VIT BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No :25MIP10141

Name of Student :SATVIK SRIVASTAVA

Course Name : Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name : School of Computing Science Engineering and Artificial Intelligence (SCAI)

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	aliquot_sum(n) Function	10/11/20225	
2	are_amicable(a, b) Function	10/11/20225	
3	multiplicative_persistence(n) function	10/11/20225	
4	is_highly_composite(n) Function	10/11/20225	
5	mod_exp(base, exponent,modulus) function	10/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE:aliquot_sum(n) Function

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:

Python programming language

BRIEF DESCRIPTION:

This Python program calculates the **sum of proper divisors** of a given number. It takes an integer input `n`, iterates through all numbers from 1 to `n-1`, and adds those that evenly divide `n`. Finally, it prints the total sum of these divisors as the result.

RESULTS ACHIEVED:

```
n = int(input("Enter a number: "))
sum_divisors = 0
for i in range(1, n):
    if n % i == 0:
        sum_divisors += i
print("Sum of proper divisors:",sum_divisors)
```

```
Enter a number:  12
Sum of proper divisors: 16
```

SKILLS ACHIEVED:

Iteration and conditional accumulation.

Understanding of divisors and proper divisors.

Practical No: 2

Date: 10/11/2025

TITLE: are_amicable(a, b) Function

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

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

This Python program checks if two numbers are **amicable**. It defines a function `aliquot_sum(n)` that calculates the sum of proper divisors of `n`. Then, it compares whether each number equals the other's divisor sum. If true, it prints "Amicable numbers"; otherwise, it prints "Not amicable."

Result:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

sum_a = 0
for i in range(1, a):
    if a % i == 0:
        sum_a += i

sum_b = 0
for i in range(1, b):
    if b % i == 0:
        sum_b += i

if sum_a == b and sum_b == a:
    print("Amicable numbers")
else:
    print("Not amicable numbers")
```

```
Enter first number:  13
Enter second number:  15
Not amicable numbers
```

SKILLS ACHIEVED:

Function Reusability: Leveraging existing functions to build new solutions

Logical Condition Construction: Combining multiple conditions with AND logic

Practical No: 3

Date: 11/11/2025

TITLE:multiplicative_persistence(n) function

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

Python programming language

BRIEF DESCRIPTION:
This Python program finds the **multiplicative persistence** of a number — the number of steps required to reduce it to a single digit by multiplying its digits repeatedly. It loops while the number has more than one digit, multiplies all digits together, updates the number, and counts the steps

RESULTS ACHIEVED:

```
n = int(input("Enter a number: "))
count = 0

while n >= 10:
    product = 1
    for digit in str(n):
        product *= int(digit)
    n = product
    count += 1

print("Multiplicative persistence:",count)
```

```
Enter a number:  15
Multiplicative persistence: 1
```

SKILLS ACHIEVED:

Type Conversion: converting between integers and strings for digit manipulation

Iterative Process Control: Managing loops until convergence criteria are met

Practical No: 4

Date: 10/11/2025

TITLE:is_highly_composite(n) Function

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:

Python programming language

BRIEF DESCRIPTION:

This Python program checks whether a number is highly composite. It defines a function `count_divisors(x)` to count total divisors of any number. Then, it compares the divisor count of the input number with all smaller numbers. If any smaller number has equal or more divisors, it's not highly composite.

Result:

```
n = int(input("Enter a number: "))

def count_divisors(x):
    c = 0
    for i in range(1, x + 1):
        if x % i == 0:
            c += 1
    return c

max_divisors = 0
is_highly_composite = True

for i in range(1, n):
    if count_divisors(i) >= count_divisors(n):
        is_highly_composite = False
        break

if is_highly_composite:
    print("Highly composite number")
else:
    print("Not highly composite number")
```

```
Enter a number:  17
Not highly composite number
```

SKILLS ACHIEVED:

Algorithm Optimization: Using square root method for efficient divisor counting.

Comparative Analysis: Evaluating a number against all smaller numbers.

Practical No: 5

Date: 10/11/2025

TITLE:mod_exp(base, exponent,modulus) function

AIM/OBJECTIVE(s):Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (base^exponent) % modulus.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

This Python program performs **modular exponentiation**. It takes base, exponent, and modulus as inputs, then repeatedly multiplies the base and applies the modulus in each iteration. This keeps intermediate results small and prevents overflow. Finally, it prints `(base^exponent) % modulus`, a key operation in cryptography and modular arithmetic..

RESULT:

```
base = int(input("Enter base: "))
exponent = int(input("Enter exponent: "))
modulus = int(input("Enter modulus: "))

result = 1
for i in range(exponent):
    result = (result * base) % modulus

print("Result:",result)
```

```
Enter base:  17
Enter exponent:  19
Enter modulus:  11
Result: 2
```

SKILLS ACHIEVED:

Binary Exponentiation: Implementing efficient power computation using bit manipulation.

Modular Arithmetic: Applying mathematical properties to handle large numbers efficiently.