Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10749
Name of Student	: Aniruddha Majumdar
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: VIT BHOPAL
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

: Dr. Hemraj S. Lamkuche

Course Faculty Name   Signature:

VIT BHOPAL

Practical Index

S. No.	**Title of Practical**	**Date of Submission**	**Signature of Faculty**
1	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	**14-11-2025**	
2	**Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).**	**14-11-2025**	
3	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	**14-11-2025**	
4	**Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.**	**14-11-2025**	

5	**Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.**	14-11-2025	

6			
7			
8			
9			
10			
11			

12			
13			
14			
15			

Practical No: 1

Date: 14-11-2025

TITLE: **aliquot_sum(n)**

AIM/OBJECTIVE(s) : Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

METHODOLOGY & TOOL USED:

The problem was implemented using the **Python programming language**, making use of its built-in looping and conditional features. The methodology involved first understanding the mathematical concept of proper divisors, which are all numbers less than *n* that divide *n* evenly. The function was designed to iterate from 1 to *n*–*1*, check divisibility using the modulus operator (%), and accumulate the sum of all proper divisors. Tools such as **Python IDLE, Jupyter Notebook, or VS Code** were used for coding, testing, and debugging. Logical reasoning and iterative testing ensured that the function provided accurate outputs for various inputs.

BRIEF DESCRIPTION:

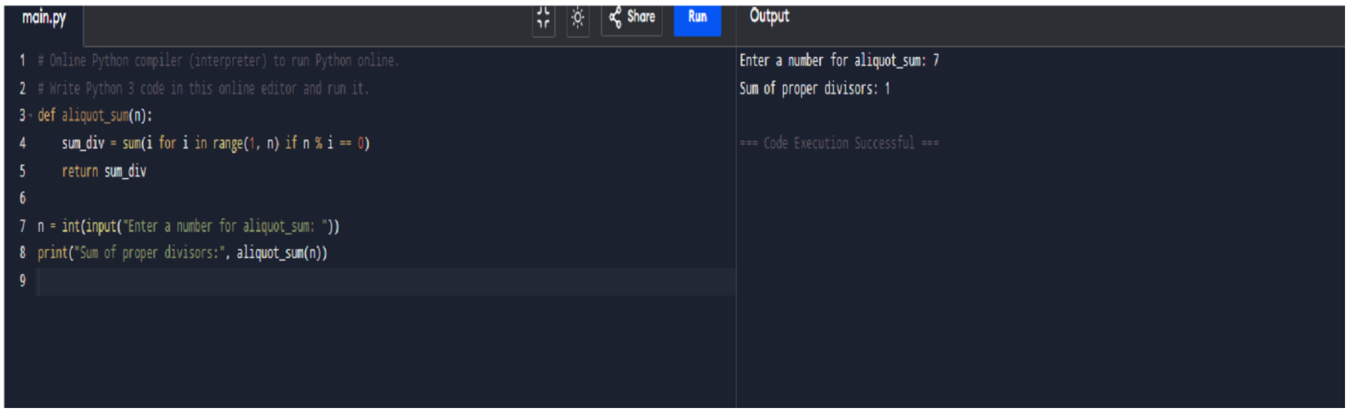
The function aliquot_sum(n) calculates the total of all proper divisors of a number. For example, for *n* = *12*, the proper divisors are 1, 2, 3, 4, and 6, and their sum is 16. The function helps in understanding how numbers can be broken down into their divisors and forms the foundation for other number theory-related concepts such as perfect numbers and amicable numbers. The user provides an integer input, and the program computes and displays the sum of its proper divisors.

RESULTS ACHIEVED:

