FOA Practical

Q1) Finding Specific Weight of Hen – Binary Search Algorithm Will Use.

It seems like you're describing a variant of a classic problem known as the "**Egg Dropping Puzzle**" or "Two Egg Problem." In this problem, you have a certain number of eggs (hens in your case) and you need to determine the lowest floor from which an egg will break when dropped from a building, given a limited number of drops.

In your scenario, instead of floors, you have hens, and instead of eggs breaking, you're dealing with weight. The objective is still to minimize the number of attempts (or in this case, separations) needed to determine the specific weight of a hen.

One approach you could use is a **binary search algorithm**. Here's how it might be adapted to your problem:

Step 1 : Initial Separation: First, separate the hens into two groups of roughly equal size.

Step 2 : Weighing: Weigh each group separately. If the weight is less than 1.5 kg, it means the specific hen is in one of these groups.

Step 3 : Recursive Separation: For the group that contains the hen, repeat the process by further dividing it into two smaller groups and weighing each until you narrow down to the specific hen.

This approach is similar to **binary search** because with each weighing, you're effectively halving the search space. It's efficient because it reduces the number of weighings needed to find the specific hen.

Here's a Pythonic pseudocode representation of this approach:

```
import random

def find_specific_hen(hens, target_weight):
    # Initial separation into two groups
    group1, group2 = separate_hens(hens)

    # Weighing the first two groups
    weight1 = weigh(group1)
    weight2 = weigh(group2)

    if weight1 < target_weight:
        # Hen is in group1, recursively search within group1
        return find_specific_hen_recursive(group1, target_weight)
    else:
        # Hen is in group2, recursively search within group2
        return find_specific_hen_recursive(group2, target_weight)

def find_specific_hen_recursive(group, target_weight):
    # Base case: group contains only one hen
    if len(group) == 1:
        return group[0]

    # Separate the group into two subgroups
    subgroup1, subgroup2 = separate_hens(group)

    # Weighing the subgroups
    weight1 = weigh(subgroup1)
    weight2 = weigh(subgroup2)

    if weight1 < target_weight:
        # Hen is in subgroup1, recursively search within subgroup1
        return find_specific_hen_recursive(subgroup1, target_weight)
    else:
        # Hen is in subgroup2, recursively search within subgroup2
        return find_specific_hen_recursive(subgroup2, target_weight)

# Function to separate hens into two groups
def separate_hens(hens):
    # Implement your separation logic here
    pass

# Function to weigh a group of hens
def weigh(group):
```

```
    # Implement your weighing logic here
    pass


# Function to separate hens into two groups
def separate_hens(hens):
    # Shuffle the hens to make the separation random
    random.shuffle(hens)

    # Divide the hens into two groups of roughly equal size
    mid = len(hens) // 2
    group1 = hens[:mid]
    group2 = hens[mid:]

    return group1, group2

# Function to weigh a group of hens
def weigh(group):
    # For simplicity, let's assume the weight of the group is the sum of weights
of all hens in the group
    total_weight = sum(group)
    return total_weight

# Example usage:
hens = [1.3, 1.4, 1.5, 1.6, 1.7]  # Example list of hens with weights
target_weight = float(input("Enter the target weight: "))
specific_hen = find_specific_hen(hens, target_weight)
print("The specific hen is:", specific_hen)
```

Explaination of Code : -

1.Importing random Module: The random module is imported to shuffle the list of hens, ensuring random separation into two groups.

2.find_specific_hen Function: This function takes a list of hens (hens) and a target weight (target_weight) as input parameters. It first separates the hens into two groups (group1 and group2) using the separate_hens function. Then, it weighs these groups using the weigh function and recursively searches for the specific hen within the appropriate subgroup based on the comparison of weights with the target weight.

3.find_specific_hen_recursive Function: This is a helper function used by find_specific_hen. It recursively searches within a group of hens (group) to find the specific hen with the target weight.

4.separate_hens Function: This function separates the list of hens into two groups. It shuffles the hens using random.shuffle() to ensure randomness in separation and then divides the list into two roughly equal-sized groups.

5.weigh Function: This function calculates the total weight of a group of hens. For simplicity, it assumes that the weight of the group is the sum of weights of all hens in the group.

6.Example Usage: In this section, an example list of hens (hens) with their weights is provided. The user is prompted to enter a target weight (target_weight). The find_specific_hen function is called with these parameters, and the specific hen with the target weight is printed as the output.

Q2) Saurabh Kundan is searching for his wife – DFS Algorithm Will Use.

The scenario you've described resembles a classic problem in computer science known as the "Chain of Responsibility" design pattern. This pattern allows an object to send a request along a chain of potential handlers until one of them handles the request.

