Problem 1 :

def quicksort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quicksort(left) + middle + quicksort(right)  
  
**Time Complexity Analysis:**

* **Best and Average Case:**
  + If the pivot splits the array into two equal halves, the recurrence relation for the time complexity is: T(n)=2T(n2)+O(n)T(n) = 2T\left(\frac{n}{2}\right) + O(n)T(n)=2T(2n​)+O(n)
  + Solving this recurrence relation using the Master Theorem gives: T(n)=O(nlog⁡n)T(n) = O(n \log n)T(n)=O(nlogn)
  + This is the time complexity in the best and average cases, where the array is divided into approximately equal halves.
* **Worst Case:**
  + The worst case occurs when the pivot is the smallest or largest element, leading to highly unbalanced partitions (e.g., one subarray of size n-1 and the other of size 0).
  + The recurrence relation in this case is: T(n)=T(n−1)+O(n)T(n) = T(n-1) + O(n)T(n)=T(n−1)+O(n)
  + Solving this gives: T(n)=O(n2)T(n) = O(n^2)T(n)=O(n2)
  + This is the time complexity in the worst case, which happens when the array is already sorted or reverse sorted.

Problem 2 :

def nested\_loop\_example(matrix):

rows, cols = len(matrix), len(matrix[0])

total = 0

for i in range(rows):

for j in range(cols):

total += matrix[i][j]

return total

**Solution:-**

The time complexity of the given code is **O(rows \* cols)**.

This is because the code iterates through each element of the matrix using two nested loops: the outer loop runs rows times, and the inner loop runs cols times, leading to a total of rows \* cols iterations.

Problem 3 :

def example\_function(arr):

result = 0

for element in arr:

result += element

return result

**Solution:-**

The time complexity of the given code is O(n), where n is the length of the array arr.

This is because the code iterates through each element of the array exactly once, performing a constant-time operation (addition) for each element.

Problem 4 :

def longest\_increasing\_subsequence(nums):

n = len(nums)

lis = [1] \* n

for i in range(1, n):

for j in range(0, i):

if nums[i] > nums[j] and lis[i] < lis[j] + 1:

lis[i] = lis[j] + 1

return max(lis)

**Solution:-**

The time complexity of the given code is **O(n^2)**, where n is the length of the input list nums.

This is because there are two nested loops: the outer loop runs n times, and for each iteration of the outer loop, the inner loop runs up to i times, leading to a quadratic time complexity.

Problem 5 :

def mysterious\_function(arr):

n = len(arr)

result = 0

for i in range(n):

for j in range(i, n):

result += arr[i] \* arr[j]

return result

**Solution:-**

The time complexity of the given code is **O(n^2)**, where n is the length of the input list arr.

This is because there are two nested loops, and each loop runs up to n times, leading to a quadratic time complexity.

**Solve the following problems on recursion**

Problem 6 : Sum of Digits Write a recursive function to calculate the sum of digits of a given positive integer.

sum\_of\_digits(123) -> 6

**Solution:-**def sum\_of\_digits(n):

# Base case: if n is a single-digit number

if n < 10:

return n

# Recursive case: sum of last digit and the sum of digits of the remaining number

return n % 10 + sum\_of\_digits(n // 10)

# Example usage

print(sum\_of\_digits(123)) # Output: 6

Problem 7: Fibonacci Series

Write a recursive function to generate the first n numbers of the Fibonacci series.

fibonacci\_series(6) -> [0, 1, 1, 2, 3, 5]

**Solution:-**

def fibonacci\_series(n):

def fibonacci(n, memo={0: 0, 1: 1}):

if n not in memo:

memo[n] = fibonacci(n - 1) + fibonacci(n - 2)

return memo[n]

return [fibonacci(i) for i in range(n)]

# Example usage

print(fibonacci\_series(6)) # Output: [0, 1, 1, 2, 3, 5]

Problem 8 : Subset Sum

Given a set of positive integers and a target sum, write a recursive function to determine if there exists a subset

of the integers that adds up to the target sum.

subset\_sum([3, 34, 4, 12, 5, 2], 9) -> True

**Solution:-**

def subset\_sum(nums, target):

def helper(index, remaining\_sum):

# Base cases

if remaining\_sum == 0:

return True

if index >= len(nums) or remaining\_sum < 0:

return False

# Include the current number and check

include = helper(index + 1, remaining\_sum - nums[index])

# Exclude the current number and check

exclude = helper(index + 1, remaining\_sum)

return include or exclude

return helper(0, target)

# Example usage

print(subset\_sum([3, 34, 4, 12, 5, 2], 9)) # Output: True

Problem 9: Word Break

Given a non-empty string and a dictionary of words, write a recursive function to determine if the string can be

segmented into a space-separated sequence of dictionary words.

word\_break( leetcode , [ leet , code ]) -> True

**Solution:-**

def word\_break(s, word\_dict):

def can\_break(start\_index):

# Base case: If we've reached the end of the string

if start\_index == len(s):

return True

# Try every possible end index for the current prefix

for end\_index in range(start\_index + 1, len(s) + 1):

prefix = s[start\_index:end\_index]

if prefix in word\_dict and can\_break(end\_index):

return True

return False

# Convert the dictionary to a set for faster look-up

word\_dict = set(word\_dict)

return can\_break(0)

# Example usage

print(word\_break("leetcode", ["leet", "code"])) # Output: True

Problem 10 : N-Queens

Implement a recursive function to solve the N Queens problem, where you have to place N queens on an N×N

chessboard in such a way that no two queens threaten each other.

n\_queens(4)

[

[".Q..",

"...Q",

"Q...",

"..Q."],

["..Q.",

"Q...",

"...Q",

".Q.."]

]

**Solution:-**

def n\_queens(n):

def solve(board, row):

# Base case: If all queens are placed successfully

if row == n:

result.append([''.join(row) for row in board])

return

# Try placing a queen in each column of the current row

for col in range(n):

if is\_valid(board, row, col):

board[row][col] = 'Q'

solve(board, row + 1)

board[row][col] = '.' # Backtrack

def is\_valid(board, row, col):

# Check this column

for i in range(row):

if board[i][col] == 'Q':

return False

# Check upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 'Q':

return False

# Check upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j] == 'Q':

return False

return True

result = []

board = [['.' for \_ in range(n)] for \_ in range(n)]

solve(board, 0)

return result

# Example usage

print(n\_queens(4))