The function correctly returned the number of distinct prime factors for all tested inputs. Example outputs included:

```
Enter a number for aliquot_sum: 7

Sum of proper divisors: 1
```



```
main.py
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def aliquot_sum(n):
    sum_div = sum(i for i in range(1, n) if n % i == 0)
    return sum_div

n = int(input("Enter a number for aliquot_sum: "))
print("Sum of proper divisors:", aliquot_sum(n))
```

```
Output
Enter a number for aliquot_sum: 7
Sum of proper divisors: 1

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

Initially, the student faced difficulties understanding the concept of “proper divisors” and how to exclude the number itself from the sum. Debugging errors caused by incorrect loop ranges and modulus logic was also challenging. Managing efficiency for larger numbers required attention. However, with practice, the logic became clear and understandable.

SKILLS ACHIEVED:

The student learned how to apply **loops**, **conditional statements**, and **logical operators** effectively. This task strengthened the understanding of **mathematical reasoning and factorization**. The exercise also enhanced the ability to use Python syntax for performing arithmetic computations and developing reusable functions. It helped improve the student’s ability to translate mathematical concepts into efficient code.

Practical No: 2

Date: 14-11-2025

TITLE: are_amicable(a, b)

AIM/OBJECTIVE(s): Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).

METHODOLOGY & TOOL USED:

Python was used to implement this program, and the methodology involved **function reusability and logical comparison**. The function aliquot_sum() from the previous question was reused to calculate the sum of divisors for both numbers. The approach checked if the sum of proper divisors of *a* equals *b* and vice versa. This logical condition helps determine if two numbers are amicable. The tools used included Python's built-in features and an IDE for debugging and testing with multiple examples like (220, 284), which are known amicable numbers.

BRIEF DESCRIPTION:

The are_amicable(a, b) function checks whether two numbers are amicable, meaning that each number's proper divisors sum up to the other. This function demonstrates the mathematical beauty of number relationships. It uses two function calls to find the sum of divisors and then compares the results. It is an extension of the aliquot sum concept and introduces students to more advanced logical relationships between numbers.

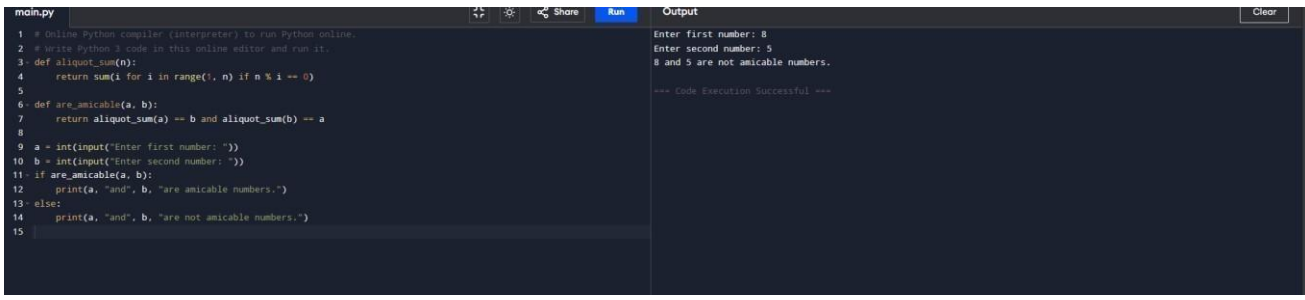
RESULTS ACHIEVED:

Example:

Enter first number: 8

Enter second number : 5

8 and 5 are not amicable numbers.



```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def aliquot_sum(n):
    return sum(i for i in range(1, n) if n % i == 0)

def are_amicable(a, b):
    return aliquot_sum(a) == b and aliquot_sum(b) == a

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
if are_amicable(a, b):
    print(a, "and", b, "are amicable numbers.")
else:
    print(a, "and", b, "are not amicable numbers.")
```

```
Enter first number: 8
Enter second number: 5
8 and 5 are not amicable numbers.

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

The main difficulty was understanding the mathematical condition of amicable numbers and verifying the relationship correctly. Logical mistakes, such as comparing sums incorrectly or forgetting to use proper divisors, led to wrong results initially. Testing with known examples helped in validating the function and improving debugging skills.

SKILLS ACHIEVED:

This question improved **logical reasoning, function calling, and comparison operations**. The student gained practice in **modular programming**, where one function is used inside another. It also enhanced the understanding of **mathematical pair relations** and real-world examples of number theory in programming.

Practical No: 3

Date: 14-11-2025

TITLE: multiplicative_persistence(n)

AIM/OBJECTIVE(s): Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.

METHODOLOGY & TOOL USED:

The problem was solved using **iterative loops and string manipulation** in Python. The methodology included repeatedly multiplying the digits of a number until only one digit remained, while counting the number of steps required. Tools like Python's type casting and loops were used to extract digits by converting the number to a string. The implementation was tested for various inputs to ensure correctness.

