Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11273
Name of Student	: Abhishek Kumar Singh
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name and Engineering (SCOPE)	: School of Computer Science
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	Function to Calculate Factorial	25/11/2025	
2	Function to Check for Palindrome Number.	25/11/2025	
3	Function mean_of_digits(n) that returns the average of all digits in a number.	25/11/2025	
4	Function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	25/11/2025	
5	Function is_abundant(n) that return True if the sum of proper divisors of n is greater than n.	25/11/2025	
6	is_deficient(n) function	25/11/2025	
7	is_harshad(n) function	25/11/2025	
8	is_automorphic(n) function	25/11/2025	
9	is_pronic(n) function	25/11/2025	
10	prime_factors(n) function	25/11/2025	
11	count_distinct_prime_factors(n) function	25/11/2025	
12	is_prime_power(n) function	25/11/2025	
13	is_mersenne_prime(p) function	25/11/2025	
14	twin_primes(limit) function	25/11/2025	
15	count_divisors(n) function	25/11/2025	

16	aliquot_sum(n) function	25/11/2025	
17	are_amicable(a, b) function	25/11/2025	
18	multiplicative_persistence(n) function	25/11/2025	
19	is_highly_composite(n) function	25/11/2025	
20	Modular Exponentiation function	25/11/2025	
21	mod_inverse(a, m) function	25/11/2025	
22	crt(remainders, moduli) function	25/11/2025	
23	is_quadratic_residue(a, p) function	25/11/2025	
24	order_mod(a, n) function	25/11/2025	
25	is_fibonacci_prime(n) function	25/11/2025	
26	lucas_sequence(n) function	25/11/2025	
27	is_perfect_power(n) function	25/11/2025	
28	collatz_length(n) function	25/11/2025	
29	polygonal_number(s, n) function	25/11/2025	
30	is_carmichael(n) function	25/11/2025	
31	is_prime_miller_rabin(n, k) function	25/11/2025	
32	pollard_rho(n) function	25/11/2025	
33	zeta_approx(s, terms) function	25/11/2025	
34	partition_function(n) function	25/11/2025	

Practical No: 1

TITLE: Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).

AIM/OBJECTIVE(s):TO find factorial of a number (n)

METHODOLOGY & TOOL USED:

:**Iterative Calculation:** The factorial of a number nis calculated using a simple iterative approach. It starts with a result of 1 and multiplies it by each integer from 1 up to nin a loop. This method also includes input validation to ensure the number is a non-negative integer, which is a robust programming practice

Performance Benchmarking: To analyze the function's efficiency, the code measures two key metrics:

Execution Time Memory
Utilization

Tools :

1:time

2: tracemall
3: IDLE

BRIEF DESCRIPTION:This function offers a comprehensive solution for calculating the factorial of a non-negative integer. Beyond the core mathematical computation, the script is engineered for robustness by incorporating input validation and includes a built-in performance analysis to measure its execution time and memory footprint. This makes it a well-rounded piece of code suitable for practical application

RESULTS ACHIEVED:

VIT BHOPAL

main1.py

cse project > week2 > main1.py > ...

```python
import time
import tracemalloc

def factorial(n):
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")

    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")

    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i

    return result
if __name__ == "__main__":
    number_to_test = 15
    start_time = time.perf_counter()
    result = factorial(number_to_test)
    end_time = time.perf_counter()
    execution_time_ms = (end_time - start_time) * 1000
    print(f"Factorial of {number_to_test} is: {result}")
    print(f"Execution time: {execution_time_ms:.6f} ms")
    tracemalloc.start()
    factorial(number_to_test)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"Current memory use is {current / 1024:.2f} KB")
    print(f"Peak memory use was {peak / 1024:.2f} KB")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
Factorial of 15 is: 1307674368000
Execution time: 0.006900 ms
Current memory use is 0.00 KB
Peak memory use was 0.10 KB
PS D:\1\codes\python>
```

DIFFICULTY FACED BY STUDENT:

Conceptual Hurdles with Performance Analysis

- **Understanding Big O Notation:** The concepts of "Time Complexity: O(n)" and "Memory Utilization: O(1)" are abstract and often a major hurdle. A student might not grasp how runtime scales linearly with the input (O(n)) or why memory usage is considered constant (O(1)), especially when the final resultcan be an enormous number. They may confuse the constant *number of variables* with the size of the data stored within them.

