AI ASSISTED CODING

ASSIGNMENT-1

Name: A.Abhilash Goud

Hall ticket: 2303a51359

Batch: 20

Task-1: Fibonacci Sequence Without Functions

```
#fibinacci series up to n terms
n=10
a,b=0,1
for i in range(n):
    print(a)
    a,b=b,a+b
```

Output:

```
0
1
1
2
3
5
8
13
21
34
```

Explanation:

In this approach, the Fibonacci sequence is generated using a single block of code without defining any function. All the logic such as initialization, calculation, and output is written inside the main program. While this method is easy to understand for beginners, the code becomes lengthy and less organized. If the same Fibonacci logic is needed again, the code must be rewritten, which reduces reusability. Debugging is also difficult because the entire logic is tightly coupled in one place.

Task-2: Improving Efficiency

```
# Generate Fibonacci sequence without using functions
fibonacci_sequence = []
a, b = 0, 1
for i in range(n):
    fibonacci_sequence.append(a)
    a, b = b, a + b

print("Fibonacci sequence:", fibonacci_sequence)
```

Output:

```
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\temba\OneDrive\Desktop\AI Asst Coding>
```

Explanation:

The efficiency of the Fibonacci program is improved by reducing unnecessary operations and using optimized logic. Instead of recalculating values repeatedly, the program stores previously computed Fibonacci numbers and uses them for further calculations. This reduces time complexity and improves performance, especially for larger inputs. Efficient logic makes the program faster and more suitable for real-world applications.

Task-3: Fibonacci Using Functions

```
# More efficient: generate Fibonacci up to n terms using a generator
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b
n = 10
# Use the generator
fibonacci_sequence = list(fibonacci(n))
print("Fibonacci sequence:", fibonacci_sequence)
```

Output:

```
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\temba\OneDrive\Desktop\AI Asst Coding>
```

Explanation:

In this approach, the Fibonacci logic is placed inside a function, which makes the program modular. The main program simply calls the function whenever required. This improves code readability and structure. Functions allow reusability, meaning the same Fibonacci logic can be used multiple times without rewriting code. Debugging becomes easier because errors can be identified and fixed inside the function without affecting the entire program.

Task-4: Comparative Analysis – Procedural vs Modular Fibonacci Code

The Fibonacci program without functions follows a purely procedural approach where all the logic is written in one continuous block, which makes the code simple but less clear and difficult to manage as the program grows. It has poor reusability because the Fibonacci logic cannot be reused without rewriting the code, and debugging becomes harder since all operations are tightly coupled. On the other hand, the Fibonacci program with functions follows a modular approach where the logic is separated into a function, improving code clarity and readability. This approach allows easy reuse of the Fibonacci logic, simplifies debugging by isolating errors within the function, and makes the program more suitable for larger systems that require scalability and maintainability.

Task 5: Different Algorithmic Approaches for Fibonacci Series

```python
# Fibonacci using memoization (optimized recursive approach)
def fib_memoization(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memoization(n - 1, memo) + fib_memoization(n - 2, memo)
    return memo[n]
n=10
fibonacci_sequence = [fib_memoization(i) for i in range(n)]
print("Fibonacci sequence (memoization):", fibonacci_sequence)

# Fibonacci using dynamic programming (bottom-up approach)
def fib_dp(n):
    if n <= 1:
        return [i for i in range(n + 1)]
    dp = [0] * n
    dp[0], dp[1] = 0, 1
    for i in range(2, n):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp
n=10
fibonacci_sequence = fib_dp(n)
print("Fibonacci sequence (dynamic programming):", fibonacci_sequence)

# Fibonacci using matrix exponentiation (most efficient for large n)
def fib_matrix(n):
    def matrix_multiply(a, b):
        return [[a[0][0]*b[0][0] + a[0][1]*b[1][0], a[0][0]*b[0][1] + a[0][1]*b[1][1]],
                [a[1][0]*b[0][0] + a[1][1]*b[1][0], a[1][0]*b[0][1] + a[1][1]*b[1][1]]]

    def matrix_power(matrix, power):
        if power == 1:
            return matrix
        if power % 2 == 0:
            half = matrix_power(matrix, power // 2)
            return matrix_multiply(half, half)
        return matrix_multiply(matrix, matrix_power(matrix, power - 1))

    if n == 0:
        return 0
    result = matrix_power([[1, 1], [1, 0]], n)
    return result[0][1]
n=10
fibonacci_sequence = [fib_matrix(i) for i in range(n)]
print("Fibonacci sequence (matrix exponentiation):", fibonacci_sequence)
```

Output:

```
PS C:\Users\temba\OneDrive\Desktop\AI Asst Coding> & C:/Users/temba/AppData/Local/Programs/Python/Python314/python.exe
a/OneDrive/Desktop/AI Asst Coding/Assig-1.py"
Fibonacci sequence (memoization): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Fibonacci sequence (dynamic programming): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Fibonacci sequence (matrix exponentiation): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\temba\OneDrive\Desktop\AI Asst Coding> & C:/Users/temba/AppData/Local/Programs/Python/Python314/python.exe
a/OneDrive/Desktop/AI Asst Coding/Assig-1.py"
Fibonacci sequence (memoization): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Fibonacci sequence (dynamic programming): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Fibonacci sequence (matrix exponentiation): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Explanation:

Different algorithmic approaches such as iterative, recursive, and dynamic programming methods can be used to generate the Fibonacci series. Iterative methods are efficient and use less memory. Recursive methods are easy to understand but inefficient due to repeated calculations. Dynamic programming provides the best performance by storing previously computed values. Choosing the right approach depends on the problem size and performance requirements.