, and the two-pointer approach iterates through the array once, performing a linear search for each element.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the three pointers and other variables.
"""


class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        result = []
        n = len(nums)

        for i in range(n - 2):
            if i > 0 and nums[i] == nums[i - 1]:
                continue

            left, right = i + 1, n - 1

            while left < right:
                current_sum = nums[i] + nums[left] + nums[right]

                if current_sum == 0:
                    result.append([nums[i], nums[left], nums[right]])

                    # Skip duplicates
                    while left < right and nums[left] == nums[left + 1]:
                        left += 1
                    while left < right and nums[right] == nums[right - 1]:
                        right -= 1

                    left += 1
                    right -= 1
                elif current_sum < 0:
                    left += 1
                else:
                    right -= 1

        return result
```

13. 02_Two_Pointers/04_Container_With_Most_Water/0011-container-with-most-water.py

```
"""
Problem: LeetCode 11 - Container With Most Water

Key Idea:
To find the maximum area of water that can be held between two vertical lines, we can use a two-pointer approach. We initialize two pointers, one at the beginning of the input array (left) and the other at the end of the array (right). The area between the two vertical lines is calculated as the minimum of the heights at the two pointers multiplied by the distance between them. We then update the maximum area found so far and move the pointer with the smaller height towards the other pointer. We continue this process until the two pointers meet or cross each other.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'height'. The two-pointer approach iterates through the array once, and at each step, we move at least one of the pointers, so we do not revisit any element.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the two pointers and other variables.
"""


class Solution:
    def maxArea(self, height: List[int]) -> int:
        left, right = 0, len(height) - 1
        max_area = 0

        while left < right:
            current_area = min(height[left], height[right]) * (right - left)
            max_area = max(max_area, current_area)

            if height[left] < height[right]:
                left += 1
            else:
                right -= 1

        return max_area
```

14. 02_Two_Pointers/05_Trapping_Rain_Water/0042-trapping-rain-water.py

```
"""
Problem: LeetCode 42 - Trapping Rain Water

Key Idea:
To find the amount of trapped rainwater in the given elevation histogram represented by the input array 'height', we can use a two-pointer approach. We initialize two pointers, one at the beginning of the array (left) and the other at the end of the array (right). We also initialize two variables to keep track of the maximum left height and maximum right height seen so far. While the left pointer is less than the right pointer, we compare the height at the left and right pointers. If the height at the left pointer is less than or equal to the height at the right pointer, it means we can trap water between the left pointer and the maximum left height. Otherwise, we can trap water between the right pointer and the maximum right height. At each step, we update the trapped water amount and move the pointers and update the maximum heights accordingly.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'height'. The two-pointer approach iterates through the array once, and at each step, we move at least one of the pointers, so we do not revisit any element.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the two pointers and other variables.
"""


class Solution:
    def trap(self, height: List[int]) -> int:
        left, right = 0, len(height) - 1
        max_left, max_right = 0, 0
        trapped_water = 0

        while left < right:
            if height[left] <= height[right]:
                if height[left] >= max_left:
                    max_left = height[left]
                else:
                    trapped_water += max_left - height[left]
                left += 1
            else:
                if height[right] >= max_right:
                    max_right = height[right]
                else:
                    trapped_water += max_right - height[right]
                right -= 1

        return trapped_water
```

15. 03_Sliding_Window/01_Best_Time_to_Buy_and_Sell_Stock/0121-best-time-to-buy-and-sell-stock.py

```
"""
Problem: LeetCode 121 - Best Time to Buy and Sell Stock

Key Idea:
To find the maximum profit from a single buy and sell transaction in the input array 'prices', we can use a simple one-pass approach. We initialize two variables, 'min_price' to keep track of the minimum price seen so far, and 'max_profit' to store the maximum profit. We iterate through the 'prices' array, and for each price, we update the 'min_price' if we find a smaller price. We also calculate the potential profit if we sell at the current price and update 'max_profit' if the current profit is greater than the previous maximum profit.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'prices'. We iterate through the array once to find the maximum profit, and at each step, we perform constant-time operations.

Space Complexity:
The space complexity is O(1) since we are not using any additional data structures that depend on the input size. We only use a constant amount of extra space for the variables to store the minimum price and maximum profit.
"""


class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        min_price = float("inf")
        max_profit = 0

        for price in prices:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)

        return max_profit
```

16. 03_Sliding_Window/02_Longest_Substring_Without_Repeating_Characters/0003-longest-substring-without-repeating-characters.py

```
"""
Problem: LeetCode 3 - Longest Substring Without Repeating Characters

Key Idea:
To find the length of the longest substring without repeating characters in the input string 's', we can use the sliding window approach. We use two pointers, 'left' and 'right', to represent the current window. As we move the 'right' pointer to the right, we expand the window and add characters to a set to keep track of unique characters in the window. If we encounter a repeating character, we move the 'left' pointer to the right to shrink the window until the repeating character is no longer in the window. At each step, we update the maximum length of the window (i.e., the length of the longest substring without repeating characters).

Time Complexity:
The time complexity of this solution is O(n), where n is the length of the input string 's'. The sliding window approach iterates through the string once, and at each step, we perform constant-time operations to update the window and the maximum length.

Space Complexity:
The space complexity is O(k), where k is the number of unique characters in the input string 's'. In the worst case, the entire string can be composed of unique characters, so the set storing the characters in the window would have k elements.
"""


class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        left, right = 0, 0
        max_length = 0
        unique_chars = set()

        while right < len(s):
            if s[right] not in unique_chars:
                unique_chars.add(s[right])
                max_length = max(max_length, right - left + 1)
                right += 1
            else:
                unique_chars.remove(s[left])
                left += 1

        return max_length
```

17. 03_Sliding_Window/03_Longest_Repeating_Character_Replacement/0424-longest-repeating-character-replacement.py

```
"""
Problem: LeetCode 424 - Longest Repeating Character Replacement

Key Idea:
To find the maximum length of a substring with at most k distinct characters in the input string 's', we can use the sliding window approach. We use two pointers, 'left' and 'right', to represent the current window. As we move the 'right' pointer to the right, we expand the window and add characters to a dictionary to keep track of their frequencies. If the number of distinct characters in the window exceeds k, we move the 'left' pointer to the right to shrink the window until the number of distinct characters is k again. At each step, we update the maximum length of the window.

Time Complexity:
The time complexity of this solution is O(n), where n is the length of the input string 's'. The sliding window approach iterates through the string once, and at each step, we perform constant-time operations to update the window and the maximum length.

Space Complexity:
The space complexity is O(k), where k is the maximum number of distinct characters allowed in the input string 's'. In the worst case, the entire string can have k distinct characters, so the dictionary storing the character frequencies in the window would have k elements.
"""


class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        left, right = 0, 0
        max_length = 0
        char_freq = {}
        max_freq = 0

        while right < len(s):
            char_freq[s[right]] = char_freq.get(s[right], 0) + 1
            max_freq = max(max_freq, char_freq[s[right]])

            if (right - left + 1) - max_freq > k:
                char_freq[s[left]] -= 1
                left += 1

            max_length = max(max_length, right - left + 1)
            right += 1

        return max_length
```

18. 03_Sliding_Window/04_Permutation_in_String/0567-permutation-in-string.py

```
"""
Problem: LeetCode 567 - Permutation in String

Key Idea:
To check whether a string 's2' contains a permutation of another string 's1', we can use a sliding window approach. First, we create a frequency dictionary for characters in 's1'. Then, we initialize two pointers, 'left' and 'right', to represent the current window in 's2'. As we move the 'right' pointer to the right, we add the character to a temporary frequency dictionary and check if it becomes equal to the frequency dictionary of 's1'. If it does, it means we found a permutation of 's1' in 's2', and we return True. If the window size exceeds the length of 's1', we remove the character at the 'left' pointer from the temporary dictionary and move the 'left' pointer to the right to shrink the window. We continue this process until we find a permutation or reach the end of 's2'.

Time Complexity:
The time complexity of this solution is O(n), where n is the length of 's2'. The sliding window approach iterates through 's2' once, and at each step, we perform constant-time operations to update the window and the frequency dictionaries.

Space Complexity:
The space complexity is O(1) since the frequency dictionaries have a constant number of elements (the number of unique characters in 's1' or the alphabet size) regardless of the size of the input strings 's1' and 's2'.
"""


class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        if len(s1) > len(s2):
            return False

        char_freq_s1 = {}
        for char in s1:
            char_freq_s1[char] = char_freq_s1.get(char, 0) + 1

        left, right = 0, 0
        char_freq_temp = {}

        while right < len(s2):
            char_freq_temp[s2[right]] = char_freq_temp.get(s2[right], 0) + 1

            if right - left + 1 == len(s1):
                if char_freq_temp == char_freq_s1:
                    return True
                char_freq_temp[s2[left]] -= 1
                if char_freq_temp[s2[left]] == 0:
                    del char_freq_temp[s2[left]]
                left += 1

            right += 1

        return False
```

19. 03_Sliding_Window/05_Minimum_Window_Substring/0076-minimum-window-substring.py

```
"""
Problem: LeetCode 76 - Minimum Window Substring

Key Idea:
To find the minimum window in the input string 's' that contains all characters from another string 't', we can use the sliding window approach. We first create a frequency dictionary for characters in 't'. Then, we initialize two pointers, 'left' and 'right', to represent the current window in 's'. As we move the 'right' pointer to the right, we add the character to a temporary frequency dictionary and check if it contains all characters from 't'. If it does, it means we found a valid window containing all characters from 't'. We update the minimum window length and move the 'left' pointer to the right to shrink the window. We continue this process until we find the minimum window or reach the end of 's'.

Time Complexity:
The time complexity of this solution is O(n), where n is the length of 's'. The sliding window approach iterates through 's' once, and at each step, we perform constant-time operations to update the window and the frequency dictionaries.

Space Complexity:
The space complexity is O(k), where k is the number of unique characters in 't'. In the worst case, the frequency dictionaries can have k elements, one for each unique character in 't'.
"""


class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if not s or not t:
            return ""

        char_freq_t = {}
        for char in t:
            char_freq_t[char] = char_freq_t.get(char, 0) + 1

        left, right = 0, 0
        char_freq_temp = {}
        required_chars = len(char_freq_t)
        formed_chars = 0
        min_length = float("inf")
        min_window = ""

        while right < len(s):
            char_freq_temp[s[right]] = char_freq_temp.get(s[right], 0) + 1

            if (
                s[right] in char_freq_t
                and char_freq_temp[s[right]] == char_freq_t[s[right]]
            ):
                formed_chars += 1

            while left <= right and formed_chars == required_chars:
                if right - left + 1 < min_length:
                    min_length = right - left + 1
                    min_window = s[left : right + 1]

                char_freq_temp[s[left]] -= 1
                if (
                    s[left] in char_freq_t
                    and char_freq_temp[s[left]] < char_freq_t[s[left]]
                ):
                    formed_chars -= 1
```

```
            left += 1

        right += 1

    return min_window
```

20. 03_Sliding_Window/06_Sliding_Window_Maximum/0239-sliding-window-maximum.py

```
"""
Problem: LeetCode 239 - Sliding Window Maximum

Key Idea:
To find the maximum sliding window of size 'k' in the input array 'nums', we can use a deque (double-ended queue). The deque will store the indices of elements in 'nums' such that the elements at these indices are in decreasing order. As we traverse the array 'nums', we add the current element to the deque, but before adding, we remove elements from the back of the deque that are smaller than the current element. This ensures that the front element of the deque will always be the maximum element in the window. At each step, we check if the front element's index is within the valid range of the current window. If it is not, we remove the front element from the deque. As we traverse the array, we can build the maximum sliding window using the elements stored in the deque.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'nums'. We traverse the array once, and at each step, we perform constant-time operations for adding and removing elements from the deque.

Space Complexity:
The space complexity is O(k), where k is the size of the sliding window. The deque will store at most 'k' elements at any point during the traversal, representing the maximum window size.
"""

from collections import deque


class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums or k <= 0:
            return []

        result = []
        window = deque()

        for i, num in enumerate(nums):
            while window and nums[window[-1]] < num:
                window.pop()

            window.append(i)

            if i - window[0] >= k:
                window.popleft()

            if i >= k - 1:
                result.append(nums[window[0]])

        return result
```

21. 04_Stack/01_Valid_Parentheses/0020-valid-parentheses.py

```
"""
Problem: LeetCode 20 - Valid Parentheses

Key Idea:
To determine if a given string of parentheses 's' is valid, we can use a stack data structure. We iterate through each character in 's', and if the character is an opening parenthesis ('(', '{', '['), we push it onto the stack. If the character is a closing parenthesis (')', '}', ']'), we check if the stack is empty or if the top element of the stack does not match the current closing parenthesis. If either of these conditions is met, we know the string is not valid. Otherwise, we pop the top element from the stack. At the end, if the stack is empty, the string is valid.

Time Complexity:
The time complexity of this solution is O(n), where n is the length of the input string 's'. We iterate through the string once, and each operation (pushing or popping from the stack) takes constant time.

Space Complexity:
The space complexity is O(n), where n is the length of the input string 's'. In the worst case, the stack could store all characters of the input string.
"""


class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        parentheses_map = {")": "(", "}": "{", "]": "["}

        for char in s:
            if char in parentheses_map.values():
                stack.append(char)
            elif char in parentheses_map:
                if not stack or stack[-1] != parentheses_map[char]:
                    return False
                stack.pop()
            else:
                return False

        return not stack
```

22. 04_Stack/02_Min_Stack/0155-min-stack.py

```
"""
Problem: LeetCode 155 - Min Stack

Key Idea:
To implement a stack that supports finding the minimum element in constant time, we can use two stacks: one for storing the actual elements (stack) and another for keeping track of the minimum elements (min_stack). The min_stack will always have the current minimum element at the top. When pushing an element onto the stack, we compare it with the top element of the min_stack and push the smaller element onto the min_stack. When popping an element from the stack, we check if the element being popped is the same as the top element of the min_stack. If it is, we also pop the element from the min_stack. This way, the top element of the min_stack will always be the minimum element in the stack.

Time Complexity:
The time complexity of push, pop, top, and getMin operations is O(1). All these operations involve constant-time operations on the two stacks.

Space Complexity:
The space complexity is O(n), where n is the number of elements in the stack. Both the stack and min_stack can potentially store all elements from the input.
"""


class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val: int) -> None:
        self.stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self) -> None:
        if self.stack:
            if self.stack[-1] == self.min_stack[-1]:
                self.min_stack.pop()
            self.stack.pop()

    def top(self) -> int:
        if self.stack:
            return self.stack[-1]

    def getMin(self) -> int:
        if self.min_stack:
            return self.min_stack[-1]
```

23. 04_Stack/03_Evaluate_Reverse_Polish_Notation/0150-evaluate-reverse-polish-notation.py

```
"""
Problem: LeetCode 150 - Evaluate Reverse Polish Notation

Key Idea:
To evaluate a given reverse Polish notation expression, we can use a stack data structure. We iterate through the tokens in the expression, and for each token, if it is a number, we push it onto the stack. If it is an operator ('+', '-', '*', '/'), we pop the top two elements from the stack, apply the operator to them, and push the result back onto the stack. At the end, the top element of the stack will be the final result of the expression.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of tokens in the expression. We iterate through the tokens once, and each operation (pushing, popping, and applying operators) takes constant time.

Space Complexity:
The space complexity is O(n), where n is the number of tokens in the expression. In the worst case, the stack could store all tokens from the input expression.
"""


class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []

        for token in tokens:
            if token.isdigit() or (token[0] == "-" and token[1:].isdigit()):
                stack.append(int(token))
            else:
                num2 = stack.pop()
                num1 = stack.pop()
                if token == "+":
                    stack.append(num1 + num2)
                elif token == "-":
                    stack.append(num1 - num2)
                elif token == "*":
                    stack.append(num1 * num2)
                elif token == "/":
                    stack.append(int(num1 / num2))

        return stack[0]
```

24. 04_Stack/04_Generate_Parentheses/0022-generate-parentheses.py

```
"""
Problem: LeetCode 22 - Generate Parentheses

Key Idea:
To generate all valid combinations of parentheses, we can use backtracking. We start with an empty string and two counters, one for the open parentheses and one for the close parentheses. At each step, we have two choices: add an open parenthesis if the count of open parentheses is less than the total number of pairs, or add a close parenthesis if the count of close parentheses is less than the count of open parentheses. We continue this process recursively until we reach the desired length of the string. If the string becomes valid, we add it to the result.

Time Complexity:
The time complexity of this solution is O(4^n / sqrt(n)), where n is the number of pairs of parentheses. This is because each valid combination is a sequence of open and close parentheses of length 2n, and there are 2^(2n) such sequences. However, not all sequences are valid, and the Catalan number (4^n / sqrt(n)) bounds the number of valid combinations.

Space Complexity:
The space complexity is O(4^n / sqrt(n)) as well, as this is the maximum number of valid combinations that can be generated.
"""


class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        def backtrack(s, open_count, close_count):
            if len(s) == 2 * n:
                result.append(s)
                return

            if open_count < n:
                backtrack(s + "(", open_count + 1, close_count)
            if close_count < open_count:
                backtrack(s + ")", open_count, close_count + 1)

        result = []
        backtrack("", 0, 0)
        return result
```

25. 04_Stack/05_Daily_Temperatures/0739-daily-temperatures.py

```
"""
Problem: LeetCode 739 - Daily Temperatures

Key Idea:
To find the daily temperatures that are warmer in the input array 'temperatures', we can use a stack. We iterate through the temperatures in reverse order, and for each temperature, we pop elements from the stack while they are smaller than the current temperature. This indicates that the current temperature is the first warmer temperature for the popped elements. We keep track of the indices of these warmer temperatures in the result array. Then, we push the current temperature's index onto the stack. At the end, the result array will contain the number of days until the next warmer temperature for each day.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input array 'temperatures'. We iterate through the array once, and each element is pushed onto and popped from the stack at most once.

Space Complexity:
The space complexity is O(n), where n is the number of elements in the input array 'temperatures'. In the worst case, all temperatures could be strictly increasing, leading to all elements being pushed onto the stack.
"""


class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        stack = []
        result = [0] * len(temperatures)

        for i in range(len(temperatures) - 1, -1, -1):
            while stack and temperatures[i] >= temperatures[stack[-1]]:
                stack.pop()
            if stack:
                result[i] = stack[-1] - i
            stack.append(i)

        return result
```

26. 04_Stack/06_Car_Fleet/0853-car-fleet.py

```
"""
Problem: LeetCode 853 - car Fleet

Key Idea:
To determine the number of car fleets that can reach the target destination in the input arrays 'target' and 'position', we can simulate the car movements and calculate the time it takes for each car to reach the target. We can then sort the cars based on their positions and iterate through them. For each car, we calculate its time to reach the target and compare it with the previous car. If the time for the current car is greater, it means the previous car cannot catch up to it, so we consider the current car as a new fleet. Otherwise, the previous car can catch up to the current car, so they form a fleet together.

Time Complexity:
The time complexity of this solution is O(n log n), where n is the number of cars. The dominating factor is the sorting step, which takes O(n log n) time. The subsequent iteration through the sorted cars takes linear time.

Space Complexity:
The space complexity is O(n), where n is the number of cars. We need additional space to store the time-to-reach values for each car.
"""


class Solution:
    def carFleet(self, target: int, position: List[int], speed: List[int]) -> int:
        cars = sorted(zip(position, speed), reverse=True)
        fleets = 0
        prev_time = -1.0

        for pos, spd in cars:
            time = (target - pos) / spd
            if time > prev_time:
                fleets += 1
                prev_time = time

        return fleets
```

27. 04_Stack/07_Largest_Rectangle_In_Histogram/0084-largest-rectangle-in-histogram.py

```
"""
Problem: LeetCode 84 - Largest Rectangle in Histogram

Key Idea:
To find the largest rectangle area in the input histogram represented by the list 'heights', we can use a stack to keep track of increasing bar heights' indices. We iterate through the heights and push the current index onto the stack if the current height is greater than or equal to the height at the top of the stack. If the current height is smaller, it indicates that the previous bars cannot form a larger rectangle, so we pop indices from the stack and calculate the area for each popped bar. The width of the rectangle is determined by the difference between the current index and the index at the top of the stack. The height of the rectangle is the height at the popped index. We keep track of the maximum area seen so far.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the input list 'heights'. We iterate through the heights once, and each element is pushed onto and popped from the stack at most once.

Space Complexity:
The space complexity is O(n), where n is the number of elements in the input list 'heights'. In the worst case, all elements could be pushed onto the stack.
"""


class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = []
        max_area = 0

        for i in range(len(heights)):
            while stack and heights[i] < heights[stack[-1]]:
                height = heights[stack.pop()]
                width = i if not stack else i - stack[-1] - 1
                max_area = max(max_area, height * width)
            stack.append(i)

        while stack:
            height = heights[stack.pop()]
            width = len(heights) if not stack else len(heights) - stack[-1] - 1
            max_area = max(max_area, height * width)

        return max_area
```

28. 05_Binary_Search/01_Binary_Search/0704-binary-search.py

```
"""
Problem: LeetCode 704 - Binary Search

Key Idea:
Binary search is an efficient technique to search for a target element in a sorted array. In each step, we compare the middle element of the array with the target. If the middle element is equal to the target, we have found the element and return its index. If the middle element is greater than the target, we narrow down the search to the left half of the array. If the middle element is smaller, we narrow down the search to the right half of the array. We repeat this process until we find the target element or the search space is exhausted.