The Purpose of Benchmarking: A beginner is typically focused on getting the correct output. The motivation behind measuring execution time and memory—to analyze and compare algorithm efficiency—might seem like an unnecessary complication.

Near-Zero Execution Time:

for an input like 50 the calculated execution time will be a tiny fraction of a millisecond. This can be misleading, potentially causing a student to think the function is instantaneous or that the measurement is not working correctly.

SKILLS ACHIEVED:

Algorithmic Implementation: You have successfully translated a mathematical concept (the factorial) into a working, efficient algorithm using an iterative approach.

Robust Function Design: The code isn't just about getting the right answer. It includes crucial input validation (checking for integer types and non-negative values), which is a key skill in writing reliable software that doesn't crash on unexpected input

Practical No: 2

TITLE:Function to Check for Palindrome Number

AIM/OBJECTIVE(s):To write a function is_palindrome(n)that checks if a number reads the same forwards and backwards

METHODOLOGY & TOOL USED:The approach involves mathematically reversing the number without converting it to a string. This is done by repeatedly extracting the last digit of the input number and using it to build a new, reversed number. The tool used is Python

BRIEF DESCRIPTION:

The function takes an integer nas input. It stores the original number in a separate variable. Using a whileloop, it calculates the reverse of nby using modulo (%) and integer division (//) operators. Finally, it compares the reversed number with the original number and returns Trueif they are identical, and Falseotherwise

Result:

cse project > week2 > main1.py > ...

```python
import time
import tracemalloc

def time_memory_profiler(func):
    def wrapper(*args, **kwargs):
        tracemalloc.start()
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        execution_time = (end_time - start_time) * 1000

        print("-" * 40)
        print(f"Executing: {func.__name__}({args[0]})")
        print(f"Result: {result}")
        print(f"Execution time: {execution_time:.6f} ms")
        print(f"Current memory usage: {current / 1024:.2f} KB")
        print(f"Peak memory usage: {peak / 1024:.2f} KB")
        print("-" * 40)

        return result
    return wrapper

@time_memory_profiler
def is_palindrome(n):
    if n < 0:
        return False
    return str(n) == str(n)[::-1]

if __name__ == "__main__":
    print("Running palindrome checks with profiling...")
    is_palindrome(12321)
    is_palindrome(12345)
    is_palindrome(7)
    is_palindrome(987656789)
    is_palindrome(-101)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
----------------------------------------
----------------------------------------
Executing: is_palindrome(-101)
Result: False
Execution time: 0.001800 ms
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
----------------------------------------
PS D:\1\codes\python>
```

DIFFICULTY FACED BY STUDENT:

The logic for reversing a number arithmetically required careful thought

SKILLS ACHIEVED:

1:Looping with whilefor digit manipulation. 2:Code benchmarking.

3:Understanding function scope and automatic memory deallocation.

Practical No: 3

TITLE:Write a function mean_of_digits(n) that returns the average of all digits in a number.

