Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10734
Name of Student	: Atharav Balaji Khonde
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCAI
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).	**12-11-2025**	
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).**	**12-11-2025**	
3	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	**12-11-2025**	
4	**Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.**	**12-11-2025**	
5	**Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.**	**12-11-2025**	
6			
7			

8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: 12-11-2025

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 code uses a brute-force search methodology.
It uses for loop,range() and many arithmetic operations
It employs two standard Python libraries as tools for performance analysis:

1. time module: Used to measure the execution duration (benchmarking) of the algorithm.
2. tracemalloc module: Used to measure the peak memory usage (profiling) of the traced code section.

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: 8

Sum of proper divisors: 7

```python
import time
import tracemalloc
n=int(input("enter a number"))
start=time.time()
tracemalloc.start()
def aliquot_sum(n):
    sum_div = sum(i for i in range(1, n) if n % i == 0)
    return sum_div
print("Sum of proper divisors:", aliquot_sum(n))
end=time.time()
peak,current=tracemalloc.get_traced_memory()
tracemalloc.stop()
print("memory utilzied is ",current)
print("execution time is",end-start)
```

```
enter a number8
Sum of proper divisors: 7
memory utilzied is  36037
execution time is 0.00074720382690429 69
```

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: 12-11-2025

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:
The code uses a brute-force search methodology.
It uses for loop,range() and many arithmetic operations
It employs two standard Python libraries as tools for performance analysis:

1. time module: Used to measure the execution duration (benchmarking) of the algorithm.
2. tracemalloc module: Used to measure the peak memory usage (profiling) of the traced code section.

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: 6

Enter second number : 8

6 and 8 are not amicable numbers.

```
import time
import tracemalloc
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
start=time.time()
tracemalloc.start()
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

if are_amicable(a, b):
    print(a, "and", b, "are amicable numbers.")
else:
    print(a, "and", b, "are not amicable numbers.")
end=time.time()
peak,current=tracemalloc.get_traced_memory()
tracemalloc.stop()
print("execution time is ",end-start)
print( "memory used is",peak)
```

```
Enter first number: 6
Enter second number: 8
6 and 8 are not amicable numbers.
execution time is  0.0011260509490966797
memory used is 3452
```

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: 12-11-2025

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 code uses a brute-force search methodology.
It uses for loop,range() and many arithmetic operations
It employs two standard Python libraries as tools for performance analysis:
time module: Used to measure the execution duration (benchmarking) of the algorithm.

tracemalloc module: Used to measure the peak memory usage (profiling) of the traced code section.

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 : 5

Multiplicative persistence: 0

```
import time
import tracemalloc
n = int(input("Enter a number: "))
start=time.time()
tracemalloc.start()
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
print("Multiplicative persistence:", multiplicative_persistence(n))
peak,current=tracemalloc.get_traced_memory()
tracemalloc.stop()
end=time.time()
print("memory utilzied is ",peak)
print("execution time is",end-start)
```

```
Enter a number: 5
Multiplicative persistence: 0
memory utilzied is  1998
execution time is 0.0009741783142089844
```

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: 12-11-2025

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 code uses a brute-force search methodology.
It uses for loop,range() and many arithmetic operations
It employs two standard Python libraries as tools for performance analysis:

1. time module: Used to measure the execution duration (benchmarking) of the algorithm.
2. tracemalloc module: Used to measure the peak memory usage (profiling) of the traced code section.

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.

```
import time
import tracemalloc
n = int(input("Enter a number: "))
start=time.time()
tracemalloc.start()
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

if is_highly_composite(n):
    print(n, "is a highly composite number.")
else:
    print(n, "is not a highly composite number.")
peak,current=tracemalloc.get_traced_memory()
tracemalloc.stop()
end=time.time()
print("memory utilzied is ",peak)
print("execution time is",end-start)
```

```
Enter a number: 7
7 is not a highly composite number.
memory utilzied is  3388
execution time is 0.0013537406921386719
```

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: 12-11-2025

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:
The code uses a brute-force search methodology.
It uses for loop,range() and many arithmetic operations
It employs two standard Python libraries as tools for performance analysis:

1. time module: Used to measure the execution duration (benchmarking) of the algorithm.
2. tracemalloc module: Used to measure the peak memory usage (profiling) of the traced code section.

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

```
import time
import tracemalloc
base = int(input("Enter base: "))
exponent = int(input("Enter exponent: "))
modulus = int(input("Enter modulus: "))
start=time.time()
tracemalloc.start()
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

print("Result:", mod_exp(base, exponent, modulus))
peak,current=tracemalloc.get_traced_memory()
tracemalloc.stop()
end=time.time()
print("memory utilzied is ",peak)
print("execution time is",end-start)
```

```
Enter base: 5
Enter exponent: 2
Enter modulus: 5
Result: 0
memory utilzied is  1851
```

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 computatio


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL