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:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Aliquot Sum Calculator	16/11/20225	
2	Amicable Numbers Check	16/11/20225	
3	Multiplicative Persistence Calculator	16/11/20225	
4	Highly Composite Number Check	16/11/20225	
5	Modular Exponentiation	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Aliquot Sum Calculator

AIM/OBJECTIVE(s) :To write a Python function `aliquot_sum(n)` that returns the sum of all proper divisors of `n` (which are all divisors of `n` other than `n` itself).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

The code defines a function `aliquot_sum(n)` that efficiently calculates the sum of proper divisors. It handles the edge case of `n <= 1`. The loop is optimized to only run up to `sqrt(n)`, making it much faster than checking all numbers up to `n/2`.

RESULTS ACHIEVED:

cse project > week5 > main2.py > ...

```python
import time
import tracemalloc
import math

def aliquot_sum(n):
    if n <= 1:
        return 0

    total = 1
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            total += i
            if i * i != n:
                total += n // i
    return total

if __name__ == "__main__":

    test_number = 220

    tracemalloc.start()
    start_time = time.perf_counter()

    result = aliquot_sum(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Calculating aliquot sum of {test_number}.")
    print(f"Result (sum of proper divisors): {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")


    test_number_2 = 12

    tracemalloc.start()
    start_time = time.perf_counter()

    result_2 = aliquot_sum(test_number_2)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main2.py"
Calculating aliquot sum of 12.
Result (sum of proper divisors): 16
Execution Time: 0.000028700 seconds
Peak Memory Usage: 0.75 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding the concept of a modular multiplicative inverse.
- Using Python's `pow()` function for advanced number theory operations.
- Using `try...except` blocks to handle expected mathematical errors (`ValueError`).
- Continued practice in performance measurement.

Practical No: 2

Date: 16/11/2025

TITLE: Amicable Numbers Check

AIM/OBJECTIVE(s) : To write a Python function `are_amicable(a, b)` that checks if two numbers `a` and `b` are amicable. Two numbers are amicable if the sum of the proper divisors of `a` is equal to `b`, and the sum of the proper divisors of `b` is equal to `a`.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines `are_amicable(a, b)` and re-uses the `aliquot_sum(n)` helper function. The `are_amicable` function directly implements the mathematical definition by calling the helper function twice and comparing the results.

Result:

```python
import time
import tracemalloc
from functools import reduce

def mod_inverse(a, m):
    try:
        return pow(a, -1, m)
    except ValueError:
        return None

def crt(remainders, moduli):
    if len(remainders) != len(moduli):
        return None

    M = reduce(lambda a, b: a * b, moduli)

    total = 0
    for r_i, m_i in zip(remainders, moduli):
        M_i = M // m_i
        y_i = mod_inverse(M_i, m_i)

        if y_i is None:
            return None

        total += r_i * M_i * y_i

    return total % M

if __name__ == "__main__":

    remainders = [2, 3, 2]
    moduli = [3, 5, 7]

    tracemalloc.start()
    start_time = time.perf_counter()

    result = crt(remainders, moduli)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Solving system of congruences:")
    for r, m in zip(remainders, moduli):
        print(f"x ≡ {r} mod {m}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\WEEK6\main2.py"

Result (x): 23
Execution Time: 0.000838100 seconds
Peak Memory Usage: 0.20 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED :

- Re-using helper functions to build more complex logic.
- Implementing a mathematical definition involving multiple conditions.
- Testing function with both positive and negative cases.

Practical No: 3

Date: 16/11/2025

TITLE : Multiplicative Persistence Calculator

AIM/OBJECTIVE(s) : To write a Python function `multiplicative_persistence(n)` that counts how many steps are needed to multiply a number's digits until a single-digit number is reached.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION : The code defines `multiplicative_persistence(n)`. It uses a `while` loop to repeatedly process the number. In each step, it converts the number to a list of its digits, calculates their product (using `reduce`), and updates the number to this product, incrementing a step counter. The process stops when the number is less than 10.