In your scenario:

Step 1 : Saurabh Kundan is searching for his wife, starting with the first kidnapper.

Step 2 : The first kidnapper directs Saurabh to the second kidnapper.

Step 3 : The second kidnapper directs Saurabh to the third kidnapper.

Step 4 : This process continues until Saurabh reaches the kidnapper who actually has his wife.

This scenario doesn't fit neatly into a specific algorithm but rather into a design pattern. However, if you want to frame it within the context of algorithms, you could consider it as a form of **depth-first search (DFS)** or recursive traversal, where Saurabh is traversing through a chain of kidnappers until he reaches the kidnapper who has his wife.

Here's a simplified representation in Python using a recursive approach:

```python
def search_for_wife(kidnapper_number, total_kidnappers):
    if kidnapper_number > total_kidnappers:
        print("Saurabh Kundan's wife found with kidnapper", total_kidnappers)
        return

    print("Meeting kidnapper", kidnapper_number)
    next_kidnapper = kidnapper_number + 1
    search_for_wife(next_kidnapper, total_kidnappers)

# Example usage:
total_kidnappers = 7  # Assuming there are 7 kidnappers in the chain
search_for_wife(1, total_kidnappers)  # Start search with the first kidnapper
```

This code recursively calls the search_for_wife() function, where kidnapper_number represents the current kidnapper being encountered, and total_kidnappers is the total number of kidnappers in the chain. The recursion continues until Saurabh reaches the kidnapper who has his wife.

Explaination Of Above Code :-

1.search_for_wife Function : This function takes two parameters: kidnapper_number, representing the current kidnapper being encountered, and total_kidnappers, representing the total number of kidnappers in the chain.

2.Base Case: If kidnapper_number exceeds total_kidnappers, it means Saurabh Kundan has searched through all the kidnappers without finding his wife. In this case, the function prints a message indicating that Saurabh Kundan's wife was found with the last kidnapper (total_kidnappers) and returns, terminating the recursion.

3.Recursive Case: If kidnapper_number is within the range of kidnappers, it means Saurabh Kundan still needs to search further. The function prints a message indicating that Saurabh Kundan is meeting the current kidnapper (kidnapper_number), then increments kidnapper_number to move to the next kidnapper, and recursively calls search_for_wife() with the updated kidnapper_number and total_kidnappers.

4.Example Usage: In this section, an example value for total_kidnappers is provided (7 in this case), assuming there are 7 kidnappers in the chain. The search_for_wife() function is called with the initial parameters 1 (indicating the first kidnapper) and total_kidnappers, starting the search process.

Q3) Find the path with minimum injuries to reach the Destination – Dynamic Programming Algorithm Will Use.

This problem can be approached using **dynamic programming**, particularly through a technique called "**Dynamic Programming with Memoization**" or "**Memoization with Dynamic Programming**." This technique is often used to efficiently solve problems involving overlapping subproblems by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Here's a high-level overview of how we can apply this technique to find the path with minimum injuries to reach the destination:

Step 1 : Define a recursive function that explores all possible paths from the starting point to the destination.

Step 2 : Use memoization to store the minimum injuries encountered for each location, so that we don't recalculate them unnecessarily.

Step 3 : Base cases: If the current location is the destination, return the number of injuries at that location.

Step 4 : Recursive case: For each possible move (up, down, left, right), recursively calculate the minimum injuries for the neighboring locations.

Step 5 : Choose the path with the minimum total injuries.

Here's a Python implementation of this approach:

```
def min_injuries(matrix, row, col, memo):
    if row == len(matrix) - 1 and col == len(matrix[0]) - 1:
        return matrix[row][col]  # Base case: reached destination

    if memo[row][col] != -1:
        return memo[row][col]  # Return memoized result if available

    injuries = matrix[row][col]

    down = float('inf')
    if row + 1 < len(matrix):
        down = min_injuries(matrix, row + 1, col, memo)

    right = float('inf')
    if col + 1 < len(matrix[0]):
        right = min_injuries(matrix, row, col + 1, memo)

    memo[row][col] = injuries + min(down, right)
    return memo[row][col]

# Example usage:
matrix = [
```

```
    [1, 3, 1, 5, 7],
    [1, 4, 1, 2, 3],
    [2, 5, 1, 2, 6],
    [1, 6, 1, 2, 1],
    [1, 1, 3, 1, 3]
]

memo = [[-1] * len(matrix[0]) for _ in range(len(matrix))]  # Memoization table
min_total_injuries = min_injuries(matrix, 0, 0, memo)
print("Minimum total injuries to reach (5, 5):", min_total_injuries)
```

Explaination of Above Code :-

1.min_injuries Function: This function takes four parameters: matrix, row, col, and memo.

