lexity-and-recursion-assignment-15

August 9, 2024

[1]:
```
'''24th_Jan^24_Time_Complexity_Assignment_15'''
```

[1]: '24th_Jan^24_Time_Complexity_Assignment_15'

[3]:
```
#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 time complexity of quicksort can vary depending on the input and pivot␣
↪selection. It can be O(n log n) in the best
and average cases, and O(n^2) in the worst case.'''
```

[3]: 'The time complexity of quicksort can vary depending on the input and pivot selection. It can be O(n log n) in the best\nand average cases, and O(n^2) in the worst case.'

[4]:
```
#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 time complexity of the `nested_loop_example(matrix)` function is O(n^2)␣
↪for the worst-case scenario.
In the best-case scenario, where the number of rows is represented by 'n' and␣
↪the number of columns is represented by
'm', the time complexity is O(m * n).'''
```

[4]: "The time complexity of the `nested_loop_example(matrix)` function is O(n^2) for the worst-case scenario.\nIn the best-case scenario, where the number of rows is represented by 'n' and the number of columns is represented by\n'm', the time complexity is O(m * n)."

[5]:
```python
#Problem 3 :
def example_function(arr):
    result = 0
    for element in arr:
        result += element
    return result
"""The time complexity for the "example_function(arr)" depends on the length of␣
↪arr or size of input more precisely...
hence, the function having O(n) as time complexity in worst case senario."""
```

[5]: 'The time complexity for the "example_function(arr)" depends on the length of arr or size of input more precisely…\nhence, the function having O(n) as time complexity in worst case senario.'

[6]:
```python
#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 time complexity for the "longest_increasingly_subsequence(nums)" is␣
↪O(n^2) due to two (nested) for loops in worst
case senario which resulted in quadratice time complexity...hence, the function␣
↪is having O(n^2) as time complexity.'''
```

[6]: 'The time complexity for the "longest_increasingly_subsequence(nums)" is O(n^2) due to two (nested) for loops in worst\ncase senario which resulted in quadratice time complexity…hence, the function is having O(n^2) as time complexity.'

[7]:
```python
#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
```

```
arr=list(range(2,10))
print(mysterious_function(arr))
''' The time complexity of the `mysterious_function(arr)` is indeed O(n^2) in␣
↪the worst-case scenario. The nested for
loops contribute to the quadratic time complexity.'''
```

```
88
```

[7]:
```
' The time complexity of the `mysterious_function(arr)` is indeed O(n^2) in the
worst-case scenario. The nested for\nloops contribute to the quadratic time
complexity.'
```

[8]:
```
#Solve the following problems on recursion
```

[9]:
```
'''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'''
def sum_of_digit(n):
    if n<=9:
        return n
    else:
        return (n%10)+sum_of_digit(n//10)
print(f"sum_of_digit(123) -> {sum_of_digit(123)}")
```

```
sum_of_digit(123) -> 6
```

[14]:
```
'''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]'''
#program a function to check for fibonacci number
def fib(n):
    if n<=1:
        return n
    else:
        return fib(n-1)+fib(n-2)
#Programme to append create series
def fibonacci_series(n):
    series=[]
    for i in range(n):
        series.append(fib(i))
    return series
print(f"fibonacci_series(6) -> {fibonacci_series(6)}")
```

```
fibonacci_series(6) -> [0, 1, 1, 2, 3, 5]
```

[4]:
```python
"""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"""
def subset_sum(numbers, target_sum):
    if target_sum == 0:
        return True
    if not numbers or target_sum < 0:
        return False
    return subset_sum(numbers[1:], target_sum - numbers[0]) or␣
↪subset_sum(numbers[1:], target_sum)

print(f"subset_sum([3, 34, 4, 12, 5, 2],9) -> {subset_sum([3, 34, 4, 12, 5,␣
↪2],9)}")
```

```
subset_sum([3, 34, 4, 12, 5, 2],9) -> True
```

[7]:
```python
'''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'''
def Word_Break(s,word_dict):
    if s in word_dict:
        return True

    for i in range(1,len(s)):
        suffix=s[i:]
        prefix=s[:i]
        if prefix in word_dict and Word_Break(suffix,word_dict):
            return True
    return False
string = "leetcode"
dictionary = ["leet", "code"]
result = Word_Break(string, dictionary)
print(result)
```

```
True
```

[9]:
```python
'''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.."]
]
'''
def solve_n_queens(n):
    def is_valid(board, row, col):
        # Check if placing a queen at (row, col) is valid
        for i in range(row):
            if board[i] == col or board[i] - col == i - row or board[i] - col␣
 ↪== row - i:
                return False
        return True

    def backtrack(board, row):
        if row == n:
            #All rows have been filled, add the board configuration to the␣
 ↪solutions
            solutions.append(board[:])
            return

        for col in range(n):
            if is_valid(board, row, col):
                board[row] = col
                backtrack(board, row + 1)
                board[row] = -1

    #Initialize the board
    board = [-1] * n
    solutions = []

    # Start the backtracking from the first row
    backtrack(board, 0)

    return solutions
```

```python
n = 4
solutions = solve_n_queens(n)
for solution in solutions:
    for row in solution:
        line = ""
        for col in range(n):
            if col == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()
```

```
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

[ ]:

```python

```