RESULTS ACHIEVED:

cse project > week5 > main3.py > ...

```python
import time
import tracemalloc
from functools import reduce

def multiplicative_persistence(n):
    if n < 0:
        n = abs(n)

    count = 0
    while n >= 10:
        digits = [int(d) for d in str(n)]
        n = reduce(lambda x, y: x * y, digits)
        count += 1
    return count

if __name__ == "__main__":

    test_number = 77

    tracemalloc.start()
    start_time = time.perf_counter()

    result = multiplicative_persistence(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Calculating multiplicative persistence of {test_number}.")
    print(f"Result (steps): {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")


    test_number_2 = 123

    tracemalloc.start()
    start_time = time.perf_counter()

    result_2 = multiplicative_persistence(test_number_2)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main3.py"
Calculating multiplicative persistence of 123.
Result (steps): 1
Execution Time: 0.000027000 seconds
Peak Memory Usage: 0.75 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Type conversion (int to string, string to int list).
- Using `while` loops for a process that repeats an unknown number of times.
- Using `functools.reduce` for a cumulative operation.
- String and list manipulation.

Practical No: 4

Date: 16/11/2025

TITLE : Highly Composite Number Check

AIM/OBJECTIVE(s) : To write a Python function `is_highly_composite(n)` that checks if an integer `n` is a highly composite number (HCN). An HCN is a positive integer that has more divisors than any smaller positive integer.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

1. The code defines `is_highly_composite(n)` and a helper `get_divisor_count(n)`. The main function implements the definition of an HCN by finding the divisor count of `n` and comparing it to the divisor count of all positive integers less than `n`.

Result:

cse project > week5 > main4.py > ...

```python
import time
import tracemalloc
import math

def get_divisor_count(n):
    if n == 0:
        return 0

    count = 0
    for i in range(1, int(math.sqrt(n)) + 1):
        if n % i == 0:
            if i * i == n:
                count += 1
            else:
                count += 2
    return count

def is_highly_composite(n):
    if n <= 1:
        return False

    divisor_count_n = get_divisor_count(n)

    for i in range(1, n):
        if get_divisor_count(i) >= divisor_count_n:
            return False

    return True

if __name__ == "__main__":

    test_number = 12

    tracemalloc.start()
    start_time = time.perf_counter()

    result = is_highly_composite(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Checking if {test_number} is a highly composite number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main4.py
Checking if 10 is a highly composite number.
Result: False
Execution Time: 0.000055400 seconds
Peak Memory Usage: 0.11 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED :

- Implementation of a complex number theory definition.
- Creating and using efficient helper functions.
- Writing optimized loops for comparative analysis.

Practical No: 5

Date: 16/11/2025

TITLE : Modular Exponentiation

AIM/OBJECTIVE(s) : To write a Python function `mod_exp(base, exponent, modulus)` that efficiently calculates `(base^exponent) % modulus`.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

1. The code defines `mod_exp` which directly uses `pow(base, exponent, modulus)` to get the result. This is the standard and most efficient way to perform this operation in Python. An edge case for `modulus == 1` (where the result is always 0) is handled.

RESULT:

cse project > week5 > main5.py > ...

```python
import time
import tracemalloc

def mod_exp(base, exponent, modulus):
    if modulus == 1:
        return 0
    return pow(base, exponent, modulus)

if __name__ == "__main__":

    base = 3
    exponent = 4
    modulus = 7

    tracemalloc.start()
    start_time = time.perf_counter()

    result = mod_exp(base, exponent, modulus)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Calculating ({base}^{exponent}) % {modulus}")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")


    base_2 = 5
    exponent_2 = 123456
    modulus_2 = 13

    tracemalloc.start()
    start_time = time.perf_counter()

    result_2 = mod_exp(base_2, exponent_2, modulus_2)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time_2 = end_time - start_time

    print(f"\nCalculating ({base_2}^{exponent_2}) % {modulus_2}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Calculating (5^123456) % 13
Result: 1
Execution Time: 0.000005100 seconds
Peak Memory Usage: 0.00 KiB
```

SKILLS ACHIEVED:

- Understanding the concept and utility of modular exponentiation.
- Knowing and using the correct built-in function (`pow(b, e, m)`) for an optimized mathematical operation.
- Appreciating the difference between a naive calculation and an efficient algorithm.