Time Complexity:
The time complexity of binary search is O(log n), where n is the number of elements in the sorted array. In each step, we reduce the search space by half, which leads to a logarithmic number of steps.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

        return -1
```

29. 05_Binary_Search/02_Search_a_2D_Matrix/0074-search-a-2d-matrix.py

```
"""
Problem: LeetCode 74 - Search a 2D Matrix

Key Idea:
Since both the rows and columns in the input 2D matrix are sorted, we can treat the matrix as a one-dimensional sorted array and perform binary search to find the target element. We can map the 2D indices to the corresponding index in the 1D array and then apply binary search to locate the target element.

Time Complexity:
The time complexity of this solution is O(log(m*n)), where m is the number of rows and n is the number of columns in the matrix. The binary search process reduces the search space by half in each step.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        if not matrix or not matrix[0]:
            return False

        rows, cols = len(matrix), len(matrix[0])
        left, right = 0, rows * cols - 1

        while left <= right:
            mid = left + (right - left) // 2
            num = matrix[mid // cols][mid % cols]

            if num == target:
                return True
            elif num < target:
                left = mid + 1
            else:
                right = mid - 1

        return False
```

30. 05_Binary_Search/03_Koko_Eating_Bananas/0875-koko-eating-bananas.py

```
"""
Problem: LeetCode 875 - Koko Eating Bananas

Key Idea:
The key idea is to perform binary search to find the minimum value of the integer 'k' such that Koko can eat all the bananas within 'hours' hours. We can define a binary search space for 'k' and perform binary search to find the smallest 'k' that satisfies the given condition.

Time Complexity:
The time complexity of this solution is O(n * log(max_pile)), where n is the number of piles and max_pile is the maximum size of a pile. The binary search iterates log(max_pile) times, and for each iteration, we perform a linear search through 'piles' to calculate the total hours.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        left, right = 1, max(piles)

        while left < right:
            mid = left + (right - left) // 2
            hours = sum((pile + mid - 1) // mid for pile in piles)

            if hours > h:
                left = mid + 1
            else:
                right = mid

        return left
```

31. 05_Binary_Search/04_Find_Minimum_In_Rotated_Sorted_Array/0153-find-minimum-in-rotated-sorted-array.py

```
"""
Problem: LeetCode 153 - Find Minimum in Rotated Sorted Array

Key Idea:
The key idea is to perform binary search to find the minimum element in the rotated sorted array. We compare the middle element with its neighbors to determine if it is the minimum element. Depending on the comparison, we narrow down the search to the unsorted part of the array.

Time Complexity:
The time complexity of this solution is O(log n), where n is the number of elements in the input array. Binary search reduces the search space by half in each step.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1

        while left < right:
            mid = left + (right - left) // 2

            if nums[mid] > nums[right]:
                left = mid + 1
            else:
                right = mid

        return nums[left]
```

32. 05_Binary_Search/05_Search_In_Rotated_Sorted_Array/0033-search-in-rotated-sorted-array.py

```
"""
Problem: LeetCode 33 - Search in Rotated Sorted Array

Key Idea:
The key idea is to perform binary search to find the target element in the rotated sorted array. We compare the middle element with the target and the endpoints of the subarray to determine which part of the array is sorted. Depending on the comparison, we narrow down the search to the sorted part of the array.

Time Complexity:
The time complexity of this solution is O(log n), where n is the number of elements in the input array. Binary search reduces the search space by half in each step.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid

            if nums[left] <= nums[mid]:
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1

        return -1
```

33. 05_Binary_Search/06_Time_Based_Key_Value_Store/0981-time-based-key-value-store.py

```
"""
Problem: LeetCode 981 - Time Based Key-Value Store

Key Idea:
To implement a time-based key-value store, we can use a dictionary to store the values associated with each key. For each key, we store a list of tuples representing the timestamp and the corresponding value. When querying a key at a specific timestamp, we perform binary search on the list of timestamps associated with that key to find the largest timestamp less than or equal to the given timestamp.

Time Complexity:
1. The time complexity of the 'set' operation is O(1), as it involves adding a new value to the dictionary.
2. The time complexity of the 'get' operation is O(log n), where n is the number of timestamps associated with the queried key. This is due to the binary search performed to find the appropriate value.

Space Complexity:
The space complexity is O(n), where n is the number of keys. We need to store values and timestamps for each key in the dictionary.
"""


class TimeMap:
    def __init__(self):
        self.data = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.data[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        values = self.data[key]
        left, right = 0, len(values) - 1

        while left <= right:
            mid = left + (right - left) // 2
            if values[mid][0] == timestamp:
                return values[mid][1]
            elif values[mid][0] < timestamp:
                left = mid + 1
            else:
                right = mid - 1

        if right >= 0:
            return values[right][1]
        return ""
```

34. 05_Binary_Search/07_Median_of_Two_Sorted_Arrays/0004-median-of-two-sorted-arrays.py

```
"""
Problem: LeetCode 4 - Median of Two Sorted Arrays

Key Idea:
To find the median of two sorted arrays 'nums1' and 'nums2', we can perform a binary search on the smaller array. We partition both arrays into two parts such that the left half contains smaller elements and the right half contains larger elements. The median will be the average of the maximum element in the left half and the minimum element in the right half. We adjust the partition indices based on binary search, aiming to keep the same number of elements in both halves.

Time Complexity:
The time complexity of this solution is O(log(min(m, n))), where m and n are the lengths of 'nums1' and 'nums2'. The binary search reduces the search space by half in each step.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1

        m, n = len(nums1), len(nums2)
        low, high = 0, m

        while low <= high:
            partition1 = (low + high) // 2
            partition2 = (m + n + 1) // 2 - partition1

            maxLeft1 = float("-inf") if partition1 == 0 else nums1[partition1 - 1]
            minRight1 = float("inf") if partition1 == m else nums1[partition1]

            maxLeft2 = float("-inf") if partition2 == 0 else nums2[partition2 - 1]
            minRight2 = float("inf") if partition2 == n else nums2[partition2]

            if maxLeft1 <= minRight2 and maxLeft2 <= minRight1:
                if (m + n) % 2 == 0:
                    return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) / 2
                else:
                    return max(maxLeft1, maxLeft2)
            elif maxLeft1 > minRight2:
                high = partition1 - 1
            else:
                low = partition1 + 1

        raise ValueError("Input arrays are not sorted.")
```

35. 06_Linked_List/01_Reverse_Linked_List/0206-reverse-linked-list.py

```
"""
Problem: LeetCode 206 - Reverse Linked List

Key Idea:
To reverse a singly linked list, we need to reverse the direction of the pointers while traversing the list. We maintain three pointers: 'prev' (to keep track of the previous node), 'current' (to keep track of the current node), and 'next_node' (to keep track of the next node in the original list). In each iteration, we update the 'current.next' pointer to point to the 'prev' node, and then move 'prev' and 'current' pointers one step forward. We repeat this process until we reach the end of the original list, and the 'prev' pointer will be pointing to the new head of the reversed list.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. We traverse each node once to reverse the list.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of nodes and pointers.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        prev = None
        current = head

        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        return prev
```

36. 06_Linked_List/02_Merge_Two_Sorted_Lists/0021-merge-two-sorted-lists.py

```
"""
Problem: LeetCode 21 - Merge Two Sorted Lists

Key Idea:
To merge two sorted linked lists 'l1' and 'l2', we can create a new linked list 'dummy' to hold the merged result. We maintain two pointers, 'current' and 'prev', to traverse through the two input lists. At each step, we compare the values at the 'current' pointers of 'l1' and 'l2', and add the smaller value to the 'dummy' list. We then move the 'current' pointer of the list with the smaller value one step forward. After iterating through both lists, if any list still has remaining elements, we append them to the 'dummy' list.

Time Complexity:
The time complexity of this solution is O(n), where n is the total number of nodes in the merged list. We traverse each node once to merge the lists.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of nodes and pointers.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode()
        current = dummy

        while l1 and l2:
            if l1.val < l2.val:
                current.next = l1
                l1 = l1.next
            else:
                current.next = l2
                l2 = l2.next
            current = current.next

        if l1:
            current.next = l1
        elif l2:
            current.next = l2

        return dummy.next
```

37. 06_Linked_List/03_Reorder_List/0143-reorder-list.py

```
"""
Proble: LeetCode 143 - Reorder List

Key Idea:
To reorder a singly linked list, we can break the list into two halves, reverse the second half, and then merge the two halves alternatively. First, we find the middle of the list using the slow and fast pointer technique. We reverse the second half of the list in place. Finally, we merge the two halves by alternating nodes from each half.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. We traverse the list to find the middle, reverse the second half, and merge the two halves.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of nodes and pointers.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head or not head.next or not head.next.next:
            return

        # Find the middle of the list
        slow, fast = head, head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        # Reverse the second half of the list
        prev, current = None, slow.next
        slow.next = None
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        # Merge the two halves alternately
        p1, p2 = head, prev
        while p2:
            next_p1, next_p2 = p1.next, p2.next
            p1.next = p2
            p2.next = next_p1
            p1, p2 = next_p1, next_p2
```

38. 06_Linked_List/04_Remove_Nth_Node_From_End_of_List/0019-remove-nth-node-from-end-of-list.py

```
"""
Problem: LeetCode 19 - Remove Nth Node From End of List

Key Idea:
To remove the nth node from the end of a singly linked list, we can use the two-pointer approach. We maintain two pointers, 'fast' and 'slow', where 'fast' moves n nodes ahead of 'slow'. Then we move both pointers simultaneously until 'fast' reaches the end of the list. At this point, 'slow' will be pointing to the node just before the node to be removed. We update the 'slow.next' pointer to skip the node to be removed.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. We traverse the list twice, once to find the length of the list and once to remove the node.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of nodes and pointers.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        dummy = ListNode(0)
        dummy.next = head
        fast = slow = dummy

        # Move 'fast' n nodes ahead
        for _ in range(n):
            fast = fast.next

        # Move both pointers until 'fast' reaches the end
        while fast.next:
            fast = fast.next
            slow = slow.next

        # Remove the nth node from the end
        slow.next = slow.next.next

        return dummy.next
```

39. 06_Linked_List/05_Copy_List_With_Random_Pointer/0138-copy-list-with-random-pointer.py

```
"""
Problem: LeetCode 138 - Copy List with Random Pointer

Key Idea:
To create a deep copy of a linked list with random pointers, we can follow a three-step approach. First, we duplicate each node in the original list and insert the duplicates right after their corresponding original nodes. Second, we update the random pointers of the duplicate nodes to point to the correct nodes. Finally, we split the combined list into two separate lists: the original list and the duplicated list.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. We traverse the list three times, once to duplicate the nodes, once to update the random pointers, and once to split the lists.

Space Complexity:
The space complexity is O(n), as we need to create a duplicate node for each original node. Additionally, we use a dictionary to map original nodes to their corresponding duplicate nodes, which takes up O(n) extra space.
"""

# Definition for a Node.
# class Node:
#     def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
#         self.val = int(x)
#         self.next = next
#         self.random = random


class Solution:
    def copyRandomList(self, head: "Node") -> "Node":
        if not head:
            return None

        # Step 1: Duplicate nodes and insert them in between the original nodes
        current = head
        while current:
            duplicate = Node(current.val)
            duplicate.next = current.next
            current.next = duplicate
            current = duplicate.next

        # Step 2: Update random pointers for the duplicate nodes
        current = head
        while current:
            if current.random:
                current.next.random = current.random.next
            current = current.next.next

        # Step 3: Split the combined list into two separate lists
        original = head
        duplicate_head = head.next
        current = duplicate_head
        while original:
            original.next = original.next.next
            if current.next:
                current.next = current.next.next
            original = original.next
            current = current.next

        return duplicate_head
```

40. 06_Linked_List/06_Add_Two_Numbers/0002-add-two-numbers.py

```
"""
Problem: LeetCode 2 - Add Two Numbers

Key Idea:
To add two numbers represented by linked lists, we can simulate the addition digit by digit while considering carry. We maintain a dummy node to build the resulting linked list. We iterate through the input lists, summing the corresponding digits along with any carry from the previous digit. We update the carry and create a new node with the sum digit. After processing both lists, if there is a carry remaining, we add a new node with the carry.

Time Complexity:
The time complexity of this solution is O(max(m, n)), where m and n are the lengths of the input linked lists. We traverse both lists once.

Space Complexity:
The space complexity is O(max(m, n)), as we create a new linked list to store the result. Additionally, we use a few extra variables to keep track of nodes and values.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode()
        current = dummy
        carry = 0

        while l1 or l2:
            val1 = l1.val if l1 else 0
            val2 = l2.val if l2 else 0
            total = val1 + val2 + carry

            carry = total // 10
            digit = total % 10

            current.next = ListNode(digit)
            current = current.next

            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next

        if carry:
            current.next = ListNode(carry)

        return dummy.next
```

41. 06_Linked_List/07_Linked_List_Cycle/0141-linked-list-cycle.py

```
"""
Problem: LeetCode 141 - Linked List Cycle

Key Idea:
To detect a cycle in a linked list, we can use the Floyd's Tortoise and Hare algorithm. We maintain two pointers, 'slow' and 'fast', where 'slow' moves one step at a time and 'fast' moves two steps at a time. If there is a cycle in the linked list, the two pointers will eventually meet at some point. If there is no cycle, the 'fast' pointer will reach the end of the list.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. In the worst case, both pointers traverse the entire list.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of nodes and pointers.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None


class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if not head or not head.next:
            return False

        slow = head
        fast = head.next

        while slow != fast:
            if not fast or not fast.next:
                return False
            slow = slow.next
            fast = fast.next.next

        return True
```

42. 06_Linked_List/08_Find_The_Duplicate_Number/0287-find-the-duplicate-number.py

```
"""
Problem: LeetCode 287 - Find the Duplicate Number

Key Idea:
To find the duplicate number in an array, we can treat the array as a linked list where each value points to the next value in the array. This problem is then reduced to finding the cycle in the linked list. We use the Floyd's Tortoise and Hare algorithm to detect the cycle. Once the cycle is detected, we find the entrance of the cycle, which represents the duplicate number.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of elements in the array. The Floyd's Tortoise and Hare algorithm takes linear time to detect the cycle.

Space Complexity:
The space complexity is O(1), as no extra space is used other than a few variables to keep track of indices and values.
"""


class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        slow = nums[0]
        fast = nums[0]

        # Move slow and fast pointers
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        # Find the entrance of the cycle
        slow = nums[0]
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]

        return slow
```

43. 06_Linked_List/09_LRU_Cache/0146-lru-cache.py

```
"""
Problem: LeetCode 146 - LRU Cache

Key Idea:
To implement an LRU (Least Recently Used) cache, we can use a combination of a dictionary (to store key-value pairs) and a doubly linked list (to maintain the order of usage). The dictionary allows for quick access to values, and the doubly linked list helps in efficient removal and addition of elements. When a key is accessed or a new key is added, we update its position in the linked list. When the cache is full, we remove the least recently used item from the tail of the linked list.

Time Complexity:
1. The 'get' operation takes O(1) time, as dictionary access is constant time.
2. The 'put' operation takes O(1) time, as dictionary insertion and removal are constant time.

Space Complexity:
The space complexity is O(capacity), as we store key-value pairs in the dictionary and maintain a doubly linked list for the same number of elements.
"""


class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}
        self.order = DoublyLinkedList()

    def get(self, key: int) -> int:
        if key in self.cache:
            self.order.move_to_front(key)
            return self.cache[key]
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self.cache[key] = value
            self.order.move_to_front(key)
        else:
            if len(self.cache) >= self.capacity:
                removed_key = self.order.remove_last()
                del self.cache[removed_key]
            self.cache[key] = value
            self.order.add_to_front(key)


class DoublyLinkedList:
    def __init__(self):
        self.head = ListNode()
        self.tail = ListNode()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.nodes = {}

    def add_to_front(self, key):
        node = ListNode(key)
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node
        self.nodes[key] = node
```

```
    def move_to_front(self, key):
        node = self.nodes[key]
        node.prev.next = node.next
        node.next.prev = node.prev
        self.add_to_front(key)

    def remove_last(self):
        node = self.tail.prev
        node.prev.next = self.tail
        self.tail.prev = node.prev
        del self.nodes[node.key]
        return node.key


class ListNode:
    def __init__(self, key=None):
        self.key = key
        self.prev = None
        self.next = None


"""
from collections import OrderedDict

class LRUCache:

    def __init__(self, capacity: int):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key in self.cache:
            self.cache.move_to_end(key)
            return self.cache[key]
        else:
            return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self.cache.move_to_end(key)
        elif self.capacity <= 0:
            _ = self.cache.popitem(False)
        else:
            self.capacity = max(0, self.capacity - 1)
        self.cache[key] = value
"""

# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

44. 06_Linked_List/10_Merge_K_Sorted_Lists/0023-merge-k-sorted-lists.py

```
"""
Problem: LeetCode 121 - Merge k Sorted Lists

Key Idea:
To merge k sorted linked lists, we can use a min-heap (priority queue) to keep track of the smallest element from each list. We initially add the first element from each list to the heap. Then, in each iteration, we pop the smallest element from the heap and add it to the merged result. If the popped element has a next element in its original list, we add that next element to the heap. We continue this process until the heap is empty.

Time Complexity:
The time complexity of this solution is O(N log k), where N is the total number of elements in all k linked lists. The heap operations take logarithmic time, and we perform these operations for each element.

Space Complexity:
The space complexity is O(k), as the heap can store at most k elements at a time.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next

import heapq


class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        min_heap = []
        for i, l in enumerate(lists):
            if l:
                heapq.heappush(min_heap, (l.val, i))

        dummy = ListNode()
        current = dummy

        while min_heap:
            val, idx = heapq.heappop(min_heap)
            current.next = ListNode(val)
            current = current.next
            if lists[idx].next:
                heapq.heappush(min_heap, (lists[idx].next.val, idx))
                lists[idx] = lists[idx].next

        return dummy.next
```

45. 06_Linked_List/11_Reverse_Nodes_In_K_Group/0025-reverse-nodes-in-k-group.py

```
"""
Problem: LeetCode 25 - Reverse Nodes in k-Group

Key Idea:
To reverse nodes in k-group, we can use a recursive approach. We traverse the linked list in groups of k nodes, reversing each group. For each group, we maintain pointers to the group's first node ('start') and the group's last node ('end'). We reverse the group in-place and connect the previous group's 'end' to the reversed group's 'start'. We then recursively reverse the remaining part of the linked list.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the linked list. We process each node exactly once in groups of k nodes.

Space Complexity:
The space complexity is O(k), as we use a constant amount of extra space for the pointers and variables during the recursion.
"""

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next


class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        if not head or k == 1:
            return head

        # Count the number of nodes in the list
        count = 0
        current = head
        while current:
            count += 1
            current = current.next

        if count < k:
            return head

        # Reverse the first k nodes
        prev, current = None, head
        for _ in range(k):
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node

        # Recursively reverse the remaining part of the list
        head.next = self.reverseKGroup(current, k)

        return prev
```

46. 07_Trees/01_Invert_Binary_Tree/0226-invert-binary-tree.py

```
"""
Problem: LeetCode 226 - Invert Binary Tree

Key Idea:
To invert a binary tree, we can use a recursive approach. For each node, we swap its left and right subtrees,
and then recursively invert the left and right subtrees.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each
node once to perform the inversion.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack
can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if not root:
            return None

        # Swap left and right subtrees
        root.left, root.right = root.right, root.left

        # Recursively invert left and right subtrees
        self.invertTree(root.left)
        self.invertTree(root.right)

        return root
```

47. 07_Trees/02_Maximum_Depth_of_Binary_Tree/0104-maximum-depth-of-binary-tree.py

```
"""
Problem: LeetCode 104 - Maximum Depth of Binary Tree

Key Idea:
To find the maximum depth of a binary tree, we can use a recursive approach. For each node, the maximum depth is the maximum of the depths of its left and right subtrees, plus one. We start from the root and recursively calculate the maximum depth for each subtree.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once to calculate its depth.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root:
            return 0

        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        return max(left_depth, right_depth) + 1
```

48. 07_Trees/03_Diameter_of_Binary_Tree/0543-diameter-of-binary-tree.py

```
"""
Problem: LeetCode 543 - Diameter of Binary Tree

Key Idea:
To find the diameter of a binary tree (the length of the longest path between any two nodes), we can use a recursive approach. For each node, the longest path passes either through the node or doesn't. The diameter is the maximum of three values: the diameter of the left subtree, the diameter of the right subtree, and the sum of the heights of the left and right subtrees (if the path passes through the node).

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once to calculate the diameter.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        def height(node):
            if not node:
                return 0
            left_height = height(node.left)
            right_height = height(node.right)
            self.diameter = max(self.diameter, left_height + right_height)
            return max(left_height, right_height) + 1

        self.diameter = 0
        height(root)
        return self.diameter
```

49. 07_Trees/04_Balanced_Binary_Tree/0110-balanced-binary-tree.py

```
"""
Problem: LeetCode 110 - Balanced Binary Tree

Key Idea:
To check if a binary tree is balanced, we can use a recursive approach. For each node, we calculate the height of its left and right subtrees. If the difference in heights is greater than 1, the tree is not balanced. We continue this process for all nodes, recursively checking each subtree.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once to calculate its height.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def height(node):
            if not node:
                return 0
            left_height = height(node.left)
            right_height = height(node.right)
            if abs(left_height - right_height) > 1:
                return float("inf")  # Indicate imbalance
            return max(left_height, right_height) + 1

        return height(root) != float("inf")
```

50. 07_Trees/05_Same_Tree/0100-same-tree.py

```
"""
Problem: LeetCode 100 - Same Tree

Key Idea:
To determine if two binary trees are the same, we can use a recursive approach. For each pair of corresponding nodes, we compare their values and recursively check the left and right subtrees. If the values are equal and the left and right subtrees are also equal, then the trees are the same.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once to compare its values.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if not p and not q:
            return True
        if not p or not q or p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

51. 07_Trees/06_Subtree_of_Another_Tree/0572-subtree-of-another-tree.py

```
"""
Problem: LeetCode 57 - Subtree of Another Tree

Key Idea:
To check if one binary tree is a subtree of another, we can use a recursive approach. For each node in the main tree, we check if the current subtree rooted at that node is equal to the given subtree. If not, we recursively check the left and right subtrees.

