WEEK 5 TASK

TASK 33: Find All Permutations of a String

Steps:

1. Input a string from the user.

2. Generate all permutations recursively.

3. Display all possible arrangements of the string characters.

Expected Input:

Enter a string: abc

Expected Output:

Permutations: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

Code:

```
def permutations(string, step=0):
    if step == len(string):
        print("".join(string))
        return
    for i in range(step, len(string)):
        string_copy = [c for c in string]
        string_copy[step], string_copy[i] = string_copy[i], string_copy[step]
        permutations(string_copy, step + 1)

input_string = input("Enter a string: ")
permutations(list(input_string))
```

TASK 34: N-th Fibonacci Number (Dynamic Programming)

Steps:

1. Input an integer n from the user.

2. Use a bottom-up dynamic programming approach.

3. Store previous values to optimize performance.

Expected Input:

Enter n: 10

Expected Output:

Fibonacci number: 55

Code:

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    dp = [0, 1]
    for i in range(2, n+1):
        dp.append(dp[i-1] + dp[i-2])
    return dp[n]

n = int(input("Enter n: "))
print("Fibonacci number:", fibonacci(n))
```

TASK 35: Find Duplicates in a List

Steps:

1. Input a list of integers.

2. Count occurrences of each number.

3. Identify numbers that appear more than once.

Expected Input:

Enter numbers separated by space: 1 2 3 4 2 3 5 6

Expected Output:

Duplicates: [2, 3]

Code:

```
def find_duplicates():
    arr = list(map(int, input("Enter numbers separated by space: ").split()))
    count = {}
    duplicates = []
    for num in arr:
        count[num] = count.get(num, 0) + 1
    for num, freq in count.items():
        if freq > 1:
            duplicates.append(num)
    print("Duplicates:", duplicates)

find_duplicates()
```

TASK 36: Longest Increasing Subsequence (LIS)

Steps:

1. Input a list of integers.

2. Use dynamic programming to store increasing subsequences.

3. Return the length of the longest increasing subsequence.

Expected Input:

Enter numbers separated by space: 10 22 9 33 21 50 41 60 80

Expected Output:

Length of LIS: 6

Code:

```
def longest_increasing_subsequence():
    arr = list(map(int, input("Enter numbers separated by space: ").split()))
    n = len(arr)
    lis = [1] * n
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j]:
                lis[i] = max(lis[i], lis[j] + 1)
    print("Length of LIS:", max(lis))

longest_increasing_subsequence()
```

TASK 37: Find K Largest Elements

Steps:

1. Input a list of integers and an integer k.

2. Sort the list in descending order.

3. Extract the top k elements.

Expected Input:

Enter numbers separated by space: 10 2 8 1 6 4

Enter k: 3

Expected Output:

K largest elements: [10, 8, 6]

Code:

```
def k_largest_elements():
    arr = list(map(int, input("Enter numbers separated by space: ").split()))
    k = int(input("Enter k: "))
    arr.sort(reverse=True)
    print("K largest elements:", arr[:k])

k_largest_elements()
```

TASK 38: Rotate Matrix

Steps:

1. Input a 2D matrix from the user.

2. Transpose the matrix.

3. Reverse each row to achieve 90-degree rotation.

Expected Input:

Matrix:

[[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

Expected Output:

Rotated Matrix:

[[7, 4, 1],

[8, 5, 2],

[9, 6, 3]]

Code:

```
def rotate_matrix():
    matrix = []
    n = int(input("Enter matrix size (n x n): "))
    for i in range(n):
        matrix.append(list(map(int, input().split())))
    rotated = list(zip(*matrix[::-1]))
    print("Rotated Matrix:")
    for row in rotated:
        print(list(row))

rotate_matrix()
```

TASK 39: Sudoku Validator

Steps:

1. Input a 9x9 matrix representing the Sudoku board.

2. Validate rows, columns, and 3x3 grids for duplicates.

3. Return True if valid, False otherwise.

Expected Input:

A valid Sudoku board.

Expected Output:

True

Code:

```
def is_valid_sudoku(board):
    def is_valid_unit(unit):
        unit = [num for num in unit if num != 0]
        return len(unit) == len(set(unit))

    for row in board:
        if not is_valid_unit(row):
            return False

    for col in zip(*board):
        if not is_valid_unit(col):
            return False

    for i in range(0, 9, 3):
        for j in range(0, 9, 3):
            if not is_valid_unit([board[x][y] for x in range(i, i+3) for y in range(j, j+3)]):
                return False
    return True

sudoku_board = [list(map(int, input().split())) for _ in range(9)]
```

```
print(is_valid_sudoku(sudoku_board))
```

TASK 40: Virtual Stock Market Simulator

(Stock market simulation code remains unchanged.)

TASK 40: Virtual Stock Market Simulator

Steps:

1. Simulate stock price fluctuations using random values.
2. Allow users to buy and sell stocks.
3. Track portfolio value based on transactions.

Expected Input:

Enter initial balance: 1000

Buy/Sell? (b/s): b

Enter stock name: XYZ

Enter amount: 2

Expected Output:

Stock XYZ bought at $50 per share. Portfolio updated.

Code:

```
import random

def simulate_stock_market():
    balance = float(input("Enter initial balance: "))
    portfolio = {}
```

```
prices = {'XYZ': random.randint(10, 100), 'ABC': random.randint(20, 150)}


while True:
    action = input("Buy/Sell? (b/s) or 'q' to quit: ")
    if action == 'q':
        break
    stock = input("Enter stock name: ")
    if stock not in prices:
        print("Invalid stock name.")
        continue
    amount = int(input("Enter amount: "))


    if action == 'b':
        cost = prices[stock] * amount
        if cost > balance:
            print("Insufficient funds.")
            continue
        balance -= cost
        portfolio[stock] = portfolio.get(stock, 0) + amount
        print(f"Stock {stock} bought at ${prices[stock]} per share. Portfolio updated.")
    elif action == 's':
        if stock not in portfolio or portfolio[stock] < amount:
            print("Not enough shares to sell.")
            continue
        balance += prices[stock] * amount
        portfolio[stock] -= amount
        print(f"Stock {stock} sold at ${prices[stock]} per share. Portfolio updated.")
```

```
        print(f"Current balance: ${balance}")
        print(f"Portfolio: {portfolio}")

simulate_stock_market()
```