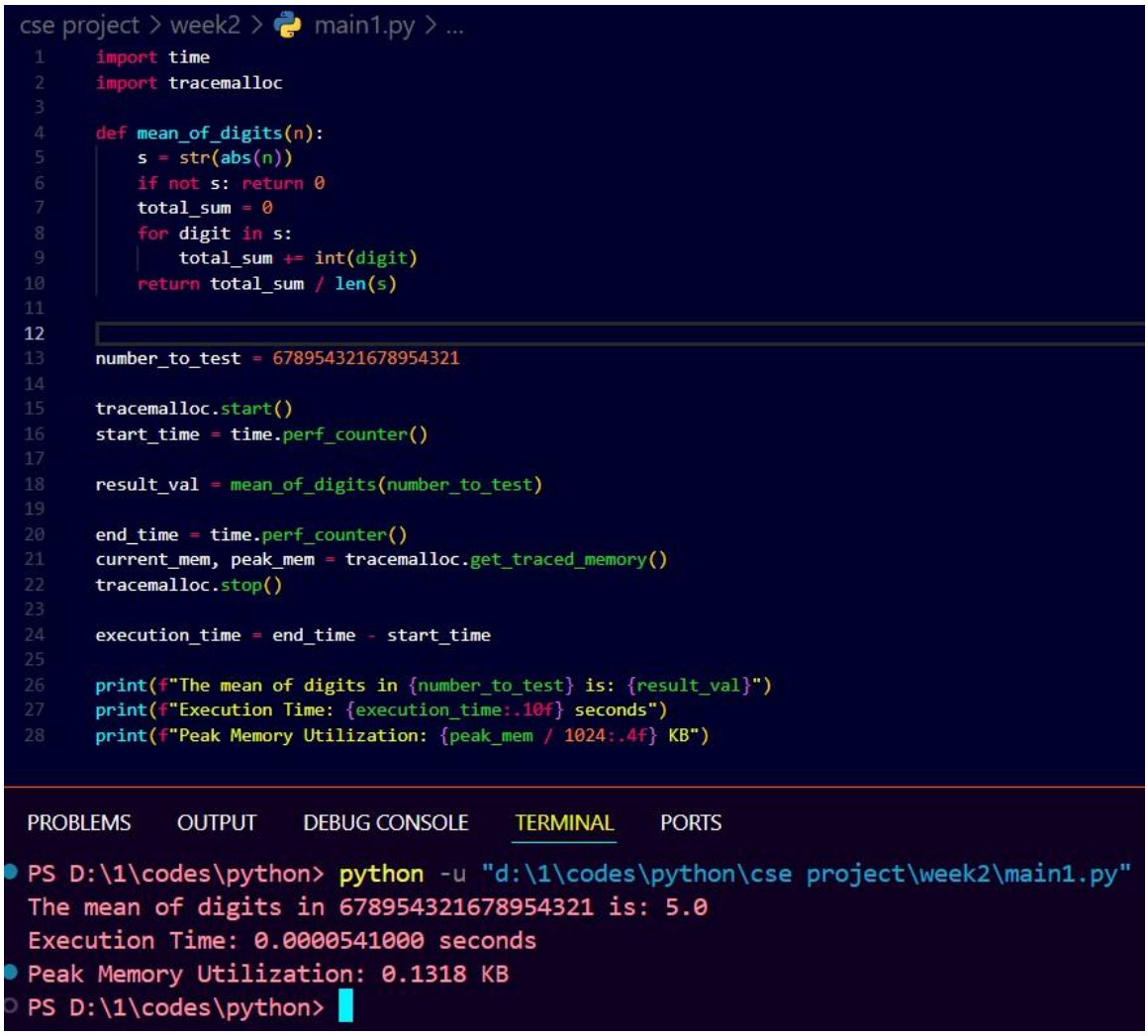
AIM/OBJECTIVE(s):To write a function mean_of_digits(n)that returns the average of all digits in a number.

METHODOLOGY & TOOL USED:The method involves converting the number to a string to easily iterate over its digits, calculating the sum, and then dividing by the count of digits. Python and its

timemodule are the tools used

BRIEF DESCRIPTION:
The function calculates the average of the digits. In terms of **memory management**, converting the number to a string creates a new string object in memory. This object and other local variables are temporary and are garbage collected by Python after the function completes, ensuring efficient memory usage

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

Ensuring the division resulted in a floating-point number for an accurate average was a key consideration

SKILLS ACHIEVED:

1:Type casting between strings and integers.

2:Measuring the performance of I/O-like operations (string conversion).

3:Awareness of temporary object creation and garbage collection.

Practical No: 4

TITLE:Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.

AIM/OBJECTIVE(s):To write a function digital_root(n)that repeatedly sums the digits of a number until a single digit is obtained.