Time Complexity:
The time complexity of this solution is O(m * n), where m is the number of nodes in the main tree and n is the number of nodes in the given subtree. In the worst case, we might need to traverse the entire main tree to find a matching subtree.

Space Complexity:
The space complexity is O(h), where h is the height of the main tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def isSubtree(self, s: TreeNode, t: TreeNode) -> bool:
        if not s:
            return False
        if self.isSameTree(s, t):
            return True
        return self.isSubtree(s.left, t) or self.isSubtree(s.right, t)

    def isSameTree(self, p, q):
        if not p and not q:
            return True
        if not p or not q or p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

52. 07_Trees/07_Lowest_Common_Ancestor_of_a_Binary_Search_Tree/0235-lowest-common-ancestor-of-a-binary-search-tree.py

```
"""
Problem: LeetCode 235 - Lowest Common Ancestor of a Binary Search Tree

Key Idea:
To find the lowest common ancestor (LCA) of two nodes in a binary search tree (BST), we can use a recursive approach. We compare the values of the two nodes with the current node's value. If both nodes are in the left subtree, we move to the left child. If both nodes are in the right subtree, we move to the right child. If one node is in the left subtree and the other is in the right subtree, we've found the LCA.

Time Complexity:
The time complexity of this solution is O(h), where h is the height of the BST. In the worst case, we might need to traverse the height of the tree to find the LCA.

Space Complexity:
The space complexity is O(h), where h is the height of the BST. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None


class Solution:
    def lowestCommonAncestor(
        self, root: TreeNode, p: TreeNode, q: TreeNode
    ) -> TreeNode:
        if root.val > p.val and root.val > q.val:
            return self.lowestCommonAncestor(root.left, p, q)
        elif root.val < p.val and root.val < q.val:
            return self.lowestCommonAncestor(root.right, p, q)
        else:
            return root
```

53. 07_Trees/08_Binary_Tree_Level_Order_Traversal/0102-binary-tree-level-order-traversal.py

```
"""
Problem: LeetCode 102 - Binary Tree Level Order Traversal

Key Idea:
To perform level order traversal of a binary tree, we can use a breadth-first search (BFS) approach. We start with the root node, and in each iteration, we process all nodes at the current level before moving to the next level. We use a queue to keep track of nodes at each level.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once during the BFS traversal.

Space Complexity:
The space complexity is O(w), where w is the maximum width of the binary tree (number of nodes in the widest level). In the worst case, the queue can hold all nodes in a single level.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []

        result = []
        queue = [root]

        while queue:
            level = []
            next_level = []

            for node in queue:
                level.append(node.val)
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)

            result.append(level)
            queue = next_level

        return result
```

54. 07_Trees/09_Binary_Tree_Right_Side_View/0199-binary-tree-right-side-view.py

```
"""
Problem: LeetCode 199 - Binary Tree Right Side View

Key Idea:
To obtain the right side view of a binary tree, we can perform a level order traversal using a breadth-first search (BFS) approach. For each level, we add the last node's value to the result list. This way, we capture the rightmost nodes at each level.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once during the BFS traversal.

Space Complexity:
The space complexity is O(w), where w is the maximum width of the binary tree (number of nodes in the widest level). In the worst case, the queue can hold all nodes in a single level.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        result = []
        queue = [root]

        while queue:
            level_size = len(queue)

            for i in range(level_size):
                node = queue.pop(0)
                if i == level_size - 1:
                    result.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

        return result
```

55. 07_Trees/10_Count_Good_Nodes_In_Binary_Tree/1448-count-good-nodes-in-binary-tree.py

```
"""
Problem: LeetCode 1448 - Count Good Nodes in Binary Tree

Key Idea:
To count the number of good nodes in a binary tree, we can use a recursive depth-first search (DFS) approach. For each node, we keep track of the maximum value encountered on the path from the root to the current node. If the value of the current node is greater than or equal to the maximum value on the path, it is a good node. We increment the count and continue the DFS for the left and right subtrees.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once during the DFS traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        def dfs(node, max_val):
            if not node:
                return 0

            if node.val >= max_val:
                max_val = node.val
                count = 1
            else:
                count = 0

            count += dfs(node.left, max_val)
            count += dfs(node.right, max_val)

            return count

        return dfs(root, float("-inf"))
```

56. 07_Trees/11_Validate_Binary_Search_Tree/0098-validate-binary-search-tree.py

```
"""
Problem: LeetCode 98 - Validate Binary Search Tree

Key Idea:
To validate if a binary tree is a valid binary search tree (BST), we can perform an in-order traversal and check if the values are in ascending order. During the in-order traversal, we keep track of the previously visited node's value and compare it with the current node's value.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once during the in-order traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        def inorder_traversal(node, prev):
            if not node:
                return True

            if not inorder_traversal(node.left, prev):
                return False

            if prev[0] is not None and node.val <= prev[0]:
                return False
            prev[0] = node.val

            return inorder_traversal(node.right, prev)

        prev = [None]
        return inorder_traversal(root, prev)
```

57. 07_Trees/12_Kth_Smallest_Element_In_a_BST/0230-kth-smallest-element-in-a-bst.py

```
"""
Problem: LeetCode 230 - Kth Smallest Element in a BST

Key Idea:
To find the kth smallest element in a binary search tree (BST), we can perform an in-order traversal and keep track of the count of visited nodes. When the count reaches k, we've found the kth smallest element.

Time Complexity:
The time complexity of this solution is O(h + k), where h is the height of the binary tree. In the average case, when the tree is balanced, the height is O(log n), where n is the number of nodes in the tree. In the worst case, when the tree is skewed, the height is O(n). The additional factor of k is due to the in-order traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        def inorder_traversal(node):
            if not node:
                return []

            left = inorder_traversal(node.left)
            right = inorder_traversal(node.right)

            return left + [node.val] + right

        inorder_values = inorder_traversal(root)
        return inorder_values[k - 1]
```

58. 07_Trees/13_Construct_Binary_Tree_from_Preorder_and_Inorder_Traversal/0105-construct-binary-tree-from-preorder-and-inorder-traversal.py

```
"""
Problem: LeetCode 105 - Construct Binary Tree from Preorder and Inorder Traversal

Key Idea:
To construct a binary tree from its preorder and inorder traversals, we can use a recursive approach. The first
element in the preorder list is the root of the current subtree. We locate its position in the inorder list to
determine the left and right subtrees. We recursively construct the left and right subtrees for each subtree.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each
node once during the construction process.

Space Complexity:
The space complexity is O(n), where n is the number of nodes in the binary tree. In the worst case, the
recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> TreeNode:
        if not preorder or not inorder:
            return None

        root_val = preorder.pop(0)
        root = TreeNode(root_val)
        root_index = inorder.index(root_val)

        root.left = self.buildTree(preorder, inorder[:root_index])
        root.right = self.buildTree(preorder, inorder[root_index + 1 :])

        return root
```

59. 07_Trees/14_Binary_Tree_Maximum_Path_Sum/0124-binary-tree-maximum-path-sum.py

```
"""
Problem: LeetCode 124 - Binary Tree Maximum Path Sum

Key Idea:
To find the maximum path sum in a binary tree, we can use a recursive approach. For each node, we calculate the maximum path sum that includes that node. This can be either the node's value itself or the value plus the maximum path sum from its left and right subtrees. We update the global maximum as we traverse the tree.

Time Complexity:
The time complexity of this solution is O(n), where n is the number of nodes in the binary tree. We visit each node once during the traversal.

Space Complexity:
The space complexity is O(h), where h is the height of the binary tree. In the worst case, the recursion stack can go as deep as the height of the tree.
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right


class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        def maxPathSumHelper(node):
            if not node:
                return 0

            left_sum = max(0, maxPathSumHelper(node.left))
            right_sum = max(0, maxPathSumHelper(node.right))

            self.max_sum = max(self.max_sum, left_sum + right_sum + node.val)

            return max(left_sum, right_sum) + node.val

        self.max_sum = float("-inf")
        maxPathSumHelper(root)

        return self.max_sum
```

60. 07_Trees/15_Serialize_and_Deserialize_Binary_Tree/0297-serialize-and-deserialize-binary-tree.py

```
"""
Problem: LeetCode 297 - Serialize and Deserialize Binary Tree

Key Idea:
To serialize a binary tree, we can perform a preorder traversal and serialize the nodes into a string. When deserializing, we split the string into a list of values and reconstruct the binary tree using a recursive approach.

Time Complexity:
- Serialization: The time complexity of serializing the binary tree is O(n), where n is the number of nodes in the tree. We visit each node once during the traversal.
- Deserialization: The time complexity of deserializing the binary tree is also O(n), as we process each value in the list exactly once.

Space Complexity:
- Serialization: The space complexity for serializing is O(n), where n is the number of nodes in the tree. This is due to the space required to store the serialized string.
- Deserialization: The space complexity for deserializing is O(n), where n is the number of nodes in the tree. This is due to the space required for the recursion stack during deserialization.
"""

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None


class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """

        def preorder(node):
            if not node:
                return "None,"
            return str(node.val) + "," + preorder(node.left) + preorder(node.right)

        return preorder(root)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """

        def build_tree(values):
            if values[0] == "None":
                values.pop(0)
                return None

            root = TreeNode(int(values.pop(0)))
            root.left = build_tree(values)
            root.right = build_tree(values)
```

```
            return root

        values = data.split(",")
        return build_tree(values[:-1])


# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))
```

61. 08_Tries/01_Implement_Trie_Prefix_Tree/0208-implement-trie-prefix-tree.py

```
"""
Problem: LeetCode 208 - Implement Trie (Prefix Tree)

Key Idea:
To implement a Trie (prefix tree), we create a TrieNode class that represents each node in the trie. Each node contains a dictionary that maps characters to child nodes. We start with an empty root node and add words by traversing the characters and creating nodes as needed.

Time Complexity:
- Insertion: The time complexity of inserting a word into the trie is O(m), where m is the length of the word.
- Search: The time complexity of searching for a word in the trie is O(m), where m is the length of the word.
- StartsWith: The time complexity of checking if there is any word in the trie that starts with a given prefix is O(m), where m is the length of the prefix.

Space Complexity:
- The space complexity of the trie is O(n * m), where n is the number of words in the trie and m is the average length of the words. This is due to the space required to store the trie nodes and the characters in the words.
"""


class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def startsWith(self, prefix: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True


# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

62. 08_Tries/02_Design_Add_and_Search_Words_Data_Structure/0211-design-add-and-search-words-data-structure.py

```
"""
Problem: LeetCode 211 - Design Add and Search Words Data Structure

Key Idea:
To design a data structure that supports adding and searching words, we can use a Trie (prefix tree) with a special character '.' to represent any character. When searching, we traverse the Trie and recursively search in all child nodes for matching characters or '.'.

Time Complexity:
- Insertion: The time complexity of adding a word to the Trie is O(m), where m is the length of the word.
- Search: The time complexity of searching for a word in the Trie is O(m), where m is the length of the word.

Space Complexity:
- The space complexity of the Trie is O(n * m), where n is the number of words in the Trie and m is the average length of the words. This is due to the space required to store the Trie nodes and the characters in the words.
"""


class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class WordDictionary:
    def __init__(self):
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word: str) -> bool:
        def search_in_node(node, word):
            for i, char in enumerate(word):
                if char not in node.children:
                    if char == ".":
                        for child in node.children:
                            if search_in_node(node.children[child], word[i + 1 :]):
                                return True
                    return False
                else:
                    node = node.children[char]
            return node.is_end

        return search_in_node(self.root, word)


# Your WordDictionary object will be instantiated and called as such:
# obj = WordDictionary()
# obj.addWord(word)
# param_2 = obj.search(word)

# class WordDictionary:

#     def __init__(self):
```

```
#         self.word_set = set()

#     def addWord(self, word: str) -> None:
#         self.word_set.add(word)
#         for i in range(len(word)):
#             # Add all possible variations with a '.' in each position
#             self.word_set.add(word[:i] + '.' + word[i + 1:])

#     def search(self, word: str) -> bool:
#         if word in self.word_set:
#             return True

#         # Check if the word contains a '.'
#         if '.' not in word:
#             return False

#         # Split the word into two parts at the first occurrence of '.'
#         first_part, rest_part = word.split('.', 1)

#         # Iterate over lowercase letters and create variations to search
#         for char in 'abcdefghijklmnopqrstuvwxyz':
#             new_word = first_part + char + rest_part
#             if new_word in self.word_set:
#                 return True

#         return False
```

63. 08_Tries/03_Word_Search_II/0212-word-search-ii.py

```
"""
Problem: LeetCode 212 - Word Search II

Key Idea:
To find all the words from a given list that can be formed by a 2D board of characters, we can use a Trie (prefix tree) to efficiently search for words while traversing the board. We start by building a Trie from the given list of words. Then, we perform a depth-first search (DFS) on the board, checking if the current path forms a valid prefix in the Trie. If it does, we continue the DFS until we find words or reach dead ends.

Time Complexity:
- Building Trie: The time complexity of building the Trie is O(n * m), where n is the number of words in the list and m is the average length of the words.
- DFS: The time complexity of the DFS traversal on the board is O(n * m * 4^k), where n and m are the dimensions of the board and k is the maximum length of the words.

Space Complexity:
- The space complexity of the Trie is O(n * m), where n is the number of words in the list and m is the average length of the words.
- The space complexity of the DFS recursion stack is O(n * m) as well, due to the potential depth of the recursion.
"""

from collections import Counter
from itertools import chain, product
from typing import List


class TrieNode:
    def __init__(self):
        self.children = {}  # Store child nodes for each character
        self.refcnt = 0  # Count of references to this node
        self.is_word = False  # Flag to indicate if a complete word ends at this node
        self.is_rev = False  # Flag to indicate if a word should be reversed


class Trie:
    def __init__(self):
        self.root = TrieNode()  # Initialize the root of the trie

    def insert(self, word, rev):
        node = self.root
        for c in word:
            node = node.children.setdefault(c, TrieNode())
            node.refcnt += 1
        node.is_word = True
        node.is_rev = rev

    def remove(self, word):
        node = self.root
        for i, c in enumerate(word):
            parent = node
            node = node.children[c]

            if node.refcnt == 1:
                path = [(parent, c)]
                for c in word[i + 1 :]:
                    path.append((node, c))
                    node = node.children[c]
                for parent, c in path:
                    parent.children.pop(c)
```

```
                return
            node.refcnt -= 1
        node.is_word = False


class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        res = []
        n, m = len(board), len(board[0])
        trie = Trie()

        # Count characters on the board
        boardcnt = Counter(chain(*board))

        # Insert words into trie with appropriate orientation
        for w, wrdcnt in ((w, Counter(w)) for w in words):
            if any(wrdcnt[c] > boardcnt[c] for c in wrdcnt):
                continue  # Skip if the word cannot be formed from the board
            if wrdcnt[w[0]] < wrdcnt[w[-1]]:
                trie.insert(w, False)
            else:
                trie.insert(w[::-1], True)

        def dfs(r, c, parent) -> None:
            if not (node := parent.children.get(board[r][c])):
                return
            path.append(board[r][c])
            board[r][c] = "#"  # Mark visited cell

            if node.is_word:
                word = "".join(path)
                res.append(word[::-1] if node.is_rev else word)
                trie.remove(word)

            # Explore neighboring cells
            if r > 0:
                dfs(r - 1, c, node)
            if r < n - 1:
                dfs(r + 1, c, node)
            if c > 0:
                dfs(r, c - 1, node)
            if c < m - 1:
                dfs(r, c + 1, node)

            board[r][c] = path.pop()  # Backtrack and unmark cell

        path = []
        for r, c in product(range(n), range(m)):
            dfs(r, c, trie.root)
        return res
```

64. 09_Heap_Priority_Queues/01_Kth_Largest_Element_in_a_Stream/0703-kth-largest-element-in-a-stream.py

```
"""
Problem: LeetCode 703 - Kth Largest Element in a Stream

Key Idea:
To find the kth largest element in a stream of integers, we can use a min-heap (priority queue) with a maximum size of k. As new elements arrive, we add them to the min-heap. If the size of the heap exceeds k, we remove the smallest element. The top element of the min-heap will be the kth largest element.

Time Complexity:
- Adding an element: The time complexity of adding an element to the min-heap is O(log k), where k is the maximum size of the heap.
- Retrieving the kth largest element: The time complexity of retrieving the kth largest element (the top of the heap) is O(1).

Space Complexity:
- The space complexity is O(k), where k is the maximum size of the min-heap.
"""

import heapq


class KthLargest:
    def __init__(self, k: int, nums: List[int]):
        self.min_heap = []
        self.k = k

        for num in nums:
            self.add(num)

    def add(self, val: int) -> int:
        heapq.heappush(self.min_heap, val)

        if len(self.min_heap) > self.k:
            heapq.heappop(self.min_heap)

        return self.min_heap[0]


# Your KthLargest object will be instantiated and called as such:
# obj = KthLargest(k, nums)
# param_1 = obj.add(val)
```

65. 09_Heap_Priority_Queues/02_Last_Stone_Weight/1046-last-stone-weight.py

```
"""
Problem: LeetCode 1046 - Last Stone Weight

Key Idea:
To simulate the process of smashing stones, we can use a max-heap (priority queue) to keep track of the stone weights. At each step, we pop the two largest stones from the heap, smash them together, and push the resulting weight back into the heap. We repeat this process until there is only one stone left in the heap, which will be the last stone weight.

Time Complexity:
- Building the max-heap: The time complexity of building the max-heap is O(n), where n is the number of stones.
- Popping and pushing elements: The time complexity of each pop and push operation is O(log n), where n is the number of stones. We perform these operations until there is only one stone left.

Space Complexity:
- The space complexity is O(n), where n is the number of stones, due to the space required to store the max-heap.
"""

import heapq


class Solution:
    def lastStoneWeight(self, stones: List[int]) -> int:
        max_heap = [-stone for stone in stones]  # Use negative values for max-heap

        heapq.heapify(max_heap)

        while len(max_heap) > 1:
            x = -heapq.heappop(max_heap)  # Extract the largest stone
            y = -heapq.heappop(max_heap)  # Extract the second largest stone

            if x != y:
                heapq.heappush(max_heap, -(x - y))  # Push the remaining weight

        return -max_heap[0] if max_heap else 0
```

66. 09_Heap_Priority_Queues/03_K_Closest_Points_to_Origin/0973-k-closest-points-to-origin.py

```
"""
Problem: LeetCode 973 - K Closest Points to Origin

Key Idea:
To find the k closest points to the origin, we can calculate the distance of each point from the origin and use a min-heap (priority queue) to keep track of the k closest points. As we iterate through the points, we push each point into the min-heap. If the size of the heap exceeds k, we remove the farthest point. The remaining points in the heap will be the k closest points.

Time Complexity:
- Calculating distances: The time complexity of calculating the Euclidean distance for each point is O(n), where n is the number of points.
- Pushing and popping elements: The time complexity of each push and pop operation in the min-heap is O(log k), where k is the number of closest points we want.

Space Complexity:
- The space complexity is O(k), where k is the number of closest points we want to find. This is due to the space required to store the min-heap.
"""

import heapq


class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        def distance(point):
            return point[0] ** 2 + point[1] ** 2

        min_heap = [(distance(point), point) for point in points]
        heapq.heapify(min_heap)

        result = []
        for _ in range(k):
            result.append(heapq.heappop(min_heap)[1])

        return result
```

67. 09_Heap_Priority_Queues/04_Kth_Largest_Element_in_an_Array/0215-kth-largest-element-in-an-array.py

```
"""
Problem: LeetCode 215 - Kth Largest Element in an Array

Key Idea:
To find the kth largest element in an array, we can use a min-heap (priority queue) with a maximum size of k. As we iterate through the array, we push elements into the min-heap. If the size of the heap exceeds k, we remove the smallest element. The top element of the min-heap will be the kth largest element.