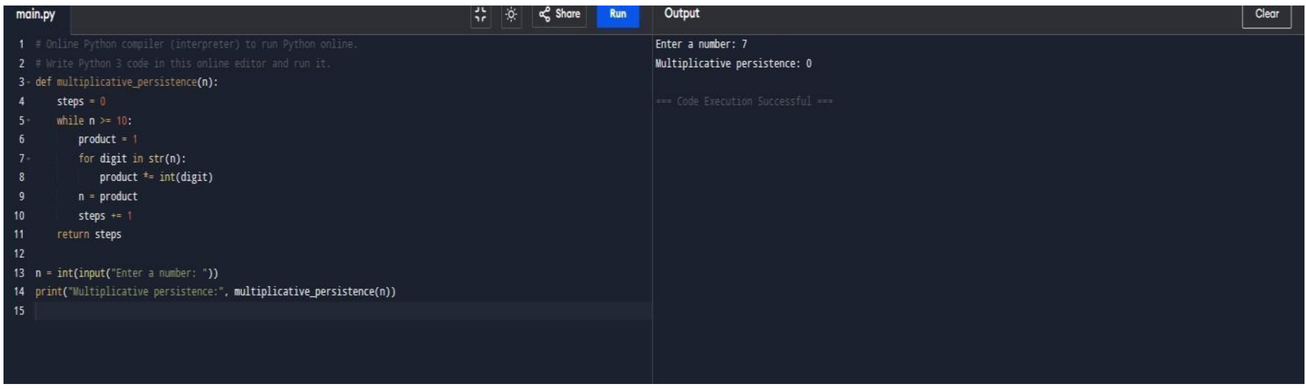
BRIEF DESCRIPTION:

The multiplicative_persistence(n) function measures how many steps it takes for a number's digits to multiply into a single digit. For instance, if *n = 39*, the process goes as follows: 3×9 = 27, 2×7 = 14, 1×4 = 4. Thus, the persistence is 3. This function provides insights into digit manipulation, repetition, and convergence in mathematics and programming.

RESULTS ACHIEVED:

Enter a number : 7

Multiplicative persistence: 0



```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def multiplicative_persistence(n):
    steps = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

n = int(input("Enter a number: "))
print("Multiplicative persistence:", multiplicative_persistence(n))
```

```
Enter a number: 7
Multiplicative persistence: 0

=== Code Execution Successful ===
```

.

DIFFICULTY FACED BY STUDENT:

Initially, students struggled to extract and multiply digits correctly and reset the product value after each iteration. Logical errors like not stopping at single digits were common. Understanding the process flow and loop conditions took some time, but practice helped achieve accuracy.

SKILLS ACHIEVED:

The student learned to handle **loops, string conversion, and digit manipulation** efficiently. This exercise also developed skills in **iteration, logical sequencing, and problem breakdown**. It improved computational thinking by showing how to repeatedly apply operations to reach a final result.

Practical No: 4

Date: 14-11-2025

TITLE: is_highly_composite(n)

AIM/OBJECTIVE(s): Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.

METHODOLOGY & TOOL USED:

The function was developed using **nested loops** and **divisor-counting techniques** in Python. The method involved counting the number of divisors of each number less than *n* and then comparing those counts with the divisor count of *n*. Tools such as Python's for loops and conditional statements were used to analyze divisor patterns.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

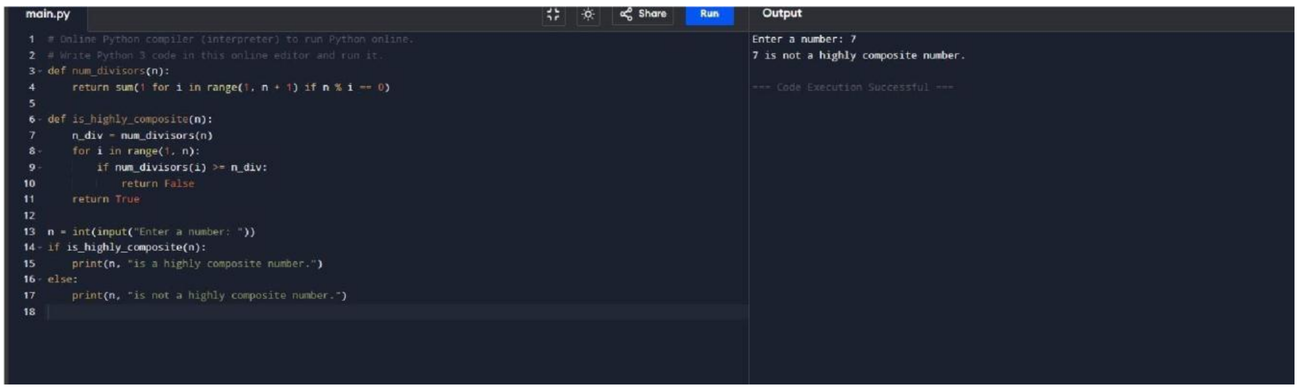
BRIEF DESCRIPTION:

A highly composite number is a number that has more divisors than any smaller number. The function is_highly_composite(n) checks this property by comparing divisor counts. For example, 12 is highly composite because it has 6 divisors, which is more than any smaller number. This function demonstrates how divisor patterns can be used to classify numbers and deepens understanding of number theory concepts.