2.matrix: Represents the 5x5 grid where each cell contains the number of injuries at that location.

3.row and col: Represent the current row and column indices being explored.

4.memo: Is a memoization table used to store the minimum injuries encountered for each location. It helps avoid redundant calculations by caching results.

5.Base Case: If the current position (row, col) is at the destination (5, 5), the function returns the number of injuries at that location. This is the base case for the recursion.

6.Recursive Case: If the minimum injuries for the current position (row, col) are not yet calculated (memo[row][col] == -1), the function calculates them recursively.

- It calculates the minimum injuries for moving down (row + 1, col) and moving right (row, col + 1). It takes the minimum of these two paths and adds the current position's injuries to it. This represents the minimum injuries encountered while moving to the destination.

- The calculated result is stored in the memoization table to avoid redundant calculations in the future.

7.Finally, the function returns the minimum injuries encountered to reach the destination (5, 5).

8.Example Usage: In this section, an example matrix is provided representing the 5x5 grid with the number of injuries at each location. A memoization table memo is initialized with -1 indicating that no minimum injuries are calculated yet. The min_injuries function is called with the initial parameters (matrix, 0, 0, memo) to start the search for the path with minimum injuries from (0, 0) (the starting position). The minimum total injuries encountered to reach (5, 5) is printed as the output.

Q4) Find the path with minimum injuries to reach the Destination – Dynamic Programming Algorithm Will Use.

To generate a list of 50 songs using the Fibonacci series and map them to buttons in a music player interface, we first need to generate the Fibonacci series up to the 50th number. We'll then create a list of songs where each song is represented by a Fibonacci number. Finally, we'll simulate a music player interface with Next and Previous buttons to navigate through the list of songs. Below is the algorithm and code to achieve this:

Algorithm:

Step 1 : Generate Fibonacci series up to the 50th number.

Step 2 : Create a list of 50 songs where each song corresponds to a Fibonacci number.

Step 3 : Implement a music player interface with Next and Previous buttons.

Step 4 : Use the Fibonacci sequence to determine which song to play next or previous.

Code:

```
def generate_fibonacci(n):
    fib_sequence = [0, 1]  # Initialize the sequence with the first two Fibonacci
numbers
    for i in range(2, n):
        fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])  # Generate
subsequent Fibonacci numbers
    return fib_sequence

def create_song_list(fib_sequence):
    song_list = ['Song {}'.format(num) for num in fib_sequence]
    return song_list

def music_player(song_list):
    current_song_index = 0
    while True:
        print("Current song:", song_list[current_song_index])
        user_input = input("Press 'n' for next song, 'p' for previous song, or
'q' to quit: ")
        if user_input == 'n':
            current_song_index = (current_song_index + 1) % len(song_list)
        elif user_input == 'p':
            current_song_index = (current_song_index - 1) % len(song_list)
        elif user_input == 'q':
            print("Exiting music player.")
            break
        else:
            print("Invalid input. Please try again.")

if __name__ == "__main__":
    fib_sequence = generate_fibonacci(50)
    song_list = create_song_list(fib_sequence)
    print("Generated song list:", song_list)
    music_player(song_list)
```

Explaination of Above Code :-

This code defines three functions: generate_fibonacci, create_song_list, and music_player, and then it executes the main logic to generate a song list based on the **Fibonacci sequence** and simulate a music player interface.

1.generate_fibonacci(n): This function generates the Fibonacci sequence up to the nth number. It initializes a list fib_sequence with the first two Fibonacci numbers (0 and 1) and then iterates from the third number up to n, appending each subsequent Fibonacci number to the list by adding the last two numbers in the sequence.

2.create_song_list(fib_sequence): This function takes the generated Fibonacci sequence and creates a list of songs where each song is represented by the corresponding Fibonacci number. It generates a list of strings with the format 'Song {}'.format(num) where num is each Fibonacci number in the sequence.

3.music_player(song_list): This function simulates a music player interface. It initializes a variable current_song_index to keep track of the index of the currently playing song. It runs an infinite loop where it continuously prints the current song from the song_list based on the current_song_index. It prompts the user to input 'n' for the next song, 'p' for the previous song, or 'q' to quit. It updates the current_song_index accordingly based on the user input.

The algorithm used in this code is **Fibonacci Sequence Generation**. It employs a simple iterative approach to generate Fibonacci numbers up to a given n, and then it maps these numbers to a list of songs to create a playlist. Finally, it simulates a music player interface to allow navigation through the playlist.