Time Complexity:
- Building the min-heap: The time complexity of building the min-heap is O(k), where k is the maximum size of the heap.
- Adding elements: The time complexity of adding an element to the min-heap is O(n * log k), where n is the length of the array and k is the maximum size of the heap.

Space Complexity:
- The space complexity is O(k), where k is the maximum size of the min-heap.
"""

import heapq


class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        min_heap = []

        for num in nums:
            heapq.heappush(min_heap, num)
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        return min_heap[0]
```

68. 09_Heap_Priority_Queues/05_Task_Scheduler/0621-task-scheduler.py

```
"""
Problem: LeetCode 621 - Task Scheduler

Key Idea:
To schedule tasks with maximum cooling time, we can use a greedy approach. We first count the frequency of each task and sort them in descending order. We then iterate through the tasks and use a cooldown counter to keep track of the remaining time until the next valid task can be scheduled. During each iteration, we schedule the task with the highest frequency that is not on cooldown. If there are no available tasks, we simply wait.

Time Complexity:
- Counting tasks: The time complexity of counting the frequency of each task is O(n), where n is the number of tasks.
- Sorting tasks: The time complexity of sorting the tasks is O(26 * log 26) = O(1), since there are at most 26 different tasks.
- Iterating through tasks: The time complexity of iterating through the tasks is O(n), where n is the number of tasks.

Space Complexity:
- The space complexity is O(26) = O(1), since there are at most 26 different tasks.
"""

import heapq
from collections import Counter


class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        task_counts = Counter(tasks)
        max_heap = [-count for count in task_counts.values()]
        heapq.heapify(max_heap)

        cooldown = 0
        while max_heap:
            temp = []
            for _ in range(n + 1):
                if max_heap:
                    temp.append(heapq.heappop(max_heap) + 1)

            for count in temp:
                if count < 0:
                    heapq.heappush(max_heap, count)

            if max_heap:
                cooldown += n + 1
            else:
                cooldown += len(temp)

        return cooldown
```

69. 09_Heap_Priority_Queues/06_Design_Twitter/0355-design-twitter.py

```
"""
Problem: LeetCode 355 - Design Twitter

Key Idea:
To design a simplified version of Twitter, we can use a combination of data structures. We maintain a dictionary to map users to their tweets, and a set of followees for each user. When a user posts a tweet, we add it to their tweet list. When a user wants to retrieve their news feed, we merge their own tweets and the tweets of their followees, and then sort them based on timestamps.

Time Complexity:
- Post Tweet: The time complexity of posting a tweet is O(1).
- Get News Feed: The time complexity of retrieving a user's news feed is O(f + t log t), where f is the number of followees and t is the total number of tweets.

Space Complexity:
- The space complexity is O(u + t), where u is the number of users and t is the total number of tweets.
"""

import heapq


class Tweet:
    def __init__(self, tweet_id, timestamp):
        self.tweet_id = tweet_id
        self.timestamp = timestamp


class Twitter:
    def __init__(self):
        self.user_tweets = {}  # User ID -> List of Tweet objects
        self.user_followees = {}  # User ID -> Set of followees
        self.timestamp = 0

    def postTweet(self, userId: int, tweetId: int) -> None:
        self.timestamp += 1
        if userId not in self.user_tweets:
            self.user_tweets[userId] = []
        self.user_tweets[userId].append(Tweet(tweetId, self.timestamp))

    def getNewsFeed(self, userId: int) -> List[int]:
        tweets = []

        if userId in self.user_tweets:
            tweets.extend(self.user_tweets[userId])

        if userId in self.user_followees:
            for followee in self.user_followees[userId]:
                if followee in self.user_tweets:
                    tweets.extend(self.user_tweets[followee])

        tweets.sort(key=lambda x: x.timestamp, reverse=True)
        return [tweet.tweet_id for tweet in tweets[:10]]

    def follow(self, followerId: int, followeeId: int) -> None:
        if followerId != followeeId:
            if followerId not in self.user_followees:
                self.user_followees[followerId] = set()
            self.user_followees[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
```

```
        if (
            followerId in self.user_followees
            and followeeId in self.user_followees[followerId]
        ):
            self.user_followees[followerId].remove(followeeId)


# Your Twitter object will be instantiated and called as such:
# obj = Twitter()
# obj.postTweet(userId,tweetId)
# param_2 = obj.getNewsFeed(userId)
# obj.follow(followerId,followeeId)
# obj.unfollow(followerId,followeeId)
```

70. 09_Heap_Priority_Queues/07_Find_Median_from_Data_Stream/0295-find-median-from-data-stream.py

```
"""
Problem: LeetCode 295 - Find Median from Data Stream

Key Idea:
To find the median from a stream of data, we can use two heaps: a max-heap to store the lower half of the data and a min-heap to store the upper half of the data. The max-heap will contain the smaller elements, and the min-heap will contain the larger elements. To maintain the balance of the heaps, we ensure that the size difference between the two heaps is at most 1. The median will be the average of the top elements of both heaps if the total number of elements is even, or the top element of the larger heap if the total number of elements is odd.

Time Complexity:
- Adding an element: The time complexity of adding an element to the heaps is O(log n), where n is the number of elements.
- Finding the median: The time complexity of finding the median is O(1), as it involves accessing the top elements of the heaps.

Space Complexity:
- The space complexity is O(n), where n is the number of elements in the data stream.
"""

import heapq


class MedianFinder:
    def __init__(self):
        self.min_heap = []  # To store larger elements
        self.max_heap = []  # To store smaller elements

    def addNum(self, num: int) -> None:
        if not self.max_heap or num <= -self.max_heap[0]:
            heapq.heappush(self.max_heap, -num)
        else:
            heapq.heappush(self.min_heap, num)

        # Balance the heaps
        if len(self.max_heap) > len(self.min_heap) + 1:
            heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))
        elif len(self.min_heap) > len(self.max_heap):
            heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))

    def findMedian(self) -> float:
        if len(self.max_heap) == len(self.min_heap):
            return (-self.max_heap[0] + self.min_heap[0]) / 2
        else:
            return -self.max_heap[0]
```

71. 10_Backtracking/01_Subsets/0078-subsets.py

```
"""
Problem: LeetCode 78 - Subsets

Key Idea:
To generate all possible subsets of a given set of distinct integers, we can use a recursive approach. For each element, we have two choices: either include it in the current subset or exclude it. We explore both choices recursively to generate all subsets.

Time Complexity:
- The total number of subsets is 2^n, where n is the number of elements in the input set. Therefore, the time complexity is O(2^n), as we need to generate all possible subsets.

Space Complexity:
- The space complexity is O(n), where n is the number of elements in the input set. This is due to the space required for the recursive call stack.
"""


class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def backtrack(start, subset):
            subsets.append(subset[:])  # Append a copy of the current subset

            for i in range(start, len(nums)):
                subset.append(nums[i])
                backtrack(i + 1, subset)
                subset.pop()  # Backtrack

        subsets = []
        backtrack(0, [])
        return subsets
```

72. 10_Backtracking/02_Combination_Sum/0039-combination-sum.py

```
"""
Problem: LeetCode 39 - Combination Sum

Key Idea:
To find all unique combinations that sum up to a target value, we can use a backtracking approach. Starting from each candidate element, we explore all possible combinations by adding the element to the current combination and recursively searching for the remaining sum. If the sum becomes equal to the target, we add the current combination to the result. This process is repeated for each candidate element.

Time Complexity:
- In the worst case, each candidate element can be used multiple times to reach the target sum. Therefore, the time complexity is O(k * 2^n), where k is the average number of times each element can be used and n is the number of candidate elements.

Space Complexity:
- The space complexity is O(target), as the recursive call stack can go up to the target value.
"""


class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def backtrack(start, target, combination):
            if target == 0:
                result.append(
                    combination[:]
                )  # Append a copy of the current combination
                return

            for i in range(start, len(candidates)):
                if candidates[i] > target:
                    continue  # Skip if the candidate is too large

                combination.append(candidates[i])
                backtrack(i, target - candidates[i], combination)
                combination.pop()  # Backtrack

        result = []
        backtrack(0, target, [])
        return result
```

73. 10_Backtracking/03_Permutations/0046-permutations.py

```
"""
Problem: LeetCode 46 - Permutations

Key Idea:
To generate all permutations of a given list of distinct integers, we can use a backtracking approach. Starting from each element, we explore all possible permutations by swapping the current element with other elements and recursively generating permutations for the remaining elements. Once we reach the end of the list, we add the current permutation to the result.

Time Complexity:
- The total number of permutations is n!, where n is the number of elements in the input list. Therefore, the time complexity is O(n!).

Space Complexity:
- The space complexity is O(n), where n is the number of elements in the input list. This is due to the space required for the recursive call stack.
"""


class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        def backtrack(start):
            if start == len(nums) - 1:
                permutations.append(nums[:])  # Append a copy of the current permutation

            for i in range(start, len(nums)):
                nums[start], nums[i] = nums[i], nums[start]  # Swap elements
                backtrack(start + 1)
                nums[start], nums[i] = nums[i], nums[start]  # Backtrack

        permutations = []
        backtrack(0)
        return permutations
```

74. 10_Backtracking/04_Subsets_II/0090-subsets-ii.py

```
"""
Problem: LeetCode 90 - Subsets II

Key Idea:
To generate all possible subsets of a given list of integers, accounting for duplicates, we can use a backtracking approach. Similar to the previous subset problem, we explore all possible choices for each element: either include it in the current subset or exclude it. To handle duplicates, we skip adding the same element if it has already been processed at the same depth level.

Time Complexity:
- The total number of subsets is still 2^n, but the presence of duplicates may reduce the number of valid subsets. Therefore, the time complexity is O(2^n), as we need to generate all possible subsets.

Space Complexity:
- The space complexity is O(n), where n is the number of elements in the input list. This is due to the space required for the recursive call stack.
"""


class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def backtrack(start, subset):
            subsets.append(subset[:])  # Append a copy of the current subset

            for i in range(start, len(nums)):
                if i > start and nums[i] == nums[i - 1]:
                    continue  # Skip duplicates at the same depth level
                subset.append(nums[i])
                backtrack(i + 1, subset)
                subset.pop()  # Backtrack

        nums.sort()  # Sort the input to handle duplicates
        subsets = []
        backtrack(0, [])
        return subsets
```

75. 10_Backtracking/05_Combination_Sum_II/0040-combination-sum-ii.py

```
"""
Problem: LeetCode 40 -  Combination Sum II

Key Idea:
To find all unique combinations that sum up to a target value, accounting for duplicates, we can use a backtracking approach. Starting from each candidate element, we explore all possible combinations by adding the element to the current combination and recursively searching for the remaining sum. To handle duplicates, we skip adding the same element if it has already been processed at the same depth level.

Time Complexity:
- The total number of subsets is still 2^n, but the presence of duplicates may reduce the number of valid subsets. Therefore, the time complexity is O(2^n), as we need to generate all possible subsets.

Space Complexity:
- The space complexity is O(target), as the recursive call stack can go up to the target value.
"""


class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        def backtrack(start, target, combination):
            if target == 0:
                result.append(
                    combination[:]
                )  # Append a copy of the current combination
                return

            for i in range(start, len(candidates)):
                if i > start and candidates[i] == candidates[i - 1]:
                    continue  # Skip duplicates at the same depth level

                if candidates[i] > target:
                    continue  # Skip if the candidate is too large

                combination.append(candidates[i])
                backtrack(i + 1, target - candidates[i], combination)
                combination.pop()  # Backtrack

        candidates.sort()  # Sort the input to handle duplicates
        result = []
        backtrack(0, target, [])
        return result
```

76. 10_Backtracking/06_Word_Search/0079-word-search.py

```
"""
Problem: LeetCode 79 - Word Search

Key Idea:
To determine if a given word exists in the 2D board, we can perform a depth-first search (DFS) from each cell in the board. At each cell, we check if the current character of the word matches the character in the cell. If it does, we mark the cell as visited, recursively search its neighboring cells, and backtrack if the search is unsuccessful. We repeat this process for all cells in the board.

Time Complexity:
- In the worst case, for each cell, we perform a DFS with a maximum depth of the length of the word, which leads to a time complexity of O(m * n * 4^k), where m and n are the dimensions of the board and k is the length of the word.

Space Complexity:
- The space complexity is O(k), where k is the length of the word, as it represents the maximum depth of the recursive call stack.
"""


class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(row, col, index):
            if index == len(word):
                return True

            if (
                row < 0
                or row >= len(board)
                or col < 0
                or col >= len(board[0])
                or board[row][col] != word[index]
            ):
                return False

            original_char = board[row][col]
            board[row][col] = "#"  # Mark the cell as visited

            found = (
                dfs(row + 1, col, index + 1)
                or dfs(row - 1, col, index + 1)
                or dfs(row, col + 1, index + 1)
                or dfs(row, col - 1, index + 1)
            )

            board[row][col] = original_char  # Backtrack

            return found

        for row in range(len(board)):
            for col in range(len(board[0])):
                if board[row][col] == word[0] and dfs(row, col, 0):
                    return True

        return False
```

77. 10_Backtracking/07_Palindrome_Partitioning/0131-palindrome-partitioning.py

```
"""
Problem: LeetCode 131 - Palindrome Partitioning

Key Idea:
To partition a given string into palindromic substrings, we can use backtracking. Starting from each position in the string, we check if the substring from that position to the end is a palindrome. If it is, we recursively partition the remaining substring and continue the process. We keep track of the current partition in a list and store valid partitions in the result.

Time Complexity:
- In the worst case, for each position in the string, we recursively generate all possible partitions. Therefore, the time complexity is O(2^n), where n is the length of the string.

Space Complexity:
- The space complexity is O(n), where n is the length of the string. This is due to the space required for the recursive call stack and the space for storing the current partition.
"""


class Solution:
    def partition(self, s: str) -> List[List[str]]:
        def is_palindrome(sub):
            return sub == sub[::-1]

        def backtrack(start, partition):
            if start == len(s):
                result.append(partition[:])  # Append a copy of the current partition
                return

            for end in range(start + 1, len(s) + 1):
                sub = s[start:end]
                if is_palindrome(sub):
                    partition.append(sub)
                    backtrack(end, partition)
                    partition.pop()  # Backtrack

        result = []
        backtrack(0, [])
        return result
```

78. 10_Backtracking/08_Letter_Combinations_of_a_Phone_Number/0017-letter-combinations-of-a-phone-number.py

```
"""
Problem: LeetCode 17 - Letter Combinations of a Phone Number

Key Idea:
To generate all possible letter combinations of a phone number, we can use a recursive approach. Starting from each digit of the phone number, we generate combinations by appending each letter corresponding to the digit to the current combinations. We repeat this process for all digits and all possible letters, building up the combinations.

Time Complexity:
- The total number of combinations is 4^n (where n is the number of digits) in the worst case, as each digit maps to up to 4 letters on a phone keypad. Therefore, the time complexity is O(4^n).

Space Complexity:
- The space complexity is O(n), where n is the number of digits, representing the maximum depth of the recursive call stack.
"""


class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:
            return []

        phone_mapping = {
            "2": "abc",
            "3": "def",
            "4": "ghi",
            "5": "jkl",
            "6": "mno",
            "7": "pqrs",
            "8": "tuv",
            "9": "wxyz",
        }

        def backtrack(index, combination):
            if index == len(digits):
                combinations.append(combination)
                return

            digit = digits[index]
            letters = phone_mapping[digit]

            for letter in letters:
                backtrack(index + 1, combination + letter)

        combinations = []
        backtrack(0, "")
        return combinations
```

79. 10_Backtracking/09_N_Queens/0051-n-queens.py

```
"""
Problem: LeetCode 51 - N-Queens

Key Idea:
To solve the N-Queens problem, we can use backtracking. Starting from each row, we try placing a queen in each column of that row and recursively move on to the next row. If a valid placement is found, we continue the process. We keep track of the board state and the positions of the queens to avoid conflicts.

Time Complexity:
- In the worst case, we explore all possible combinations of queen placements, leading to a time complexity of O(N!), where N is the size of the board (number of rows/columns).

Space Complexity:
- The space complexity is O(N^2), as we need to store the board state and the positions of the queens.
"""


class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        def is_safe(row, col):
            # Check for conflicts with previous rows
            for prev_row in range(row):
                if board[prev_row][col] == "Q":
                    return False
                if (
                    col - (row - prev_row) >= 0
                    and board[prev_row][col - (row - prev_row)] == "Q"
                ):
                    return False
                if (
                    col + (row - prev_row) < n
                    and board[prev_row][col + (row - prev_row)] == "Q"
                ):
                    return False
            return True

        def place_queen(row):
            if row == n:
                result.append(["".join(row) for row in board])
                return

            for col in range(n):
                if is_safe(row, col):
                    board[row][col] = "Q"
                    place_queen(row + 1)
                    board[row][col] = "."

        board = [["." for _ in range(n)] for _ in range(n)]
        result = []
        place_queen(0)
        return result


# class Solution:
#     def solveNQueens(self, n: int) -> List[List[str]]:
#         result = []  # List to store solutions
#         board = [['.'] * n for _ in range(n)]  # Chessboard representation
#         left_diagonal = [False] * (2 * n - 1)  # Left diagonals availability
#         right_diagonal = [False] * (2 * n - 1)  # Right diagonals availability
#         column = [False] * n  # Columns availability
```

```
#         def backtrack(row):
#             if row == n:
#                 solution = ["".join(row) for row in board]  # Convert the board to a solution format
#                 result.append(solution)
#                 return

#             for col in range(n):
#                 # Check if placing a queen in the current position is valid
#                 if column[col] or left_diagonal[row - col] or right_diagonal[row + col]:
#                     continue

#                 # Place a queen and mark unavailable positions
#                 board[row][col] = 'Q'
#                 column[col] = left_diagonal[row - col] = right_diagonal[row + col] = True

#                 # Move to the next row
#                 backtrack(row + 1)

#                 # Backtrack: Reset the board and availability
#                 board[row][col] = '.'
#                 column[col] = left_diagonal[row - col] = right_diagonal[row + col] = False

#         backtrack(0)  # Start the backtracking process from the first row
#         return result
```

80. 11_Graphs/01_Number_of_Islands/0200-number-of-islands.py

```
"""
Problem: LeetCode 200 - Number of Islands

Key Idea:
The problem is to count the number of islands in a 2D grid where '1' represents land and '0' represents water. We can solve this problem using Depth-First Search (DFS) algorithm. For each cell that contains '1', we perform DFS to explore all adjacent land cells and mark them as visited by changing their value to '0'. This way, we count each connected component of '1's as a separate island.

Time Complexity:
- In the worst case, we visit each cell in the grid once. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the maximum space required for the call stack during DFS traversal.
"""


class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        count = 0

        def dfs(row, col):
            if (
                row < 0
                or row >= rows
                or col < 0
                or col >= cols
                or grid[row][col] == "0"
            ):
                return

            grid[row][col] = "0"  # Mark the cell as visited
            directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

            for dr, dc in directions:
                dfs(row + dr, col + dc)

        for i in range(rows):
            for j in range(cols):
                if grid[i][j] == "1":
                    count += 1
                    dfs(i, j)

        return count
```

81. 11_Graphs/02_Clone_Graph/0133-clone-graph.py

```
"""
Problem: LeetCode 133 - Clone Graph

Key Idea:
The problem is to clone an undirected graph. We can solve this using Depth-First Search (DFS) or Breadth-First Search (BFS). Here, I'm use DFS to traverse the original graph and create a new graph.

Time Complexity:
- In the worst case, we visit each node and each edge in the graph exactly once. Therefore, the time complexity is O(V + E), where V is the number of nodes (vertices) and E is the number of edges in the graph.

Space Complexity:
- The space complexity is O(V), where V is the number of nodes in the graph. This is the maximum space required for the recursive call stack during DFS traversal, as well as for the hash map used to store the cloned nodes.
"""

"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""


class Solution:
    def cloneGraph(self, node: "Node") -> "Node":
        if not node:
            return None

        visited = {}  # Dictionary to store the cloned nodes

        def dfs(original_node):
            if original_node in visited:
                return visited[original_node]

            new_node = Node(original_node.val)
            visited[original_node] = new_node

            for neighbor in original_node.neighbors:
                new_neighbor = dfs(neighbor)
                new_node.neighbors.append(new_neighbor)

            return new_node

        return dfs(node)
```

82. 11_Graphs/03_Max_Area_of_Island/0695-max-area-of-island.py

```
"""
Problem: LeetCode 695 - Max Area of Island

Key Idea:
The problem is to find the maximum area of an island in a grid where 1 represents land and 0 represents water. We can solve this using Depth-First Search (DFS) to traverse each cell of the grid and identify connected land cells forming an island.