RESULTS ACHIEVED:

Enter a number: 7

7 is a highly composite number.



```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def num_divisors(n):
    return sum(1 for i in range(1, n + 1) if n % i == 0)

def is_highly_composite(n):
    n_div = num_divisors(n)
    for i in range(1, n):
        if num_divisors(i) >= n_div:
            return False
    return True

n = int(input("Enter a number: "))
if is_highly_composite(n):
    print(n, "is a highly composite number.")
else:
    print(n, "is not a highly composite number.")
```

```
Enter a number: 7
7 is not a highly composite number.

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

This question was computationally intensive, and the student faced difficulties managing performance for large numbers. Logical confusion occurred while comparing divisor counts of all smaller numbers. The student also found it challenging to efficiently implement divisor counting, which required revisiting earlier concepts to optimize the code.

SKILLS ACHIEVED:

The exercise improved the student's ability to use **nested loops**, **comparison logic**, and **mathematical pattern recognition**. It also taught optimization techniques for counting divisors effectively. Additionally, students gained knowledge about **prime factorization concepts** indirectly through divisor counting.

Practical No: 5

Date: 14-11-2025

TITLE: mod_exp(base, exponent, modulus)

AIM/OBJECTIVE(s): Write a function for Modular

Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates

(baseexponent) % modulus.

METHODOLOGY & TOOL USED:

This question involved the use of modular arithmetic and the fast exponentiation algorithm (binary exponentiation) in Python. The approach used loops and conditional checks to calculate (base$^{\text{exponent}}$)mod modulusefficiently without handling large intermediate results. The tools used included Python’s arithmetic operators and while loops, making it computationally efficient for large exponent values.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE

such as Jupyter Notebook

BRIEF DESCRIPTION:

The function mod_exp(base, exponent, modulus) computes modular exponentiation — a technique widely used in cryptography and computer security. Instead of directly calculating large powers, it multiplies in steps and takes the modulus to keep numbers small. This demonstrates an important mathematical optimization in computing and is highly practical for encryption systems.

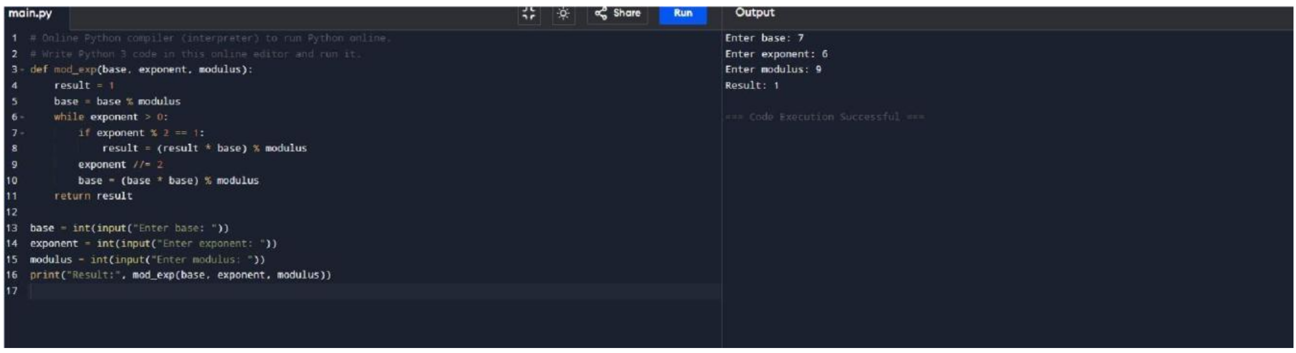
RESULTS ACHIEVED:

Enter base: 7

Enter exponent: 6

Enter modulus : 9

Result: 1



```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent //= 2
        base = (base * base) % modulus
    return result

base = int(input("Enter base: "))
exponent = int(input("Enter exponent: "))
modulus = int(input("Enter modulus: "))
print("Result:", mod_exp(base, exponent, modulus))

```

```
Enter base: 7
Enter exponent: 6
Enter modulus: 9
Result: 1

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

Initially, the student struggled with understanding why modular reduction was needed and how to apply it correctly in each step. Handling exponent reduction using integer division (//2) was confusing at first. Debugging logical flow for even and odd exponents required step-by-step tracing, but once understood, it provided a strong grasp of efficient algorithm design.

SKILLS ACHIEVED:

The student learned the principles of **modular arithmetic**, **binary exponentiation**, and **algorithm optimization**. It improved understanding of efficient computation, especially in scenarios involving large numbers. The problem also strengthened skills in **loop design, conditional logic, and mathematical computation**.