Find time complexity of below code blocks :

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)
```

The given code block implements the **Quicksort** algorithm. Let's analyze its time complexity.

Best case:

- The pivot chosen is always the median element.
- This results in the array being divided into two roughly equal halves at each recursion.
- The recurrence relation becomes: $T(n) = 2T(n/2) + O(n)$
- Using the Master theorem, the time complexity is $O(n \log n)$.

Worst case:

- The pivot chosen is always the smallest or largest element.
- This results in one subarray being empty and the other containing n-1 elements.
- The recurrence relation becomes: $T(n) = T(n-1) + O(n)$
- This solves to $O(n^2)$.

Average case:

- The pivot is chosen randomly.
- The average case is closer to the best case, but it can still be $O(n^2)$ in the worst case.

Overall: The time complexity of Quicksort is **$O(n \log n)$** in the average case, but it can degenerate to **$O(n^2)$** in the worst case. The choice of pivot significantly affects the performance.

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
```

The given code block implements a nested loop to calculate the sum of all elements in a matrix. Let's analyze its time complexity.

Time complexity:

- The outer loop iterates rows times.
- The inner loop iterates cols times for each iteration of the outer loop.
- The total number of iterations is rows * cols.
- The operations inside the inner loop (accessing matrix elements and adding to total) are constant time operations.

Therefore, the time complexity of the nested loop is **O(rows * cols)**.

In other words, the time complexity is proportional to the number of elements in the matrix. This is the expected time complexity for iterating over all elements in a matrix.

Problem 3 :

```
def example_function(arr):

result = 0

for element in arr:

result += element

return result
```

The given code block implements a simple function to calculate the sum of all elements in an array. Let's analyze its time complexity.

Time complexity:

- The loop iterates len(arr) times, where len(arr) is the length of the array.
- The operations inside the loop (accessing array elements and adding to result) are constant time operations.

Therefore, the time complexity of the function is **O(len(arr))**, or simply **O(n)** where n is the size of the array.

This means that the time taken by the function will increase linearly with the size of the input array.

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)

The given code block implements the dynamic programming approach to find the length of the longest increasing subsequence (LIS) in an array. Let's analyze its time complexity.

Time complexity:

- The outer loop iterates n times, where n is the length of the array.
- The inner loop iterates i times for each iteration of the outer loop.
- The operations inside the inner loop (comparing elements and updating lis) are constant time operations.

Therefore, the time complexity can be expressed as:

$T(n) = \Sigma(i=1 \text{ to } n)\ \Sigma(j=0 \text{ to } i-1)\ O(1)$

Simplifying the nested summation:

$T(n) = \Sigma(i=1 \text{ to } n)\ i$

Using the formula for the sum of the first n natural numbers:

$T(n) = n(n+1)/2$

Therefore, the time complexity of the algorithm is **O(n^2)**.

This is because the inner loop iterates a varying number of times for each iteration of the outer loop, resulting in 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
```

The given code block implements a nested loop that calculates a specific sum involving the elements of an array. Let's analyze its time complexity.

Time complexity:

- The outer loop iterates n times, where n is the length of the array.
- The inner loop iterates from i to n-1 for each iteration of the outer loop.
- The operations inside the inner loop (accessing array elements, multiplying, and adding to result) are constant time operations.

Therefore, the time complexity can be expressed as:

$T(n) = \Sigma(i=0 \text{ to } n-1)\ \Sigma(j=i \text{ to } n-1)\ O(1)$

Simplifying the nested summation:

$T(n) = \Sigma(i=0 \text{ to } n-1)\ (n - i)$

Using the formula for the sum of the first n natural numbers and the sum of the squares of the first n natural numbers:

$T(n) = n^2/2 + n/2$

Therefore, the time complexity of the algorithm is **O(n^2)**.

This is because the inner loop iterates a varying number of times for each iteration of the outer loop, resulting in a quadratic time complexity.

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

```
[1]: def sum_of_digits(num):
       if num == 0:
         return 0
       else:
         return num % 10 + sum_of_digits(num // 10)

     # Example usage:
     result = sum_of_digits(123)
     print(result)
```

```
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]

```
[4]: def fibonacci_series(n):
         def fibonacci_recursive(x):
             if x <= 1:
                 return x
             else:
                 return fibonacci_recursive(x - 1) + fibonacci_recursive(x - 2)

         def generate_series(length):
             if length == 0:
                 return []
             elif length == 1:
                 return [0]
             elif length == 2:
                 return [0, 1]
             else:
                 series = generate_series(length - 1)
                 series.append(fibonacci_recursive(length - 1))
                 return series

         return generate_series(n)

     # Example usage
     print(fibonacci_series(6))  # Output: [0, 1, 1, 2, 3, 5]
```

```
[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

[5]:
```python
def subset_sum(numbers, target_sum):
    def helper(index, current_sum):
        # Base cases
        if current_sum == 0:
            return True
        if index == len(numbers):
            return False

        # Include the current number and check if we can find the subset
        if helper(index + 1, current_sum - numbers[index]):
            return True

        # Exclude the current number and check if we can find the subset
        return helper(index + 1, current_sum)

    return helper(0, target_sum)

# Example usage
print(subset_sum([3, 34, 4, 12, 5, 2], 9))  # Output: True
```

```
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

```
[6]: def word_break(s, word_dict):
    def can_segment(start_index):
        # Base case: If we reached the end of the string
        if start_index == len(s):
            return True

        # Try every substring starting from start_index
        for end_index in range(start_index + 1, len(s) + 1):
            # Check if the substring is in the dictionary
            if s[start_index:end_index] in word_dict:
                # Recursively check the rest of the string
                if can_segment(end_index):
                    return True

        # If no valid segmentation is found
        return False

    return can_segment(0)

# Example usage
print(word_break("leetcode", ["leet", "code"]))  # Output: True
```

```
True
```

```
[ ]:
```

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.."]

]

```python
[8]: def n_queens(n):
    def is_safe(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

    def solve(board, row):
        if row >= n:
            # Add the current solution to the result list
            result.append(["".join(row) for row in board])
            return

        for col in range(n):
            if is_safe(board, row, col):
                board[row][col] = 'Q'
                solve(board, row + 1)
                board[row][col] = '.'  # Backtrack

    result = []
    board = [['.' for _ in range(n)] for _ in range(n)]
    solve(board, 0)
    return result

# Example usage
print(n_queens(4))
```

```
[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']]
```