Time Complexity:
- In the worst case, we visit each cell in the grid exactly once. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the maximum space required for the recursive call stack during DFS traversal.
"""


class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        def dfs(row, col):
            if (
                row < 0
                or row >= len(grid)
                or col < 0
                or col >= len(grid[0])
                or grid[row][col] == 0
            ):
                return 0

            grid[row][col] = 0  # Mark as visited
            area = 1

            area += dfs(row + 1, col)  # Check down
            area += dfs(row - 1, col)  # Check up
            area += dfs(row, col + 1)  # Check right
            area += dfs(row, col - 1)  # Check left

            return area

        max_area = 0
        for row in range(len(grid)):
            for col in range(len(grid[0])):
                if grid[row][col] == 1:
                    max_area = max(max_area, dfs(row, col))

        return max_area
```

83. 11_Graphs/04_Pacific_Atlantic_Water_Flow/0417-pacific-atlantic-water-flow.py

```
"""
Problem: LeetCode 417 - Pacific Atlantic Water Flow

Key Idea:
The problem is to find the cells in a matrix where water can flow from both the Pacific Ocean and the Atlantic Ocean. We can solve this using Depth-First Search (DFS) starting from the ocean borders. Each cell that can be reached from both oceans will be added to the final result.

Time Complexity:
- We perform two separate DFS traversals from the ocean borders, one for the Pacific Ocean and one for the Atlantic Ocean. In the worst case, we visit each cell in the matrix exactly once during each DFS traversal.
- Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix. This is the maximum space required for the recursive call stack during DFS traversal.
"""


class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        if not heights:
            return []

        rows, cols = len(heights), len(heights[0])
        pacific_reachable = set()
        atlantic_reachable = set()

        def dfs(r, c, reachable):
            reachable.add((r, c))
            for dr, dc in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                nr, nc = r + dr, c + dc
                if (
                    0 <= nr < rows
                    and 0 <= nc < cols
                    and (nr, nc) not in reachable
                    and heights[nr][nc] >= heights[r][c]
                ):
                    dfs(nr, nc, reachable)

        for r in range(rows):
            dfs(r, 0, pacific_reachable)
            dfs(r, cols - 1, atlantic_reachable)

        for c in range(cols):
            dfs(0, c, pacific_reachable)
            dfs(rows - 1, c, atlantic_reachable)

        return list(pacific_reachable & atlantic_reachable)
```

84. 11_Graphs/05_Surrounded_Regions/0130-surrounded-regions.py

```
"""
Problem: LeetCode 130 - Surrounded Regions

Key Idea:
The problem is to capture 'O' cells that are not surrounded by 'X' cells in a given board. To solve this, we can use Depth-First Search (DFS) starting from the boundary 'O' cells. All the 'O' cells that are reachable from the boundary will be retained, and the rest will be changed to 'X'.

Time Complexity:
- In the worst case, we visit each cell in the board exactly once. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the board.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the board. This is the maximum space required for the recursive call stack during DFS traversal.
"""


class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """

        def dfs(row, col):
            if (
                row < 0
                or row >= len(board)
                or col < 0
                or col >= len(board[0])
                or board[row][col] != "O"
            ):
                return

            board[row][col] = "E"  # Mark as visited but not surrounded

            # Check adjacent cells
            dfs(row + 1, col)  # Check down
            dfs(row - 1, col)  # Check up
            dfs(row, col + 1)  # Check right
            dfs(row, col - 1)  # Check left

        # Traverse the boundary and mark connected 'O' cells as 'E'
        for row in range(len(board)):
            if board[row][0] == "O":
                dfs(row, 0)
            if board[row][len(board[0]) - 1] == "O":
                dfs(row, len(board[0]) - 1)

        for col in range(len(board[0])):
            if board[0][col] == "O":
                dfs(0, col)
            if board[len(board) - 1][col] == "O":
                dfs(len(board) - 1, col)

        # Mark internal 'O' cells as 'X' and restore 'E' cells to 'O'
        for row in range(len(board)):
            for col in range(len(board[0])):
                if board[row][col] == "O":
                    board[row][col] = "X"
```

```
elif board[row][col] == "E":
    board[row][col] = "O"
```

85. 11_Graphs/06_Rotting_Oranges/0994-rotting-oranges.py

```
"""
Problem: LeetCode 994 - Rotting Oranges

Key Idea:
The problem is to determine the minimum time needed for all oranges to become rotten, considering that rotten oranges can also infect adjacent fresh oranges in each minute. We can model this problem using Breadth-First Search (BFS), where each minute corresponds to a level of the BFS traversal.

Time Complexity:
- In the worst case, all cells are fresh oranges, and each cell can be visited at most once. Therefore, the BFS traversal has a time complexity of O(m * n), where m is the number of rows and n is the number of columns in the grid.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the maximum space required for the BFS queue and visited set.
"""


class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        if not grid:
            return -1

        rows, cols = len(grid), len(grid[0])
        fresh_count = 0  # Count of fresh oranges
        rotten = deque()  # Queue to store coordinates of rotten oranges
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Possible adjacent cells

        # Initialize the queue with coordinates of rotten oranges
        for row in range(rows):
            for col in range(cols):
                if grid[row][col] == 2:
                    rotten.append((row, col))
                elif grid[row][col] == 1:
                    fresh_count += 1

        minutes = 0  # Timer

        while rotten:
            level_size = len(rotten)

            for _ in range(level_size):
                row, col = rotten.popleft()

                for dr, dc in directions:
                    new_row, new_col = row + dr, col + dc

                    # Check if the new cell is within bounds and has a fresh orange
                    if (
                        0 <= new_row < rows
                        and 0 <= new_col < cols
                        and grid[new_row][new_col] == 1
                    ):
                        grid[new_row][new_col] = 2  # Infect the fresh orange
                        fresh_count -= 1
                        rotten.append((new_row, new_col))

            if rotten:
                minutes += 1
```

```
# If there are fresh oranges left, return -1; otherwise, return the elapsed minutes
return minutes if fresh_count == 0 else -1
```

86. 11_Graphs/07_Walls_and_Gates/0286-walls-and-gates.py

```
"""
Problem: LeetCode 286 - Walls and Gates

Key Idea:
The problem is to fill each empty room (represented by INF) with the distance to the nearest gate. This can be approached using Breadth-First Search (BFS), where the gates are the starting points and the empty rooms are visited layer by layer.

Time Complexity:
- Each cell is visited at most once, and BFS takes linear time proportional to the number of cells in the grid. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid.

Space Complexity:
- The space complexity is O(m * n), where m is the number of rows and n is the number of columns in the grid. This is the maximum space required for the BFS queue and visited set.
"""


class Solution:
    def wallsAndGates(self, rooms: List[List[int]]) -> None:
        if not rooms:
            return

        rows, cols = len(rooms), len(rooms[0])
        gates = [(i, j) for i in range(rows) for j in range(cols) if rooms[i][j] == 0]
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

        for gate_row, gate_col in gates:
            visited = set()  # To track visited cells in BFS
            queue = deque([(gate_row, gate_col, 0)])

            while queue:
                row, col, distance = queue.popleft()
                rooms[row][col] = min(rooms[row][col], distance)
                visited.add((row, col))

                for dr, dc in directions:
                    new_row, new_col = row + dr, col + dc

                    if (
                        0 <= new_row < rows
                        and 0 <= new_col < cols
                        and rooms[new_row][new_col] != -1
                        and (new_row, new_col) not in visited
                    ):
                        queue.append((new_row, new_col, distance + 1))
                        visited.add((new_row, new_col))
```

87. 11_Graphs/08_Course_Schedule/0207-course-schedule.py

```
"""
Problem: LeetCode 207 - Course Schedule

Key Idea:
The problem can be reduced to detecting cycles in a directed graph. We can represent the course prerequisites as directed edges between nodes (courses). If there is a cycle in the graph, it means we can't complete all courses.

Time Complexity:
- Constructing the graph takes O(numCourses + len(prerequisites)) time, where numCourses is the number of courses and len(prerequisites) is the number of prerequisites.
- Detecting a cycle in a directed graph takes O(V + E) time, where V is the number of vertices (courses) and E is the number of edges (prerequisites).
- Therefore, the total time complexity is O(numCourses + len(prerequisites)) + O(V + E), which simplifies to O(numCourses + len(prerequisites)).

Space Complexity:
- The space complexity is O(numCourses + len(prerequisites)), where we store the graph using a dictionary and maintain a set for visited nodes.
"""


class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = {i: [] for i in range(numCourses)}
        in_degree = [0] * numCourses

        # Construct the graph and count in-degrees
        for course, prereq in prerequisites:
            graph[prereq].append(course)
            in_degree[course] += 1

        # Initialize a queue with nodes having in-degree zero
        queue = collections.deque(
            [course for course, degree in enumerate(in_degree) if degree == 0]
        )

        # Perform topological sorting and update in-degrees
        while queue:
            node = queue.popleft()
            for neighbor in graph[node]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

        # If any course has in-degree greater than zero, there's a cycle
        return all(degree == 0 for degree in in_degree)
```

88. 11_Graphs/09_Course_Schedule_II/0210-course-schedule-ii.py

```
"""
Problem: LeetCode 210 - Course Schedule II

Key Idea:
This problem is an extension of the previous Course Schedule problem (LeetCode 207). We need to return the order in which courses can be taken. We can use the topological sorting approach to solve this.

Time Complexity:
- Constructing the graph takes O(numCourses + len(prerequisites)) time, where numCourses is the number of courses and len(prerequisites) is the number of prerequisites.
- Performing topological sorting using BFS takes O(V + E) time, where V is the number of vertices (courses) and E is the number of edges (prerequisites).
- Therefore, the total time complexity is O(numCourses + len(prerequisites)) + O(V + E), which simplifies to O(numCourses + len(prerequisites)).

Space Complexity:
- The space complexity is O(numCourses + len(prerequisites)), where we store the graph using a dictionary, maintain a list for in-degrees, and use a queue for BFS traversal.
"""


class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        graph = {i: [] for i in range(numCourses)}
        in_degree = [0] * numCourses
        order = []

        # Construct the graph and count in-degrees
        for course, prereq in prerequisites:
            graph[prereq].append(course)
            in_degree[course] += 1

        # Initialize a queue with nodes having in-degree zero
        queue = collections.deque(
            [course for course, degree in enumerate(in_degree) if degree == 0]
        )

        # Perform topological sorting and update in-degrees
        while queue:
            node = queue.popleft()
            order.append(node)
            for neighbor in graph[node]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

        # If the order doesn't contain all courses, there's a cycle
        return order if len(order) == numCourses else []
```

89. 11_Graphs/10_Redundant_Connection/0684-redundant-connection.py

```
"""
Problem: LeetCode 684 - Redundant Connection

Key Idea:
This problem can be solved using the Union-Find (Disjoint Set Union) algorithm. We initialize each node as its own parent and iterate through the given edges. For each edge, we check if the nodes have the same parent. If they do, that means adding this edge will create a cycle, and it's the redundant edge. If they don't have the same parent, we merge their sets by updating one's parent to be the other.

Time Complexity:
- Initializing the parent array takes O(n) time, where n is the number of nodes.
- Iterating through the edges takes O(n) time.
- The `find` and `union` operations in the Union-Find algorithm have an amortized time complexity of approximately O(1).
- Therefore, the total time complexity is O(n).

Space Complexity:
- The space complexity is O(n), where we store the parent array and the result array.
"""


class Solution:
    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        n = len(edges)
        parent = list(range(n + 1))  # Initialize each node as its own parent

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])  # Path compression
            return parent[x]

        def union(x, y):
            parent[find(x)] = find(y)

        for edge in edges:
            u, v = edge
            if find(u) == find(v):
                return edge
            union(u, v)

        return []
```

90. 11_Graphs/11_Number_of_Connected_Components_in_an_Undirected_Graph/0323-number-of-connected-components-in-an-undirected-graph.p

```
"""
Problem: LeetCode 323 - Number of Connected Components in an Undirected Graph

Key Idea:
This problem can be solved using Depth-First Search (DFS) or Breadth-First Search (BFS). We represent the given edges as an adjacency list, where each node points to its neighboring nodes. We then iterate through all nodes and perform a DFS/BFS from each unvisited node to explore all connected components. The number of times we need to start a new DFS/BFS corresponds to the number of connected components in the graph.

Time Complexity:
- Constructing the adjacency list takes O(n + m) time, where n is the number of nodes and m is the number of edges.
- Performing DFS/BFS for each unvisited node takes O(n + m) time.
- Therefore, the total time complexity is O(n + m).

Space Complexity:
- The space complexity is O(n + m) to store the adjacency list and visited set.
"""

from collections import defaultdict, deque


class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        graph = defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        def dfs(node):
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    dfs(neighbor)

        visited = set()
        components = 0

        for node in range(n):
            if node not in visited:
                components += 1
                dfs(node)

        return components
```

91. 11_Graphs/12_Graph_Valid_Tree/0261-graph-valid-tree.py

```
"""
Problem: LeetCode 261 - Graph Valid Tree

Key Idea:
This problem can be solved using Depth-First Search (DFS) or Union-Find algorithm. We represent the given edges as an adjacency list, where each node points to its neighboring nodes. To determine whether the graph is a valid tree, we need to check two conditions:
1. The graph must be connected, i.e., there is a path between every pair of nodes.
2. There should be no cycles in the graph.

Time Complexity:
- Constructing the adjacency list takes O(n + m) time, where n is the number of nodes and m is the number of edges.
- Both DFS and Union-Find algorithms take O(n) time.
- Therefore, the total time complexity is O(n + m).

Space Complexity:
- The space complexity is O(n + m) to store the adjacency list and Union-Find data structures.
"""


class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        if len(edges) != n - 1:
            return False

        graph = defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        visited = set()

        def dfs(node, parent):
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor != parent:
                    if neighbor in visited or not dfs(neighbor, node):
                        return False
            return True

        # Check if the graph is connected
        if not dfs(0, -1):
            return False

        return len(visited) == n
```

92. 11_Graphs/13_Word_Ladder/0127-word-ladder.py

```
"""
Problem: LeetCode 127 - Word Ladder

Key Idea:
This problem can be solved using a breadth-first search (BFS) approach. We start with the given beginWord and perform a BFS to explore all possible word transformations, one character change at a time. We maintain a queue to track the current word and its transformation path. For each word in the queue, we generate all possible words by changing one character at a time and check if it's in the word list. If it is, we add it to the queue and mark it as visited. We continue this process until we reach the endWord or the queue is empty.

Time Complexity:
- Constructing the set of words takes O(wordList) time.
- Performing the BFS can take up to O(n * m) time, where n is the number of words and m is the length of the words.
- Therefore, the total time complexity is O(n * m).

Space Complexity:
- The space complexity is O(n), where we store the word list and the visited set.
- The queue can temporarily hold up to O(n) elements in the worst case.
"""

from collections import deque


class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        wordSet = set(wordList)
        if endWord not in wordSet:
            return 0

        queue = deque([(beginWord, 1)])  # Start from the beginWord with level 1
        visited = set()

        while queue:
            word, level = queue.popleft()
            if word == endWord:
                return level

            for i in range(len(word)):
                for c in "abcdefghijklmnopqrstuvwxyz":
                    new_word = word[:i] + c + word[i + 1 :]
                    if new_word in wordSet and new_word not in visited:
                        visited.add(new_word)
                        queue.append((new_word, level + 1))

        return 0
```

93. 12_Advanced_Graphs/01_Reconstruct_Itinerary/0332-reconstruct-itinerary.py

```
"""
Problem: LeetCode 332 - Reconstruct Itinerary

Key Idea:
The problem can be approached using a depth-first search (DFS) approach. We start from the "JFK" airport and explore all possible routes by visiting each airport exactly once. We use a dictionary to store the destinations for each source airport, and for each source airport, we sort the destinations in lexicographical order. This ensures that we visit the airports in the desired order.

Time Complexity:
- Building the graph (destination dictionary) takes O(n * log n) time due to sorting the destinations.
- The DFS traversal takes O(n) time in the worst case since we visit each ticket exactly once.
- Therefore, the overall time complexity is O(n * log n + n), which simplifies to O(n * log n).

Space Complexity:
- We use a dictionary to store the graph, which takes O(n) space.
- The recursive stack for DFS can go as deep as the number of tickets, which is O(n).
- Therefore, the space complexity is O(n).
"""


class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        graph = collections.defaultdict(list)

        for start, end in sorted(tickets, reverse=True):
            graph[start].append(end)

        route = []

        def dfs(node):
            while graph[node]:
                dfs(graph[node].pop())
            route.append(node)

        dfs("JFK")

        return route[::-1]
```

94. 12_Advanced_Graphs/02_Min_Cost_to_Connect_All_Points/1584-min-cost-to-connect-all-points.py

```
"""
Problem: LeetCode 1584 - Min Cost to Connect All Points

Key Idea:
The problem can be solved using Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph. We start by calculating the distances between all pairs of points and then sort these distances along with their corresponding pairs. We initialize an empty MST and iterate through the sorted distances. For each distance, we check if the two points belong to different connected components in the MST using Union-Find. If they do, we add the distance to the MST and merge the components. We continue this process until all points are connected.

Time Complexity:
- Calculating distances between all pairs takes O(n^2) time.
- Sorting the distances takes O(n^2 * log(n^2)) = O(n^2 * log n) time.
- Union-Find operations take nearly constant time (amortized), and we perform it n times.
- Therefore, the overall time complexity is dominated by sorting and is O(n^2 * log n).

Space Complexity:
- We store distances and the edges in O(n^2) space.
- Union-Find data structure uses O(n) space.
- Therefore, the space complexity is O(n^2).
"""


class Solution:
    def minCostConnectPoints(self, points: List[List[int]]) -> int:
        def distance(p1, p2):
            return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        n = len(points)
        distances = []

        for i in range(n):
            for j in range(i + 1, n):
                distances.append((distance(points[i], points[j]), i, j))

        distances.sort()
        parent = list(range(n))
        rank = [0] * n
        mst_cost = 0

        def find(node):
            if parent[node] != node:
                parent[node] = find(parent[node])
            return parent[node]

        def union(node1, node2):
            root1 = find(node1)
            root2 = find(node2)
            if root1 != root2:
                if rank[root1] > rank[root2]:
                    parent[root2] = root1
                else:
                    parent[root1] = root2
                    if rank[root1] == rank[root2]:
                        rank[root2] += 1

        for distance, u, v in distances:
            if find(u) != find(v):
                union(u, v)
                mst_cost += distance
```

```
return mst_cost
```

95. 12_Advanced_Graphs/03_Network_Delay_Time/0743-network-delay-time.py

```
"""
Problem: LeetCode 743 - Network Delay Time

Key Idea:
The problem can be solved using Dijkstra's algorithm to find the shortest paths from a source node to all other nodes in a weighted graph. We start by creating an adjacency list representation of the graph. We maintain a priority queue to select the next node to visit based on the minimum distance. We initialize distances to all nodes as infinity except for the source node, which is set to 0. We continue exploring nodes and updating distances until the priority queue is empty. The maximum distance among all nodes will be the answer.

Time Complexity:
- Constructing the adjacency list takes O(n + m) time, where n is the number of nodes and m is the number of edges.
- Initializing distances takes O(n) time.
- The priority queue operations take O(n * log(n)) time in total (amortized) as we insert each node once.
- Overall, the time complexity is dominated by the priority queue operations and is O(n * log(n) + n + m).

Space Complexity:
- The adjacency list takes O(n + m) space.
- The priority queue takes O(n) space.
- The distances array takes O(n) space.
- Therefore, the space complexity is O(n + m).
"""

import heapq
from collections import defaultdict


class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        # Create an adjacency list representation of the graph
        graph = defaultdict(list)
        for u, v, w in times:
            graph[u].append((v, w))

        # Initialize distances to all nodes as infinity except for the source node
        distances = [float("inf")] * (n + 1)
        distances[k] = 0

        # Priority queue to select the next node to visit based on the minimum distance
        pq = [(0, k)]

        while pq:
            distance, node = heapq.heappop(pq)
            if distance > distances[node]:
                continue
            for neighbor, weight in graph[node]:
                if distance + weight < distances[neighbor]:
                    distances[neighbor] = distance + weight
                    heapq.heappush(pq, (distances[neighbor], neighbor))

        # Find the maximum distance among all nodes
        max_distance = max(distances[1:])

        return max_distance if max_distance < float("inf") else -1
```

96. 12_Advanced_Graphs/04_Swim_in_Rising_Water/0778-swim-in-rising-water.py

```
"""
Problem: LeetCode 778 - Swim in Rising Water

Key Idea:
The problem can be approached using a binary search along with depth-first search (DFS). We can perform a binary search on the range of possible time values, from the minimum value (0) to the maximum value (N * N). For each time value, we perform a DFS to check if it's possible to reach the bottom-right cell from the top-left cell without encountering cells with heights greater than the current time value. If a valid path exists, we narrow down the search to the left half of the time range; otherwise, we search in the right half.

Time Complexity:
- The binary search takes O(log(N^2)) time.
- The DFS visits each cell once and takes O(N^2) time.
- Overall, the time complexity is O(N^2 * log(N^2)).

