Assi 1 DAA

```
import time
import sys
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
n = int(input("Enter n: "))
start_time = time.time()  # Start timing
series = []
for i in range(n):
    series.append(fib(i))
    print("Step", i + 1, ":", series)
end_time = time.time()  # End timing
execution_time = end_time - start_time
memory_usage = sys.getsizeof(series) + sum(sys.getsizeof(num) for num in series)
print("Fibonacci Series:", series)
print(f"Execution Time (Time Complexity): {execution_time:.6f} seconds")
print(f"Memory Usage (Space Complexity): {memory_usage} bytes")
pos = int(input("Enter position: "))
if 0 <= pos < len(series):
    print("Value at position", pos, "is:", series[pos])
else:
    print("Position out of range")
```

Assi 2 DAA

```
class HuffmanTreeNode:
    def __init__(self, character, frequency):
        self.char = character
        self.freq = frequency
        self.left = None
        self.right = None
def build_huffman_tree(data, freq):
    nodes = [HuffmanTreeNode(data[i], freq[i]) for i in range(len(data))]
        while len(nodes) > 1:
        nodes.sort(key=lambda node: node.freq)
        left = nodes.pop(0)
        right = nodes.pop(0)
        merged = HuffmanTreeNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        nodes.append(merged)
    return nodes[0]  # The remaining node is the root of the Huffman tree
def get_huffman_codes(node, current_code="", codes={}):
    if node is None:
        return
    if node.char is not None:
        codes[node.char] = current_code
    get_huffman_codes(node.left, current_code + "0", codes)
    get_huffman_codes(node.right, current_code + "1", codes)
    return codes
def huffman_encoding(data, freq):
    root = build_huffman_tree(data, freq)
    return get_huffman_codes(root)
def encode_data(data, huffman_codes):
    return ''.join(huffman_codes[char] for char in data)
if __name__ == '__main__':
    size = int(input("Enter the number of characters: "))
    data = []
    freq = []
```

```
for i in range(size):
    character = input(f"Enter character {i + 1}: ")
    frequency = int(input(f"Enter frequency for '{character}': "))
    data.append(character)
    freq.append(frequency)
codes = huffman_encoding(data, freq)
original_data = ''.join(data)  # Assuming you want to encode the original data string
encoded_data = encode_data(original_data, codes)
print("\nEncoded data:", encoded_data)
print("\nHuffman Codes:")
for char, code in codes.items():
    print(f"{char}: {code}")
```

Assi 3 DAA

```
class Item:
    def __init__(self,value,weight):
        self.value=value
        self.weight=weight
        self.density=value/weight
def knapsack(capacity,items):
    items.sort(key=lambda x:x.density,reverse=True)
    total_value=0.0
    for item in items:
        if capacity>=item.weight:
            capacity-=item.weight
            total_value+=item.value
        else:
            total_value+=item.density*capacity
            break
    return total_value
n=int(input("enter a number of items"))
items=[]
for i in range(n):
    value = float(input(f"Enter the value of item {i+1}: "))
    weight = float(input(f"Enter the weight of item {i+1}: "))
    item = Item(value, weight)
    items.append(item)
    print(f"Item {i+1}: Value = {item.value}, Weight = {item.weight}, Density = {item.density:.2f}")
capacity = float(input("Enter the capacity of the knapsack: "))
max_value = knapsack(capacity, items)
print(f"The maximum value that can be carried in the knapsack is: {max_value}")
```

Assi 4 DAA

```
def knapsack_01(weights, values, capacity):
    n = len(values)
    k = capacity
    dp = [[0 for _ in range(k + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, k + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    print("DP Matrix:")
    for row in dp:
        print(row)
    return dp[n][capacity]
if __name__ == "__main__":
    n = int(input("Enter the number of items: "))
    weights = []
    values = []
    for i in range(n):
        weight = int(input(f"Enter weight of item {i + 1}: "))
        value = int(input(f"Enter value of item {i + 1}: "))
        weights.append(weight)
        values.append(value)
    capacity = int(input("Enter the capacity of the knapsack: "))
    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value in Knapsack = {max_value}")


# Time Complexity: O(N * K), where N is the number of items and K is the knapsack capacity
# Space Complexity: O(N * K)# direct input #  weights = [3, 4, 6, 5]  # Weights of items
#     values = [2, 3, 1, 4]   # Values of items
#     capacity = 8            # Capacity of knapsack
#     max_value = knapsack_01(weights, values, capacity)
#     print(f"Maximum value in Knapsack = {max_value}")
```

Assi 5 DAA

```
def solve_queen(board=None, row=0, solutions=None, n=None):
    if n is None:
        n = int(input("Enter the size of the board (N): "))  # Get the board size from the user
    if board is None:
        board = [[0] * n for _ in range(n)]
    if solutions is None:
        solutions = []
    if row >= n:
        solutions.append([r[:] for r in board])
        return solutions
    for col in range(n):
        if all(board[i][col] == 0 for i in range(row)) and \
           all(board[i][j] == 0 for i, j in zip(range(row - 1, -1, -1), range(col - 1, -1, -1))) and \
           all(board[i][j] == 0 for i, j in zip(range(row - 1, -1, -1), range(col + 1, n))):
            board[row][col] = 1  # Place the queen
            solve_queen(board, row + 1, solutions, n)  # Recur to place the next queen
            board[row][col] = 0  # Backtrack and remove the queen
    return solutions
solutions = solve_queen()
print(f"Total number of solutions found: {len(solutions)}")
for idx, solution in enumerate(solutions, start=1):
    print(f"Solution #{idx}:")
    for row in solution:
        print(row)
    print()
```