Assignment-1

2303A52481

Sindhu Katakam

Batch-41

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

Prompt: write a python program to print Fibonacci series upto n terms

Code:

```
#write a python program to print fibonacci series up to n terms.
n=int(input("Enter the number of terms in Fibonacci series: "))
a,b=0,1
for i in range(n):
    print(a,end=' ')
    a,b=b,a+b
```

Output:

Justification

By doing this task, printed the Fibonacci series up to *n* terms without using any functions. All the steps are written directly in the main program. The program takes a number from the user and then uses one `for` loop to show the Fibonacci sequence.

Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

Prompt: optimize the code by removing unnecessary variables and improving readability

Code:

```python
#optimize the code below by removing unnecessary variables and improving readability
n= int(input("Enter the number of terms in Fibonacci series: "))
a,b=0,1
for _ in range(n):
    print(a, end=' ')
    a, b = b, a + b
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/I
IT_CODING/AI-assit coding day-1.py"
Enter the number of terms in Fibonacci series: 10
0 1 1 2 3 5 8 13 21 34
PS D:\AI_ASSIT_CODING>
```

Justification

By doing this task, improved the previous code to make it work better. The code from Task 1 was already simple and good, but optimizing the code is important. Better optimization makes the program run faster and use less memory. If a program takes too long to run, it has high time complexity. If it uses too much memory, it has high space complexity. So, optimizing the code helps us finish tasks faster and more efficiently.

Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Prompt: #optimize the code by removing unnecessary variables and improving readability and avoiding unnecessary computations using functions

Code:

```python
#optimize the code below by removing unnecessary variables and improving readability
def fibonacci_series(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b

n = int(input("Enter the number of terms in Fibonacci series: "))
fibonacci_series(n)
```

Output:

JUSTIFICATION:

In this task, created the Fibonacci series using our own function. The function contains all the steps to find the Fibonacci numbers, which makes the code organized and reusable. Using functions helps keep the code neat, easy to read, easy to fix, and useful for bigger programs

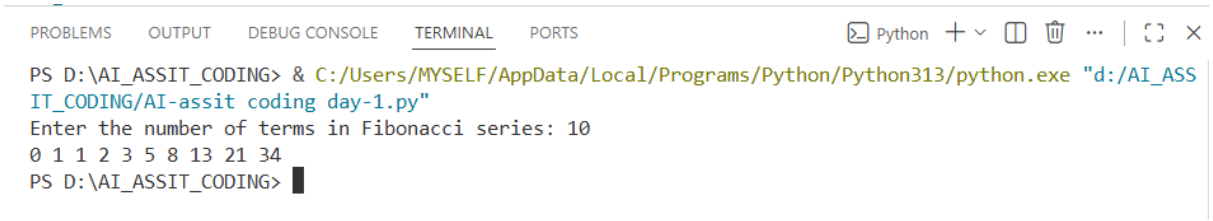
Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Prompt: write a python program to print Fibonacci series up to n terms without functions

Code:

```
#write a python program to print fibonacci series up to n terms without functions.
n=int(input("Enter the number of terms in Fibonacci series: "))
a,b=0,1
for i in range(n):
    print(a,end=' ')
    a,b=b,a+b
```

Output:



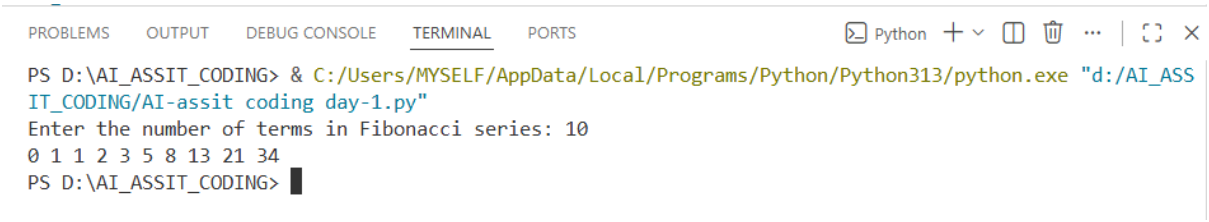
Prompt: write a python code for modular Fibonacci series

Code:

```python
#write a python code for modular fibonacci series.
def modular_fibonacci_series(n, mod):
    a, b = 0, 1
    for _ in range(n):
        print(a % mod, end=' ')
        a, b = b, a + b

n = int(input("Enter the number of terms in Fibonacci series: "))
mod = int(input("Enter the modulus value: "))
modular_fibonacci_series(n, mod)
```

Output:



JUSTIFICATION:
In this task, learned the difference between Procedural programming (without using functions) and Modular programming (using functions). Using functions is helpful because:
• We can use the same code again
• It is easier to find and fix errors
• The code looks cleaner and easier to understand
• It works well for bigger programs

From this, we can see that the modular method is a better and neater way to write programs.

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

Prompt: give code for iterative Fibonacci implementation

Code:

```python
#give code for iterative fibonacci implementation
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
n = int(input("Enter a number: "))
print(f"The {n}th Fibonacci number is: {fibonacci(n)}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Loc
IT_CODING/AI-assit coding day-1.py"
Enter a number: 10
The 10th Fibonacci number is: 55
PS D:\AI_ASSIT_CODING>
```

Prompt: give code for recursive Fibonacci implementation

Code:

```python
#give code for recursive fibonacci implementation
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
n = int(input("Enter a number: "))
print(f"The {n}th Fibonacci number is: {fibonacci(n)}")
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/AppData/Local/Programs/Python/Python313/python.exe "d:/AI_ASS
IT_CODING/AI-assit coding day-1.py"
Enter a number: 12
The 12th Fibonacci number is: 144
```

JUSTIFICATION

Using a loop to find Fibonacci numbers is usually better because it works faster, uses less memory, and can handle very large numbers more easily. The recursive way (where a function calls itself) is slower, uses more memory, and can even crash if the number is too big. That's why real programs usually use loops instead of recursion.