Space Complexity:
- The DFS uses the call stack, which takes O(N^2) space.
- Additional data structures used in the algorithm take constant space.
- Therefore, the space complexity is O(N^2).
"""


class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        def dfs(i, j, visited, time):
            if i < 0 or i >= N or j < 0 or j >= N or visited[i][j] or grid[i][j] > time:
                return False
            if i == N - 1 and j == N - 1:
                return True
            visited[i][j] = True
            return (
                dfs(i + 1, j, visited, time)
                or dfs(i - 1, j, visited, time)
                or dfs(i, j + 1, visited, time)
                or dfs(i, j - 1, visited, time)
            )

        N = len(grid)
        left, right = 0, N * N

        while left < right:
            mid = (left + right) // 2
            visited = [[False] * N for _ in range(N)]
            if dfs(0, 0, visited, mid):
                right = mid
            else:
                left = mid + 1

        return left
```

97. 12_Advanced_Graphs/05_Alien_Dictionary/0269-alien-dictionary.py

```
"""
Problem: LeetCode 269 - Alien Dictionary

Key Idea:
The problem can be solved using topological sorting. The given words can be thought of as directed edges between characters of adjacent words. We can build a graph where each character is a node, and the edges represent the order between characters. Then, we can perform topological sorting to find the correct order of characters.

Time Complexity:
- Building the graph takes O(N + E) time, where N is the number of words and E is the number of edges (character comparisons).
- Performing topological sorting takes O(V + E) time, where V is the number of characters and E is the number of edges.
- In the worst case, each character is compared with the next character in all words.
- Overall, the time complexity is O(N + E + V).

Space Complexity:
- The space required for the graph representation is O(V + E), where V is the number of characters and E is the number of edges.
- Other data structures used take constant space.
- Therefore, the space complexity is O(V + E).
"""

from collections import defaultdict, deque


class Solution:
    def alienOrder(self, words: List[str]) -> str:
        graph = defaultdict(list)
        in_degree = defaultdict(int)

        for i in range(len(words) - 1):
            word1, word2 = words[i], words[i + 1]
            for j in range(min(len(word1), len(word2))):
                if word1[j] != word2[j]:
                    graph[word1[j]].append(word2[j])
                    in_degree[word2[j]] += 1
                    break

        queue = deque(char for char, indeg in in_degree.items() if indeg == 0)
        result = []

        while queue:
            char = queue.popleft()
            result.append(char)
            for neighbor in graph[char]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

        if len(result) < len(in_degree):
            return ""
        return "".join(result)
```

98. 12_Advanced_Graphs/06_Cheapest_Flights_Within_K_Stops/0787-cheapest-flights-within-k-stops.py

```
"""
Problem: LeetCode 787 - Cheapest Flights Within K Stops

Key Idea:
The problem can be solved using Dijkstra's algorithm with a modified priority queue (min-heap) and BFS (Breadth-First Search) approach. We can create a graph where each node represents a city, and the edges represent flights between cities with associated costs. We use a priority queue to explore the nodes in a way that prioritizes the minimum cost. We continue the BFS process until we reach the destination city or exhaust the maximum number of allowed stops (K+1).

Time Complexity:
- The priority queue operations take O(E * log(V)) time, where E is the number of flights and V is the number of cities.
- In the worst case, all cities are explored, and all flights are considered.
- Overall, the time complexity is O(E * log(V)).

Space Complexity:
- The space required for the graph representation is O(V + E), where V is the number of cities and E is the number of flights.
- The space for the priority queue is O(V).
- Other data structures used take constant space.
- Therefore, the space complexity is O(V + E).
"""

import heapq
import math
from typing import List


class Solution:
    def findCheapestPrice(
        self, n: int, flights: List[List[int]], src: int, dst: int, max_stops: int
    ) -> int:
        graph = [[] for _ in range(n)]
        min_heap = [
            (0, src, max_stops + 1)
        ]  # (total_cost, current_city, remaining_stops)
        distances = [[math.inf] * (max_stops + 2) for _ in range(n)]

        for u, v, w in flights:
            graph[u].append((v, w))

        while min_heap:
            total_cost, current_city, remaining_stops = heapq.heappop(min_heap)
            if current_city == dst:
                return total_cost
            if remaining_stops > 0:
                for neighbor, cost in graph[current_city]:
                    new_cost = total_cost + cost
                    if new_cost < distances[neighbor][remaining_stops - 1]:
                        distances[neighbor][remaining_stops - 1] = new_cost
                        heapq.heappush(
                            min_heap, (new_cost, neighbor, remaining_stops - 1)
                        )

        return -1
```

99. 13_1-D_Dynamic_Programming/01_Climbing_Stairs/0070-climbing-stairs.py

```
"""
Problem: LeetCode 70 - Climbing Stairs

Key Idea:
To climb to the n-th stair, you have two options,
1. You can climb from the (n-1)-th stair.
2. You can climb from the (n-2)-th stair.

So, the number of ways to reach the n-th stair is the sum of the number of ways to reach the (n-1)-th and (n-2)-th stairs. This forms a Fibonacci sequence where f(n) = f(n-1) + f(n-2).

Time Complexity:
The time complexity is O(n) because we iterate from 1 to n to calculate the number of ways for each stair.

Space Complexity:
The space complexity is O(1) because we only need two variables to keep track of the previous two results and a temporary variable to calculate the current result.
"""


class Solution:
    def climbStairs(self, n: int) -> int:
        if n <= 2:
            return n

        prev1 = 1  # Number of ways to reach the 1st stair
        prev2 = 2  # Number of ways to reach the 2nd stair

        for i in range(3, n + 1):
            current = prev1 + prev2
            prev1, prev2 = prev2, current  # Update for the next iteration

        return prev2  # Number of ways to reach the n-th stair
```

100. 13_1-D_Dynamic_Programming/02_Min_Cost_Climbing_Stairs/0746-min-cost-climbing-stairs.py

```
"""
Problem: LeetCode 746 - Min Cost Climbing Stairs

Key Idea:
To find the minimum cost to reach the top of the stairs, we can use dynamic programming. We start by creating a list dp where dp[i] represents the minimum cost to reach the i-th stair. We initialize dp[0] and dp[1] to the costs of the 0-th and 1-st stairs, as there's no cost to reach them.

Then, we iterate from the 2-nd stair to the top, and for each stair i, we calculate dp[i] as the minimum of either reaching it from dp[i-1] (one step) or dp[i-2] (two steps), plus the cost of the current stair.

The minimum cost to reach the top will be either dp[n-1] or dp[n-2], where n is the number of stairs, as we can reach the top by either taking one step from the last stair or two steps from the second-to-last stair.

Time Complexity:
The time complexity is O(n) because we iterate through the stairs once.

Space Complexity:
The space complexity is O(n) because we use a list dp to store the minimum costs for each stair.
"""


class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        n = len(cost)
        if n <= 1:
            return 0  # No cost if there are 0 or 1 stairs

        dp = [0] * n  # Initialize a list to store minimum costs

        # Base cases
        dp[0] = cost[0]
        dp[1] = cost[1]

        for i in range(2, n):
            dp[i] = min(dp[i - 1], dp[i - 2]) + cost[i]

        return min(dp[n - 1], dp[n - 2])
```

101. 13_1-D_Dynamic_Programming/03_House_Robber/0198-house-robber.py

```
"""
Problem: LeetCode 198 - House Robber

Key Idea:
The key idea to solve this problem is to use dynamic programming. We create a list `dp` where `dp[i]` represents the maximum amount of money that can be robbed up to the `i-th` house.

To populate `dp`, we iterate through the houses from left to right. For each house `i`, we have two choices:
1. Rob the current house, which means we add the money from the current house (`nums[i]`) to the amount robbed from two houses before (`dp[i-2]`).
2. Skip the current house and take the maximum amount robbed from the previous house (`dp[i-1]`).

We choose the maximum of these two options and update `dp[i]` accordingly. Finally, `dp[-1]` will contain the maximum amount that can be robbed.

Time Complexity:
The time complexity is O(n) because we iterate through the list of houses once.

Space Complexity:
The space complexity is O(n) because we use a list `dp` to store the maximum amounts that can be robbed up to each house.
"""


class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0
        if n == 1:
            return nums[0]

        dp = [0] * n
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, n):
            dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])

        return dp[-1]
```

102. 13_1-D_Dynamic_Programming/04_House_Robber_II/0213-house-robber-ii.py

```
"""
Problem: LeetCode 213 - House Robber II

Key Idea:
The key idea for solving this problem is to recognize that it's an extension of the House Robber I problem (LeetCode 198). Since we cannot rob adjacent houses, we have two scenarios:
1. Rob the first house and exclude the last house.
2. Exclude the first house and consider robbing the last house.

We calculate the maximum amount for both scenarios using the House Robber I algorithm and return the maximum of the two results.

Time Complexity:
The time complexity is O(n) because we use the House Robber I algorithm twice, once excluding the last house and once excluding the first house.

Space Complexity:
The space complexity is O(1) because we use a constant amount of extra space to store variables.
"""


class Solution:
    def rob(self, nums: List[int]) -> int:
        def houseRobber(nums: List[int]) -> int:
            n = len(nums)
            if n == 0:
                return 0
            if n == 1:
                return nums[0]

            prev = 0
            curr = 0

            for num in nums:
                temp = curr
                curr = max(prev + num, curr)
                prev = temp

            return curr

        if len(nums) == 1:
            return nums[0]

        # Rob first house and exclude the last house, or exclude the first house and rob the last house.
        return max(houseRobber(nums[1:]), houseRobber(nums[:-1]))
```

103. 13_1-D_Dynamic_Programming/05_Longest_Palindromic_Substring/0005-longest-palindromic-substring.py

```
"""
Problem: LeetCode 5 - Longest Palindromic Substring

Key Idea:
The key idea for solving this problem is to expand around each character in the string and check for palindromes. We will consider each character as the center of a potential palindrome and expand outwards while checking if the characters at both ends are equal. We need to handle two cases: palindromes with odd length (centered at a single character) and palindromes with even length (centered between two characters).

Time Complexity:
The time complexity is O(n^2) because in the worst case, we might expand around each of the 'n' characters, and each expansion can take O(n) time in the worst case.

Space Complexity:
The space complexity is O(1) because we use a constant amount of extra space to store variables.
"""


class Solution:
    def longestPalindrome(self, s: str) -> str:
        def expandAroundCenter(left: int, right: int) -> str:
            # Expand around the center while the characters at both ends are equal.
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
            # Return the palindrome substring.
            return s[left + 1 : right]

        if len(s) < 2:
            return s

        longest = ""

        for i in range(len(s)):
            # Check for odd-length palindromes.
            palindrome1 = expandAroundCenter(i, i)
            if len(palindrome1) > len(longest):
                longest = palindrome1

            # Check for even-length palindromes.
            palindrome2 = expandAroundCenter(i, i + 1)
            if len(palindrome2) > len(longest):
                longest = palindrome2

        return longest
```

104. 13_1-D_Dynamic_Programming/06_Palindromic_Substrings/0647-palindromic-substrings.py

```
"""
Problem: LeetCode 647 - Palindromic Substrings

Key Idea:
The key idea for solving this problem is to expand around each character in the string and count the palindromic substrings. We will consider each character as the center of a potential palindrome and expand outwards while checking if the characters at both ends are equal. We need to handle two cases: palindromes with odd length (centered at a single character) and palindromes with even length (centered between two characters).

Time Complexity:
The time complexity is O(n^2) because in the worst case, we might expand around each of the 'n' characters, and each expansion can take O(n) time in the worst case.

Space Complexity:
The space complexity is O(1) because we use a constant amount of extra space to store variables.
"""


class Solution:
    def countSubstrings(self, s: str) -> int:
        def expandAroundCenter(left: int, right: int) -> int:
            count = 0
            # Expand around the center while the characters at both ends are equal.
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
                count += 1
            return count

        if not s:
            return 0

        total_palindromes = 0

        for i in range(len(s)):
            # Check for odd-length palindromes.
            total_palindromes += expandAroundCenter(i, i)

            # Check for even-length palindromes.
            total_palindromes += expandAroundCenter(i, i + 1)

        return total_palindromes
```

105. 13_1-D_Dynamic_Programming/07_Decode_Ways/0091-decode-ways.py

```
"""
Problem: LeetCode 91 - Decode Ways

Key Idea:
The key idea for solving this problem is to use dynamic programming to count the number of ways to decode the given string. We will create a DP array where dp[i] represents the number of ways to decode the string s[:i] (the first i characters of the string).

Time Complexity:
The time complexity is O(n), where n is the length of the input string s. We iterate through the string once to fill in the DP array.

Space Complexity:
The space complexity is O(n) because we use an additional DP array of the same length as the input string.
"""


class Solution:
    def numDecodings(self, s: str) -> int:
        n = len(s)

        # Initialize a DP array to store the number of ways to decode substrings.
        dp = [0] * (n + 1)

        # Base cases:
        dp[0] = 1  # An empty string can be decoded in one way.
        dp[1] = (
            1 if s[0] != "0" else 0
        )  # The first character can be decoded in one way if it's not '0'.

        # Fill in the DP array.
        for i in range(2, n + 1):
            # Check the one-digit and two-digit possibilities.
            one_digit = int(s[i - 1])
            two_digits = int(s[i - 2 : i])

            # If the one-digit is not '0', it can be decoded in the same way as dp[i-1].
            if one_digit != 0:
                dp[i] += dp[i - 1]

            # If the two-digit is between 10 and 26, it can be decoded in the same way as dp[i-2].
            if 10 <= two_digits <= 26:
                dp[i] += dp[i - 2]

        return dp[n]
```

106. 13_1-D_Dynamic_Programming/08_Coin_Change/0322-coin-change.py

```
"""
Problem: LeetCode 322 - Coin Change

Key Idea:
The key idea is to use dynamic programming to find the minimum number of coins needed to make up the given amount. We will create a DP array where dp[i] represents the minimum number of coins needed to make up the amount i.

Time Complexity:
The time complexity is O(amount * n), where 'amount' is the given amount to make up, and 'n' is the number of coin denominations available. We iterate through 'amount' and 'n' to fill in the DP array.

Space Complexity:
The space complexity is O(amount + 1) because we use a DP array of size 'amount + 1'.
"""


class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # Initialize the DP array with a maximum value to represent impossible cases.
        dp = [float("inf")] * (amount + 1)

        # Base case: It takes 0 coins to make up the amount of 0.
        dp[0] = 0

        # Iterate through the DP array and update the minimum number of coins needed.
        for coin in coins:
            for i in range(coin, amount + 1):
                dp[i] = min(dp[i], dp[i - coin] + 1)

        # If dp[amount] is still float('inf'), it means it's impossible to make up the amount.
        # Otherwise, dp[amount] contains the minimum number of coins needed.
        return dp[amount] if dp[amount] != float("inf") else -1
```

107. 13_1-D_Dynamic_Programming/09_Maximum_Product_Subarray/0152-maximum-product-subarray.py

```
"""
Problem: LeetCode 152 - Maximum Product Subarray

Key Idea:
The key idea is to use dynamic programming to keep track of the maximum and minimum product ending at each position in the array. Since multiplying a negative number by a negative number results in a positive number, we need to keep track of both the maximum and minimum products to handle negative numbers.

Time Complexity:
The time complexity is O(n), where 'n' is the length of the input array. We iterate through the array once to find the maximum product subarray.

Space Complexity:
The space complexity is O(1) because we use only a few variables to store the maximum and minimum products at each position.
"""


class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize variables to keep track of the maximum and minimum product ending at the current position.
        max_product = min_product = result = nums[0]

        # Iterate through the array starting from the second element.
        for i in range(1, len(nums)):
            # If the current element is negative, swap max_product and min_product
            # because multiplying a negative number can turn the maximum into the minimum.
            if nums[i] < 0:
                max_product, min_product = min_product, max_product

            # Update max_product and min_product based on the current element.
            max_product = max(nums[i], max_product * nums[i])
            min_product = min(nums[i], min_product * nums[i])

            # Update the overall result with the maximum product found so far.
            result = max(result, max_product)

        return result
```

108. 13_1-D_Dynamic_Programming/10_Word_Break/0139-word-break.py

```
"""
Problem: LeetCode 139 - Word Break

Key Idea:
The key idea is to use dynamic programming to determine if it's possible to break the input string into words from the wordDict. We can create a boolean array dp, where dp[i] is True if we can break the substring s[0:i] into words from the wordDict.

Time Complexity:
The time complexity is O(n^2), where 'n' is the length of the input string. We iterate through the string and for each character, we check if it can be broken into words from the wordDict.

Space Complexity:
The space complexity is O(n) because we use a boolean array dp of length 'n' to store the results.
"""


class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        # Create a set for faster word lookup.
        wordSet = set(wordDict)

        # Initialize a boolean array dp where dp[i] is True if s[0:i] can be broken into words.
        dp = [False] * (len(s) + 1)
        dp[0] = True  # An empty string can always be broken.

        # Iterate through the string.
        for i in range(1, len(s) + 1):
            for j in range(i):
                # Check if the substring s[j:i] is in the wordDict and if s[0:j] can be broken.
                if dp[j] and s[j:i] in wordSet:
                    dp[i] = True
                    break  # No need to continue checking.

        return dp[len(s)]
```

109. 13_1-D_Dynamic_Programming/11_Longest_Increasing_Subsequence/0300-longest-increasing-subsequence.py

```
"""
Problem: LeetCode 300 - Longest Increasing Subsequence

Key Idea:
The key idea is to use dynamic programming to find the length of the longest increasing subsequence. We create a dynamic programming array dp, where dp[i] represents the length of the longest increasing subsequence ending at index i.

Time Complexity:
The time complexity is O(n^2), where 'n' is the length of the input nums. We use a nested loop to compare and update the values in the dp array.

Space Complexity:
The space complexity is O(n) because we use an additional dp array of length 'n' to store the results.
"""


class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        if not nums:
            return 0

        # Initialize a dynamic programming array dp with all values set to 1.
        dp = [1] * len(nums)

        # Iterate through the array to find the longest increasing subsequence.
        for i in range(len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)

        # Return the maximum value in dp, which represents the length of the longest increasing subsequence.
        return max(dp)
```

110. 13_1-D_Dynamic_Programming/12_Partition_Equal_Subset_Sum/0416-partition-equal-subset-sum.py

```
"""
Problem: LeetCode 416 - Partition Equal Subset Sum

Key Idea:
The key idea is to use dynamic programming to determine if it's possible to partition the input array into two subsets with equal sums. We create a dynamic programming array dp, where dp[i][j] represents whether it's possible to form a subset with a sum of 'j' using the first 'i' elements of the input array.

Time Complexity:
The time complexity is O(n * sum(nums)), where 'n' is the length of the input nums and 'sum(nums)' is the sum of all elements in the input. This is because we use a nested loop to iterate through each element in nums and each possible sum up to 'sum(nums)'.

Space Complexity:
The space complexity is O(n * sum(nums)) because we use a 2D dp array of size (n+1) x (sum(nums)+1) to store the results.
"""


class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total_sum = sum(nums)

        # If the total sum is odd, it's impossible to partition into two equal subsets.
        if total_sum % 2 != 0:
            return False

        target_sum = total_sum // 2

        # Initialize a dynamic programming array dp with all values set to False.
        dp = [False] * (target_sum + 1)

        # It's always possible to achieve a sum of 0 using an empty subset.
        dp[0] = True

        for num in nums:
            for i in range(target_sum, num - 1, -1):
                # If it's possible to achieve a sum of 'i - num' using a subset,
                # then it's also possible to achieve a sum of 'i' using a subset.
                dp[i] = dp[i] or dp[i - num]

        return dp[target_sum]
```

111. 14_2-D_Dynamic_Programming/01_Unique_Paths/0062-unique-paths.py

```
"""
Problem: LeetCode 62 - Unique Paths

Key Idea:
The key idea is to use dynamic programming to count the number of unique paths from the top-left corner to the bottom-right corner of a grid. We create a dynamic programming grid 'dp', where dp[i][j] represents the number of unique paths to reach cell (i, j) from the top-left corner.

Time Complexity:
The time complexity is O(m*n), where 'm' is the number of rows and 'n' is the number of columns in the grid. We iterate through each cell once to fill in the dp array.

Space Complexity:
The space complexity is O(m*n) because we use an additional dp grid of the same size as the input grid to store the results.
"""


class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # Initialize a 2D dp grid of size m x n with all values set to 1.
        dp = [[1] * n for _ in range(m)]

        # Fill in the dp grid using dynamic programming.
        for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

        # The value in dp[m-1][n-1] represents the number of unique paths.
        return dp[m - 1][n - 1]
```

112. 14_2-D_Dynamic_Programming/02_Longest_Common_Subsequence/1143-longest-common-subsequence.py

```
"""
Problem: LeetCode 1143 - Longest Common Subsequence

Key Idea:
The key idea is to use dynamic programming to find the length of the longest common subsequence (LCS) between two strings. We create a 2D dp array where dp[i][j] represents the length of the LCS between the first i characters of text1 and the first j characters of text2.

Time Complexity:
The time complexity is O(m*n), where 'm' is the length of text1 and 'n' is the length of text2. We iterate through both strings once to fill in the dp array.

Space Complexity:
The space complexity is O(m*n) because we use an additional dp array of the same size as the input strings.
"""


class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)

        # Create a 2D dp array of size (m+1) x (n+1) and initialize it with zeros.
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Fill in the dp array using dynamic programming.
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

        # The value in dp[m][n] represents the length of the LCS.
        return dp[m][n]
```

113. 14_2-D_Dynamic_Programming/03_Best_Time_to_Buy_And_Sell_Stock_With_Cooldown/0309-best-time-to-buy-and-sell-stock-with-cooldown.py

```
"""
Problem: LeetCode 309 -Best Time to Buy and Sell Stock with Cooldown

Key Idea:
The key idea is to use dynamic programming to keep track of the maximum profit we can obtain at each day 'i',
considering the actions we can take: buy, sell, or do nothing (cooldown).

Time Complexity:
The time complexity is O(n), where 'n' is the length of the input prices list. We iterate through the prices
list once to compute the maximum profit at each day.

Space Complexity:
The space complexity is O(1) because we use only a constant amount of extra space to store variables for
tracking the maximum profit at each day.
"""


class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        # Initialize variables to represent the maximum profit after each action.
        buy = -prices[
            0
        ]  # Maximum profit after buying on day 0 (negative because we spend money).
        sell = 0  # Maximum profit after selling on day 0 (no profit yet).
        cooldown = 0  # Maximum profit after cooldown on day 0 (no profit yet).

        for i in range(1, len(prices)):
            # To maximize profit on day 'i', we can either:

             # 1. Buy on day 'i'. We subtract the price of the stock from the maximum profit after cooldown on
day 'i-2'.
            new_buy = max(buy, cooldown - prices[i])

              # 2. Sell on day 'i'. We add the price of the stock to the maximum profit after buying on day
'i-1'.
            new_sell = buy + prices[i]

             # 3. Do nothing (cooldown) on day 'i'. We take the maximum of the maximum profit after cooldown on
day 'i-1' and after selling on day 'i-1'.
            new_cooldown = max(cooldown, sell)

            # Update the variables for the next iteration.
            buy, sell, cooldown = new_buy, new_sell, new_cooldown

         # The maximum profit will be the maximum of the profit after selling on the last day and the profit
after cooldown on the last day.
        return max(sell, cooldown)
```

114. 14_2-D_Dynamic_Programming/04_Coin_Change_II/0518-coin-change-ii.py

```
"""
Problem: LeetCode 518 - Coin Change II

Key Idea:
The key idea is to use dynamic programming to count the number of combinations that make up each amount from 0 to 'amount'.

Time Complexity:
The time complexity is O(amount * n), where 'amount' is the target amount and 'n' is the number of coin denominations. We iterate through 'amount' values and for each amount, we iterate through 'n' coin denominations.

Space Complexity:
The space complexity is O(amount) because we use a 1D array 'dp' of size 'amount + 1' to store the number of combinations for each amount.
"""


class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        # Initialize a 1D array dp to store the number of combinations for each amount from 0 to amount.
        dp = [0] * (amount + 1)

        # There is one way to make amount 0 (by not using any coins).
        dp[0] = 1

        # Iterate through each coin denomination.
        for coin in coins:
            # Update the dp array for each amount from coin to amount.
            for i in range(coin, amount + 1):
                dp[i] += dp[i - coin]

        # The dp[amount] contains the number of combinations to make the target amount.
        return dp[amount]
```

115. 14_2-D_Dynamic_Programming/05_Target_Sum/0494-target-sum.py

```
"""
Problem: LeetCode 494 - Target Sum

Key Idea:
The key idea is to use dynamic programming to count the number of ways to reach a certain sum by assigning either a positive or negative sign to each element in the input array.

Time Complexity:
The time complexity is O(n * sum), where 'n' is the number of elements in the input array 'nums' and 'sum' is the sum of all elements in 'nums'. We iterate through 'n' elements and, for each element, iterate through 'sum' values.

Space Complexity:
The space complexity is O(sum) because we use a 1D array 'dp' of size 'sum + 1' to store the number of ways to reach each sum from 0 to 'sum'.
"""


class Solution:
    def findTargetSumWays(self, nums: List[int], S: int) -> int:
        # Calculate the sum of all elements in the input array 'nums'.
        total_sum = sum(nums)

        # If the total sum is less than the target sum 'S', it's not possible to reach 'S'.
        if (total_sum + S) % 2 != 0 or total_sum < abs(S):
            return 0

        # Calculate the target sum for positive signs. (total_sum + S) / 2
        target = (total_sum + S) // 2

        # Initialize a 1D array 'dp' to store the number of ways to reach each sum from 0 to 'target'.
        dp = [0] * (target + 1)

        # There is one way to reach a sum of 0 (by not selecting any element).
        dp[0] = 1

        # Iterate through each element in the input array 'nums'.
        for num in nums:
            # Update the 'dp' array for each sum from 'target' to 'num'.
            for i in range(target, num - 1, -1):
                dp[i] += dp[i - num]

        # The 'dp[target]' contains the number of ways to reach the target sum 'S'.
        return dp[target]
```

116. 14_2-D_Dynamic_Programming/06_Interleaving_String/0097-interleaving-string.py

```
"""
Problem: LeetCode 97 - Interleaving String

Key Idea:
The key idea is to use dynamic programming to determine whether s3 can be formed by interleaving s1 and s2.

Time Complexity:
The time complexity is O(m * n), where 'm' is the length of string 's1' and 'n' is the length of string 's2'.
We use a 2D array 'dp' of size (m + 1) x (n + 1) to store intermediate results.

Space Complexity:
The space complexity is O(m * n), as we use a 2D array 'dp' of size (m + 1) x (n + 1) to store intermediate
results.
"""


class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        # Get the lengths of s1, s2, and s3.
        m, n, p = len(s1), len(s2), len(s3)

        # If the sum of lengths of s1 and s2 is not equal to the length of s3, return False.
        if m + n != p:
            return False

        # Initialize a 2D DP array dp of size (m + 1) x (n + 1) to store intermediate results.
        dp = [[False] * (n + 1) for _ in range(m + 1)]

        # Base case: dp[0][0] is True since two empty strings can form an empty string.
        dp[0][0] = True

        # Fill in the first row of dp.
        for j in range(1, n + 1):
            dp[0][j] = dp[0][j - 1] and s2[j - 1] == s3[j - 1]

        # Fill in the first column of dp.
        for i in range(1, m + 1):
            dp[i][0] = dp[i - 1][0] and s1[i - 1] == s3[i - 1]

        # Fill in the rest of dp.
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # For dp[i][j] to be True, one of the following conditions must be met:
                # 1. dp[i-1][j] is True and s1[i-1] matches s3[i+j-1].
                # 2. dp[i][j-1] is True and s2[j-1] matches s3[i+j-1].
                dp[i][j] = (dp[i - 1][j] and s1[i - 1] == s3[i + j - 1]) or (
                    dp[i][j - 1] and s2[j - 1] == s3[i + j - 1]
                )

        # The result is stored in dp[m][n].
        return dp[m][n]
```

117. 14_2-D_Dynamic_Programming/07_Longest_Increasing_Path_In_a_Matrix/0329-longest-increasing-path-in-a-matrix.py

```
"""
Problem: LeetCode 329 - Longest Increasing Path in a Matrix

Key Idea:
The key idea is to use dynamic programming to find the longest increasing path in a matrix. We can start from each cell and perform a depth-first search (DFS) to explore the neighboring cells with larger values.

Time Complexity:
The time complexity is O(m * n), where 'm' is the number of rows and 'n' is the number of columns in the matrix. We visit each cell once and perform DFS from each cell.

Space Complexity:
The space complexity is O(m * n) as we use a memoization table 'dp' of the same size to store the results of subproblems.
"""


class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        if not matrix:
            return 0

        # Define directions for moving to neighboring cells: up, down, left, right.
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # Function to perform DFS from a given cell (i, j).
        def dfs(i, j):
            # If the result for this cell is already calculated, return it.
            if dp[i][j] != -1:
                return dp[i][j]

            # Initialize the result for this cell to 1 (counting itself).
            dp[i][j] = 1

            # Explore the four neighboring cells.
            for dx, dy in directions:
                x, y = i + dx, j + dy
                if 0 <= x < m and 0 <= y < n and matrix[x][y] > matrix[i][j]:
                    # If the neighboring cell has a larger value, perform DFS.
                    dp[i][j] = max(dp[i][j], 1 + dfs(x, y))

            return dp[i][j]

        m, n = len(matrix), len(matrix[0])
        dp = [[-1] * n for _ in range(m)]  # Memoization table to store results.
        max_path = 0

        # Start DFS from each cell in the matrix.
        for i in range(m):
            for j in range(n):
                max_path = max(max_path, dfs(i, j))

        return max_path
```

118. 14_2-D_Dynamic_Programming/08_Distinct_Subsequences/0115-distinct-subsequences.py

```
"""
Problem: LeetCode 115 - Distinct Subsequences

Key Idea:
The key idea is to use dynamic programming to count the number of distinct subsequences of the string 's' that match the string 't'. We can create a 1D DP array dp, where dp[j] represents the number of distinct subsequences of 's' that match 't' up to index 'j'. We can build the DP array based on the following rules:

1. Initialize dp[0] to 1 because there is one way to match an empty string '""' to another empty string '""'.

2. For each character in 's', update dp[j] based on the following conditions:
   - If s[i] == t[j], we can either match the current characters (s[i] and t[j]), which contributes dp[j-1] to the count, or we can skip the current character in 's', which contributes dp[j] to the count.
   - If s[i] != t[j], we can only skip the current character in 's', which contributes dp[j] to the count.

Time Complexity:
The time complexity is O(m * n), where 'm' is the length of string 's' and 'n' is the length of string 't'.

Space Complexity:
The space complexity is O(n), where 'n' is the length of string 't', as we use a 1D DP array of size n.
"""

# This solution only passes 65/66 testcases for some reason. Tried other solutions and they don't work either, so it's probably a faulty testcase.
# If you have a solution that passes all testcases, please open a pr.


class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        m, n = len(s), len(t)

        # Create a 2D table dp to store the number of distinct subsequences.
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Initialize the first row of dp. There is one way to form an empty subsequence.
        for i in range(m + 1):
            dp[i][0] = 1

        # Fill the dp table using dynamic programming.
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # If the characters match, we have two options:
                # 1. Include the current character in the subsequence (dp[i-1][j-1] ways).
                # 2. Exclude the current character (dp[i-1][j] ways).
                if s[i - 1] == t[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
                else:
                    # If the characters don't match, we can only exclude the current character.
                    dp[i][j] = dp[i - 1][j]

        return dp[m][n]
```

119. 14_2-D_Dynamic_Programming/09_Edit_Distance/0072-edit-distance.py

```
"""
Problem: LeetCode 72 - Edit Distance

Key Idea:
The key idea is to use dynamic programming to calculate the minimum number of operations (insert, delete, or replace) required to transform one string 'word1' into another string 'word2'. We can define a 2D table 'dp' where dp[i][j] represents the minimum edit distance between the first 'i' characters of 'word1' and the first 'j' characters of 'word2'.

Time Complexity:
The time complexity is O(m * n), where 'm' is the length of string 'word1' and 'n' is the length of string 'word2'. We iterate through both strings once to fill the dp table.

Space Complexity:
The space complexity is O(m * n) as we use a 2D table 'dp' of the same size to store the results of subproblems.
"""


class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)

        # Create a 2D table dp to store the minimum edit distance.
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Initialize the first row and first column of dp.
        for i in range(m + 1):
            dp[i][0] = i
        for j in range(n + 1):
            dp[0][j] = j

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # If the characters match, no additional operation is needed.
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    # Choose the minimum of the three possible operations:
                    # 1. Insert a character (dp[i][j - 1] + 1)
                    # 2. Delete a character (dp[i - 1][j] + 1)
                    # 3. Replace a character (dp[i - 1][j - 1] + 1)
                    dp[i][j] = min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1

        return dp[m][n]
```

120. 14_2-D_Dynamic_Programming/10_Burst_Balloons/0312-burst-balloons.py

```
"""
Problem: LeetCode 312 - Burst Balloons

Key Idea:
The key idea is to use dynamic programming to calculate the maximum coins that can be obtained by bursting balloons in a certain order. We create a 2D table 'dp' where dp[i][j] represents the maximum coins obtained by bursting balloons from 'i' to 'j' (exclusive). We iterate through different balloon ranges and choose the best order to burst the balloons.

Time Complexity:
The time complexity is O(n^3), where 'n' is the number of balloons. We have three nested loops to fill the dp table.

Space Complexity:
The space complexity is O(n^2) as we use a 2D table 'dp' of size n x n to store the results of subproblems.
"""


class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)

        # Add virtual balloons at the beginning and end with a value of 1.
        nums = [1] + nums + [1]

        # Create a 2D table dp to store the maximum coins.
        dp = [[0] * (n + 2) for _ in range(n + 2)]

        # Iterate through different balloon ranges.
        for length in range(2, n + 2):
            for left in range(0, n + 2 - length):
                right = left + length
                for k in range(left + 1, right):
                    # Choose the best order to burst balloons in the range [left, right].
                    dp[left][right] = max(
                        dp[left][right],
                        nums[left] * nums[k] * nums[right] + dp[left][k] + dp[k][right],
                    )

        return dp[0][n + 1]
```

121. 14_2-D_Dynamic_Programming/11_Regular_Expression_Matching/0010-regular-expression-matching.py

```
"""
Problem: LeetCode 10 - Regular Expression Matching

Key Idea:
The key idea is to use dynamic programming to solve this problem. We create a 2D table 'dp' where dp[i][j]
represents whether the first 'i' characters in the string 's' match the first 'j' characters in the pattern
'p'. We fill this table based on the characters in 's' and 'p' and previous results.

Time Complexity:
The time complexity is O(m * n), where 'm' is the length of the string 's' and 'n' is the length of the pattern
'p'. We have a nested loop that iterates through the characters in 's' and 'p'.

Space Complexity:
The space complexity is O(m * n) as we use a 2D table 'dp' of size m x n to store the results of subproblems.
"""


class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)

        # Create a 2D table dp to store whether s[:i] matches p[:j].
        dp = [[False] * (n + 1) for _ in range(m + 1)]

        # Base case: empty string matches empty pattern.
        dp[0][0] = True

        # Fill the first row of dp based on '*' in the pattern.
        for j in range(2, n + 1):
            if p[j - 1] == "*":
                dp[0][j] = dp[0][j - 2]

        # Fill the dp table based on characters in s and p.
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if p[j - 1] == s[i - 1] or p[j - 1] == ".":
                    dp[i][j] = dp[i - 1][j - 1]
                elif p[j - 1] == "*":
                    dp[i][j] = dp[i][j - 2] or (
                        dp[i - 1][j] and (s[i - 1] == p[j - 2] or p[j - 2] == ".")
                    )

        return dp[m][n]
```

122. 15_Greedy/01_Maximum_Subarray/0053-maximum-subarray.py

```
"""
Problem: LeetCode 53 - Maximum Subarray

Key Idea:
We can solve this problem using the Kadane's algorithm. The idea is to iterate through the array and keep track of two variables: `max_sum` which stores the maximum sum ending at the current index, and `current_sum` which stores the maximum sum of subarray ending at the current index. For each element, we update `current_sum` by taking the maximum of the current element itself or adding it to the `current_sum` of the previous index. If `current_sum` becomes negative, we reset it to 0. Meanwhile, we update `max_sum` with the maximum of `max_sum` and `current_sum` at each step. After iterating through the entire array, `max_sum` will hold the maximum subarray sum.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input array.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        max_sum = float("-inf")
        current_sum = 0

        for num in nums:
            current_sum = max(num, current_sum + num)
            max_sum = max(max_sum, current_sum)

        return max_sum
```

123. 15_Greedy/02_Jump_Game/0055-jump-game.py

```
"""
Problem: LeetCode 55 - Jump Game

Key Idea:
We can solve this problem using a greedy approach. The idea is to keep track of the maximum reachable index as we iterate through the array. At each index, we update the maximum reachable index by taking the maximum of the current index plus the value at that index (which represents the maximum jump length) and the previous maximum reachable index. If the maximum reachable index surpasses or equals the last index, we know it's possible to reach the end.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input array.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def canJump(self, nums: List[int]) -> bool:
        max_reachable = 0

        for i, jump_length in enumerate(nums):
            if i > max_reachable:
                return False
            max_reachable = max(max_reachable, i + jump_length)

        return True
```

124. 15_Greedy/03_Jump_Game_II/0045-jump-game-ii.py

```
"""
Problem: LeetCode 45 - Jump Game II

Key Idea:
We can solve this problem using a greedy approach. The idea is to keep track of the farthest reachable index and the current end of the interval. As we iterate through the array, we update the farthest reachable index based on the maximum jump length at the current index. When the current index reaches the end of the interval, we update the end of the interval to the farthest reachable index and increment the jump count. This ensures that we always make the jump with the maximum reach.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input array.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def jump(self, nums: List[int]) -> int:
        jumps = 0
        cur_end = 0
        farthest_reachable = 0

        for i in range(len(nums) - 1):
            farthest_reachable = max(farthest_reachable, i + nums[i])
            if i == cur_end:
                jumps += 1
                cur_end = farthest_reachable

        return jumps
```

125. 15_Greedy/04_Gas_Station/0134-gas-station.py

```
"""
Problem: LeetCode 134 - Gas Station

Key Idea:
We can solve this problem using a greedy approach. The idea is to keep track of the total gas available and the current gas needed as we iterate through the stations. If the current gas becomes negative, we reset the starting station to the next station and reset the current gas needed. At the end, if the total gas available is greater than or equal to the current gas needed, then the starting station is a valid solution.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input arrays.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        total_gas = 0
        current_gas = 0
        start_station = 0

        for i in range(len(gas)):
            total_gas += gas[i] - cost[i]
            current_gas += gas[i] - cost[i]

            if current_gas < 0:
                start_station = i + 1
                current_gas = 0

        return start_station if total_gas >= 0 else -1
```

126. 15_Greedy/05_Hand_of_Straights/0846-hand-of-straights.py

```
"""
Problem: LeetCode 846 - Hand of Straights

Key Idea:
We can solve this problem using a greedy approach. The idea is to sort the hand and use a Counter to keep track of the frequency of each card. Then, for each card, we check if there are enough consecutive cards to form a group of size W. If so, we decrement the frequencies accordingly. If not, the hand cannot be grouped and we return False.

Time Complexity:
- The time complexity of this approach is O(n * log n), where n is the length of the input hand due to the sorting operation.

Space Complexity:
- The space complexity is O(n), as we use a Counter to store the frequencies of the cards.
"""

from collections import Counter


class Solution:
    def isNStraightHand(self, hand: List[int], W: int) -> bool:
        if len(hand) % W != 0:
            return False

        counter = Counter(hand)
        hand.sort()

        for card in hand:
            if counter[card] > 0:
                for i in range(W):
                    if counter[card + i] <= 0:
                        return False
                    counter[card + i] -= 1

        return True
```

127. 15_Greedy/06_Merge_Triplets_to_Form_Target_Triplet/1899-merge-triplets-to-form-target-triplet.py

```
"""
Problem: LeetCode 1899 - Merge Triplets to Form Target Triplet

Key Idea:
We can solve this problem by iterating through the triplets and checking if each triplet can contribute to forming the target triplet. The idea is to keep track of the maximum values encountered for each element in the triplet. At the end, if the maximum values for each element are greater than or equal to the corresponding values in the target triplet, then it's possible to form the target triplet.

Time Complexity:
- The time complexity of this approach is O(n), where n is the number of triplets.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def mergeTriplets(self, triplets: List[List[int]], target: List[int]) -> bool:
        max_values = [0, 0, 0]

        for triplet in triplets:
            if (
                triplet[0] <= target[0]
                and triplet[1] <= target[1]
                and triplet[2] <= target[2]
            ):
                max_values = [max(max_values[i], triplet[i]) for i in range(3)]

        return max_values == target
```

128. 15_Greedy/07_Partition_Labels/0763-partition-labels.py

```
"""
Problem: LeetCode 763 - Partition Labels

Key Idea:
We can solve this problem using a greedy approach. The idea is to traverse the string and keep track of the last index at which each character appears. While traversing, we maintain a current partition's start and end indices. When the current index reaches the end index of the current partition, we add the length of the partition to the result list and update the start index for the next partition.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input string.

Space Complexity:
- The space complexity is O(1), as the result list is the only additional space used.
"""


class Solution:
    def partitionLabels(self, s: str) -> List[int]:
        last_index = {}

        for i, char in enumerate(s):
            last_index[char] = i

        result = []
        start, end = 0, 0

        for i, char in enumerate(s):
            end = max(end, last_index[char])

            if i == end:
                result.append(end - start + 1)
                start = end + 1

        return result
```

129. 15_Greedy/08_Valid_Parenthesis_String/0678-valid-parenthesis-string.py

```
"""
Problem: LeetCode 678 - Valid Parenthesis String

Key Idea:
We can solve this problem using a greedy approach. The idea is to keep track of the possible lower and upper bound of valid open parentheses count as we traverse the string. For every '(', we increment both lower and upper bounds, for every ')', we decrement both bounds, and for '*', we increment the lower bound and decrement the upper bound. At each step, we ensure that the lower bound never goes below 0.

Time Complexity:
- The time complexity of this approach is O(n), where n is the length of the input string.

Space Complexity:
- The space complexity is O(1), as we only use a constant amount of additional space for variables.
"""


class Solution:
    def checkValidString(self, s: str) -> bool:
        lower = upper = 0

        for char in s:
            if char == "(":
                lower += 1
                upper += 1
            elif char == ")":
                lower = max(lower - 1, 0)
                upper -= 1
            else:  # char == '*'
                lower = max(lower - 1, 0)
                upper += 1

            if upper < 0:
                return False

        return lower == 0
```

130. 16_Intervals/01_Insert_Interval/0057-insert-interval.py

```
"""
Problem: LeetCode 57 - Insert Interval

Key Idea:
We can solve this problem by iterating through the intervals and keeping track of the merged intervals. We initialize an empty list `result` to store the merged intervals. We iterate through the intervals, and for each interval, we compare its end with the start of the new interval and the new interval's end with the start of the current interval. If they overlap, we update the start and end of the new interval accordingly. If they don't overlap, we add the current interval to the result and reset the new interval. After iterating through all intervals, we add the new interval (if not added already) and return the result.

Time Complexity:
- The time complexity of this approach is O(n), where n is the number of intervals.

Space Complexity:
- The space complexity is O(n), as we store the merged intervals in the result list.
"""


class Solution:
    def insert(
        self, intervals: List[List[int]], newInterval: List[int]
    ) -> List[List[int]]:
        result = []
        new_start, new_end = newInterval

        for interval in intervals:
            if interval[1] < new_start:
                result.append(interval)
            elif interval[0] > new_end:
                result.append([new_start, new_end])
                new_start, new_end = interval
            else:
                new_start = min(new_start, interval[0])
                new_end = max(new_end, interval[1])

        result.append([new_start, new_end])
        return result
```

131. 16_Intervals/02_Merge_Intervals/0056-merge-intervals.py

```
"""
Problem: LeetCode 56 - Merge Intervals

Key Idea:
We can solve this problem by first sorting the intervals based on their start times. Then, we iterate through the sorted intervals and merge overlapping intervals. We initialize an empty list `result` to store the merged intervals. For each interval, if it overlaps with the previous interval (i.e., the end of the previous interval is greater than or equal to the start of the current interval), we update the end of the previous interval to be the maximum of the end of both intervals. If there is no overlap, we add the previous interval to the result and update the previous interval to be the current interval. After iterating through all intervals, we add the last interval (or the merged interval) to the result.

Time Complexity:
- The time complexity of this approach is O(n log n) due to the sorting step, where n is the number of intervals.

Space Complexity:
- The space complexity is O(n), as we store the merged intervals in the result list.
"""


class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        if not intervals:
            return []

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

        for interval in intervals[1:]:
            if interval[0] <= result[-1][1]:
                result[-1][1] = max(result[-1][1], interval[1])
            else:
                result.append(interval)

        return result
```

132. 16_Intervals/03_Non_Overlapping_Intervals/0435-non-overlapping-intervals.py

```
"""
Problem: LeetCode 435 - Non-overlapping Intervals

Key Idea:
We can solve this problem by first sorting the intervals based on their end times. Then, we iterate through the sorted intervals and count the number of overlapping intervals. If the start of the current interval is less than the end of the previous interval, it means there is an overlap, so we increment the count and skip adding the current interval to the non-overlapping set. If there is no overlap, we update the end of the previous interval to be the end of the current interval.

Time Complexity:
- The time complexity of this approach is O(n log n) due to the sorting step, where n is the number of intervals.

Space Complexity:
- The space complexity is O(1), as we use only a constant amount of extra space.
"""


class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        if not intervals:
            return 0

        intervals.sort(key=lambda x: x[1])
        non_overlapping = 1  # Count of non-overlapping intervals
        prev_end = intervals[0][1]

        for i in range(1, len(intervals)):
            if intervals[i][0] >= prev_end:
                non_overlapping += 1
                prev_end = intervals[i][1]

        return len(intervals) - non_overlapping
```

133. 16_Intervals/04_Meeting_Rooms/0252-meeting-rooms.py

```
"""
Problem: LeetCode 252 - Meeting Rooms

Key Idea:
We can solve this problem by sorting the intervals based on their start times and then checking for any overlapping intervals. If the start time of the current interval is less than the end time of the previous interval, it means there is an overlap, and the person cannot attend all meetings. Otherwise, there is no overlap, and the person can attend all meetings.

Time Complexity:
- The time complexity of this approach is O(n log n) due to the sorting step, where n is the number of intervals.

Space Complexity:
- The space complexity is O(1), as we use only a constant amount of extra space.
"""


class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        if not intervals:
            return True

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

        for i in range(1, len(intervals)):
            if intervals[i][0] < intervals[i - 1][1]:
                return False

        return True
```

134. 16_Intervals/05_Meeting_Rooms_II/0253-meeting-rooms-ii.py

```
"""
Problem: LeetCode 253 - Meeting Rooms II

Key Idea:
We can solve this problem using a priority queue (min-heap). First, we sort the intervals based on their start times. We then iterate through the sorted intervals and maintain a min-heap of end times of ongoing meetings. For each interval, if the start time is greater than or equal to the smallest end time in the min-heap, it means the current meeting can reuse an existing room, so we pop the smallest end time from the min-heap. If not, we need to allocate a new room. After processing all intervals, the size of the min-heap gives us the minimum number of meeting rooms required.

Time Complexity:
- The time complexity of this approach is O(n log n) due to the sorting step and the heap operations, where n is the number of intervals.

Space Complexity:
- The space complexity is O(n), as we store the end times in the min-heap.
"""

import heapq


class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        if not intervals:
            return 0

        intervals.sort(key=lambda x: x[0])
        min_heap = []
        heapq.heappush(min_heap, intervals[0][1])

        for i in range(1, len(intervals)):
            if intervals[i][0] >= min_heap[0]:
                heapq.heappop(min_heap)
            heapq.heappush(min_heap, intervals[i][1])

        return len(min_heap)
```

135. 16_Intervals/06_Minimum_Interval_to_Include_Each_Query/1851-minimum-interval-to-include-each-query.py

```
"""
Problem: LeetCode 1851 - Minimum Interval to Include Each Query

Key Idea:
The problem is to find the minimum interval for each query that includes at least one element from each interval in the list. We can solve this problem using a sorted list or priority queue (min-heap). First, we sort both the intervals and the queries based on their start values. Then, for each query, we iterate through the sorted intervals and add intervals to the min-heap as long as their start values are less than or equal to the current query. We also remove intervals from the min-heap whose end values are less than the start value of the current query. After processing all queries, the size of the min-heap gives us the minimum interval for each query.

Time Complexity:
- The time complexity of this approach is O((n + q) log n), where n is the number of intervals and q is the number of queries, due to the sorting and heap operations.

Space Complexity:
- The space complexity is O(n), as we store the intervals in the min-heap.
"""

import heapq


class Solution:
    def minInterval(self, intervals: List[List[int]], queries: List[int]) -> List[int]:
        intervals.sort(key=lambda x: x[0])
        queries_sorted = sorted(enumerate(queries), key=lambda x: x[1])

        min_heap = []
        ans = [-1] * len(queries)
        i = 0

        for query_index, query in queries_sorted:
            while i < len(intervals) and intervals[i][0] <= query:
                start, end = intervals[i]
                heapq.heappush(min_heap, (end - start + 1, end))
                i += 1

            while min_heap and min_heap[0][1] < query:
                heapq.heappop(min_heap)

            if min_heap:
                ans[query_index] = min_heap[0][0]

        return ans
```

136. 17_Math_&_Geometry/01_Rotate_Image/0048-rotate-image.py

```
"""
Problem: LeetCode 48 - Rotate Image

Key Idea:
To rotate the given matrix in-place, we can perform a series of swaps. We start by swapping elements symmetrically along the diagonal, and then swap elements symmetrically along the vertical midline. This process rotates the matrix by 90 degrees clockwise.

Time Complexity:
- We perform swaps for each element in the matrix exactly once. Therefore, the time complexity is O(n^2), where n is the number of rows/columns in the matrix.

Space Complexity:
- The space complexity is O(1), as we perform swaps in-place without using any additional space.
"""


class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)

        # Transpose the matrix
        for i in range(n):
            for j in range(i, n):
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

        # Reverse each row
        for i in range(n):
            matrix[i].reverse()
```

137. 17_Math_&_Geometry/02_Spiral_Matrix/0054-spiral-matrix.py

```
"""
Problem: LeetCode 54 - Spiral Matrix

Key Idea:
To traverse a matrix in a spiral order, we can iterate through each layer of the matrix and extract the elements in the desired order: top row, right column, bottom row, and left column. We update the boundaries of each layer as we traverse.

Time Complexity:
- We visit each element in the matrix exactly once. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix.

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space to store the result.
"""


class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        if not matrix:
            return []

        rows, cols = len(matrix), len(matrix[0])
        result = []

        # Define the boundaries of the current layer
        top, bottom, left, right = 0, rows - 1, 0, cols - 1

        while top <= bottom and left <= right:
            # Traverse the top row
            for j in range(left, right + 1):
                result.append(matrix[top][j])
            top += 1

            # Traverse the right column
            for i in range(top, bottom + 1):
                result.append(matrix[i][right])
            right -= 1

            # Traverse the bottom row
            if top <= bottom:  # Avoid duplicate traversal
                for j in range(right, left - 1, -1):
                    result.append(matrix[bottom][j])
                bottom -= 1

            # Traverse the left column
            if left <= right:  # Avoid duplicate traversal
                for i in range(bottom, top - 1, -1):
                    result.append(matrix[i][left])
                left += 1

        return result
```

138. 17_Math_&_Geometry/03_Set_Matrix_Zeroes/0073-set-matrix-zeroes.py

```
"""
Problem: LeetCode 73 - Set Matrix Zeroes

Key Idea:
To set the entire row and column to zeros if an element in the matrix is zero, we can use two sets to keep track of the rows and columns that need to be set to zero. We iterate through the matrix and mark the corresponding rows and columns in the sets. Then, we iterate through the matrix again and set the elements to zero if their row or column is marked.

Time Complexity:
- We iterate through the matrix twice: once to mark the rows and columns and once to set the elements to zero. Therefore, the time complexity is O(m * n), where m is the number of rows and n is the number of columns in the matrix.

Space Complexity:
- The space complexity is O(m + n), as we use two sets to store the rows and columns that need to be set to zero.
"""


class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        rows, cols = len(matrix), len(matrix[0])
        zero_rows, zero_cols = set(), set()

        # Mark rows and columns that need to be set to zero
        for i in range(rows):
            for j in range(cols):
                if matrix[i][j] == 0:
                    zero_rows.add(i)
                    zero_cols.add(j)

        # Set elements to zero based on marked rows and columns
        for i in range(rows):
            for j in range(cols):
                if i in zero_rows or j in zero_cols:
                    matrix[i][j] = 0
```

139. 17_Math_&_Geometry/04_Happy_Number/0202-happy-number.py

```
"""
Problem: LeetCode 202 - Happy Number

Key Idea:
A number is a happy number if, after repeatedly replacing it with the sum of the square of its digits, the process eventually reaches 1. To determine if a number is a happy number, we can use Floyd's Cycle Detection Algorithm. We use two pointers: a slow pointer that advances one step at a time and a fast pointer that advances two steps at a time. If the fast pointer eventually reaches the slow pointer, we have a cycle, indicating that the number is not a happy number.

Time Complexity:
- The time complexity of the Floyd's Cycle Detection Algorithm is O(log n), where n is the input number.

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def isHappy(self, n: int) -> bool:
        def get_next(num):
            next_num = 0
            while num > 0:
                num, digit = divmod(num, 10)
                next_num += digit**2
            return next_num

        slow, fast = n, get_next(n)

        while fast != 1 and slow != fast:
            slow = get_next(slow)
            fast = get_next(get_next(fast))

        return fast == 1
```

140. 17_Math_&_Geometry/05_Plus_One/0066-plus-one.py

```
"""
Problem: LeetCode 66 - Plus One

Key Idea:
Given a non-empty array representing a non-negative integer, we need to add one to the integer. We can perform this addition by starting from the least significant digit (the last element of the array) and moving towards the most significant digit. If the current digit is less than 9, we can simply increment it and return the modified array. Otherwise, we set the current digit to 0 and continue the process with the previous digit.

Time Complexity:
- The time complexity is O(n), where n is the length of the input array.

Space Complexity:
- The space complexity is O(1), as we perform the addition in-place without using any additional space.
"""


class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        n = len(digits)

        for i in range(n - 1, -1, -1):
            if digits[i] < 9:
                digits[i] += 1
                return digits
            digits[i] = 0

        return [1] + digits
```

141. 17_Math_&_Geometry/06_Pow(x,n)/0050-powx-n.py

```
"""
Problem: LeetCode 50 - Pow(x, n)

Key Idea:
The problem is to calculate x raised to the power n. We can solve this problem using the concept of exponentiation by squaring. If n is even, we can compute x^n as (x^(n//2)) * (x^(n//2)). If n is odd, we can compute x^n as x * (x^(n//2)) * (x^(n//2)).

Time Complexity:
- The time complexity of this approach is O(log n), where n is the exponent.

Space Complexity:
- The space complexity is O(log n), as we use the call stack for the recursive calculations.
"""


class Solution:
    def myPow(self, x: float, n: int) -> float:
        def helper(base, exp):
            if exp == 0:
                return 1.0
            temp = helper(base, exp // 2)
            if exp % 2 == 0:
                return temp * temp
            else:
                return base * temp * temp

        if n < 0:
            x = 1 / x
            n = -n

        return helper(x, n)
```

142. 17_Math_&_Geometry/07_Multiply_Strings/0043-multiply-strings.py

```
"""
Problem: LeetCode 43 - Multiple Strings

Key Idea:
The problem is to multiply two given non-negative integers represented as strings. We can perform multiplication digit by digit, similar to how we do multiplication manually. We start from the least significant digits and move towards the most significant digits. We use two nested loops to multiply each pair of digits and keep track of carry values.

Time Complexity:
- The time complexity of this approach is O(m * n), where m and n are the lengths of the input strings.

Space Complexity:
- The space complexity is O(m + n), as the length of the result can be at most m + n digits.
"""


class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        if num1 == "0" or num2 == "0":
            return "0"

        m, n = len(num1), len(num2)
        result = [0] * (m + n)

        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                product = int(num1[i]) * int(num2[j])
                sum_val = product + result[i + j + 1]
                result[i + j + 1] = sum_val % 10
                result[i + j] += sum_val // 10

        return "".join(map(str, result)).lstrip("0")
```

143. 17_Math_&_Geometry/08_Detect_Squares/2013-detect-squares.py

```
"""
Problem: LeetCode 2013 - Detect Squares

Key Idea:
The key idea is to group points by their x-coordinates and then use these groups to efficiently find potential
square corners.

Time Complexity:
- Adding a point: O(1)
- Counting squares: O(K) in the worst case

Space Complexity:
- O(N) due to storing up to N points
"""

from collections import defaultdict
from typing import List


class DetectSquares:
    def __init__(self):
        self.points = defaultdict(lambda: defaultdict(int))

    def add(self, point: List[int]) -> None:
        x, y = point
        self.points[x][y] += 1

    def count(self, point: List[int]) -> int:
        x, y = point
        count = 0

        for y2 in self.points[x]:
            if y2 != y:
                side_length = abs(y2 - y)

                for x2 in (x + side_length, x - side_length):
                    if x2 in self.points and y in self.points[x2]:
                        count += (
                            self.points[x2][y]
                            * self.points[x2][y2]
                            * self.points[x][y2]
                        )

        return count


# Your DetectSquares object will be instantiated and called as such:
# obj = DetectSquares()
# obj.add(point)
# param_2 = obj.count(point)
```

144. 18_Bit_Manipulation/01_Single_Number/0136-single-number.py

```
"""
Problem: LeetCode 136 - Single Number

Key Idea:
To find the single number in an array where all other numbers appear twice, we can use the XOR operation. XORing a number with itself results in 0, and XORing any number with 0 results in the number itself. Therefore, if we XOR all the numbers in the array, the duplicates will cancel out, leaving only the single number.

Time Complexity:
- The XOR operation has a time complexity of O(n), where n is the number of elements in the array.

Space Complexity:
- The space complexity is O(1), as we only need a constant amount of space to store the XOR result.
"""


class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        result = 0
        for num in nums:
            result ^= num
        return result
```

145. 18_Bit_Manipulation/02_Number_of_1_Bits/0191-number-of-1-bits.py

```
"""
Problem: LeetCode 191 - Number of 1 Bits

Key Idea:
To count the number of 1 bits in an unsigned integer, we can use bit manipulation. We iterate through each bit of the number and use a bitwise AND operation with 1 to check if the least significant bit is 1. If it is, we increment the count and right-shift the number by 1 to check the next bit.

Time Complexity:
- The number of bits in an unsigned integer is constant (32 bits for 32-bit integers and 64 bits for 64-bit integers). Therefore, the time complexity is O(1).

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def hammingWeight(self, n: int) -> int:
        count = 0
        while n:
            count += n & 1
            n = n >> 1
        return count
```

146. 18_Bit_Manipulation/03_Counting_Bits/0338-counting-bits.py

```
"""
Problem: LeetCode 338 - Counting Bits

Key Idea:
To count the number of 1 bits for each number in the range [0, num], we can use dynamic programming. We observe that the number of 1 bits in a number x is equal to the number of 1 bits in x // 2 plus the value of the least significant bit (x % 2).

Time Complexity:
- We iterate through each number in the range [0, num] and perform constant-time operations for each number. Therefore, the time complexity is O(n), where n is the value of 'num'.

Space Complexity:
- The space complexity is O(n), as we use an array of size 'num + 1' to store the count of 1 bits for each number in the range.
"""


class Solution:
    def countBits(self, num: int) -> List[int]:
        bits_count = [0] * (num + 1)

        for i in range(1, num + 1):
            bits_count[i] = bits_count[i // 2] + (i % 2)

        return bits_count
```

147. 18_Bit_Manipulation/04_Reverse_Bits/0190-reverse-bits.py

```
"""
Problem: LeetCode 190 - Reverse Bits

Key Idea:
To reverse the bits of a given unsigned integer, we can iterate through each bit of the input number. For each bit, we use bitwise operations to reverse the bit and append it to the result.

Time Complexity:
- The number of bits in an unsigned integer is constant (32 bits for 32-bit integers and 64 bits for 64-bit integers). Therefore, the time complexity is O(1).

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def reverseBits(self, n: int) -> int:
        reversed_num = 0
        for _ in range(32):
            reversed_num = (reversed_num << 1) | (n & 1)
            n = n >> 1
        return reversed_num
```

148. 18_Bit_Manipulation/05_Missing_Number/0268-missing-number.py

```
"""
Problem: LeetCode 268 - Missing Number

Key Idea:
To find the missing number in an array containing distinct numbers from 0 to n, we can use the XOR operation. We XOR all the numbers from 0 to n and then XOR the result with all the elements in the array. The XOR operation cancels out the duplicate numbers, leaving only the missing number.

Time Complexity:
- XORing all the numbers from 0 to n takes O(n) time, and XORing all the elements in the array also takes O(n) time. Therefore, the total time complexity is O(n).

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        n = len(nums)
        missing_num = n

        for i in range(n):
            missing_num ^= i ^ nums[i]

        return missing_num
```

149. 18_Bit_Manipulation/06_Sum_of_Two_Integers/0371-sum-of-two-integers.py

```
"""
Problem: LeetCode 371 - Sum of Two Integers

Key Idea:
To calculate the sum of two integers without using the + and - operators, we can use bitwise operations. We use the XOR operation (^) to calculate the sum without considering the carry, and the AND operation (&) followed by a left shift (<<) to calculate the carry. We repeat these steps until there is no carry left.

Time Complexity:
- The number of iterations required to calculate the sum is proportional to the number of bits in the input integers, which is constant. Therefore, the time complexity is O(1).

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def getSum(self, a: int, b: int) -> int:
        MASK = 0xFFFFFFFF
        MAX_INT = 0x7FFFFFFF

        while b != 0:
            a, b = (a ^ b) & MASK, ((a & b) << 1) & MASK

        return a if a <= MAX_INT else ~(a ^ MASK)
```

150. 18_Bit_Manipulation/07_Reverse_Integer/0007-reverse-integer.py

```
"""
Problem: LeetCode 7 - Reverse Integer

Key Idea:
To reverse an integer, we can use integer arithmetic. We repeatedly extract the last digit of the number using the modulo operator (%) and add it to the reversed number after multiplying by 10. We then update the original number by integer division (//) to remove the last digit. We continue this process until the original number becomes 0.

Time Complexity:
- The time complexity is O(log(x)), where x is the input integer. We perform operations on each digit of the integer.

Space Complexity:
- The space complexity is O(1), as we use a constant amount of space.
"""


class Solution:
    def reverse(self, x: int) -> int:
        INT_MAX = 2**31 - 1
        INT_MIN = -(2**31)

        reversed_num = 0
        sign = 1 if x > 0 else -1
        x = abs(x)

        while x != 0:
            pop = x % 10
            x //= 10

            if reversed_num > (INT_MAX - pop) // 10:
                return 0

            reversed_num = reversed_num * 10 + pop

        return reversed_num * sign
```

151. misc/relative_file_paths.py

```
import os

directory = "../"
output_file = "output.txt"

with open(output_file, "w") as f:
    for root, dirs, files in os.walk(directory):
        for file in files:
            relative_path = os.path.relpath(os.path.join(root, file), directory)
            relative_path = "./" + relative_path
            f.write(relative_path + "\n")
```