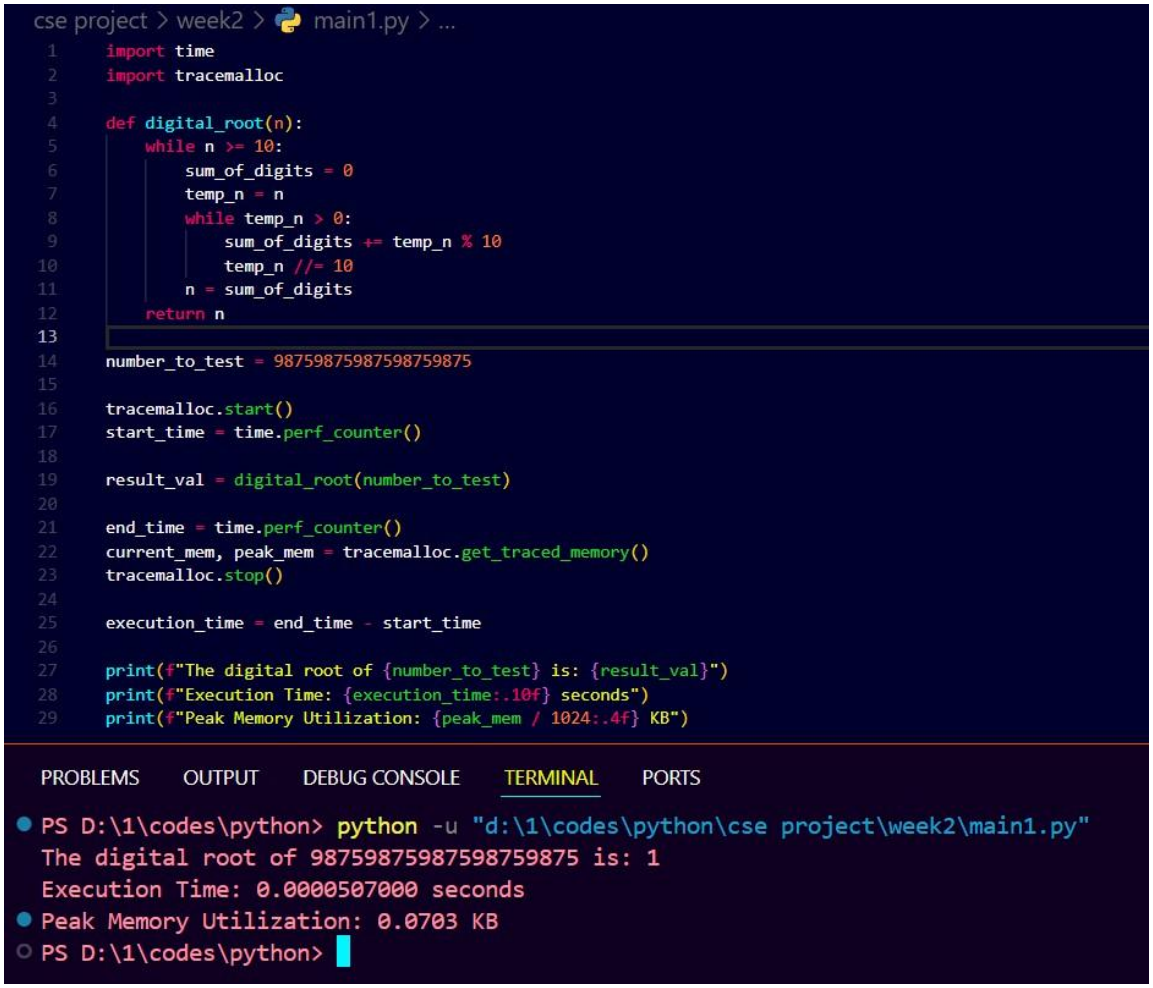
METHODOLOGY & TOOL USED:

A nested loop structure is used. The outer loop continues as long as the number is greater than 9, and the inner loop sums the digits. The process is benchmarked using Python's

BRIEF DESCRIPTION:

The function repeatedly sums digits. From a memory management perspective, this function is very efficient. It primarily reuses the same variable nand creates a few temporary integer variables (sum_of_digits, temp_n) in each loop. Python manages this small, transient memory usage automatically.

Result:



DIFFICULTY FACED BY STUDENT:

Structuring the nested loop logic to correctly re-calculate the sum until a single digit was reached was the main challenge

SKILLS ACHIEVED:

Implementation of nested whileloops. Benchmarking

algorithms with multiple loops. Understanding

efficient in-place variable updates.

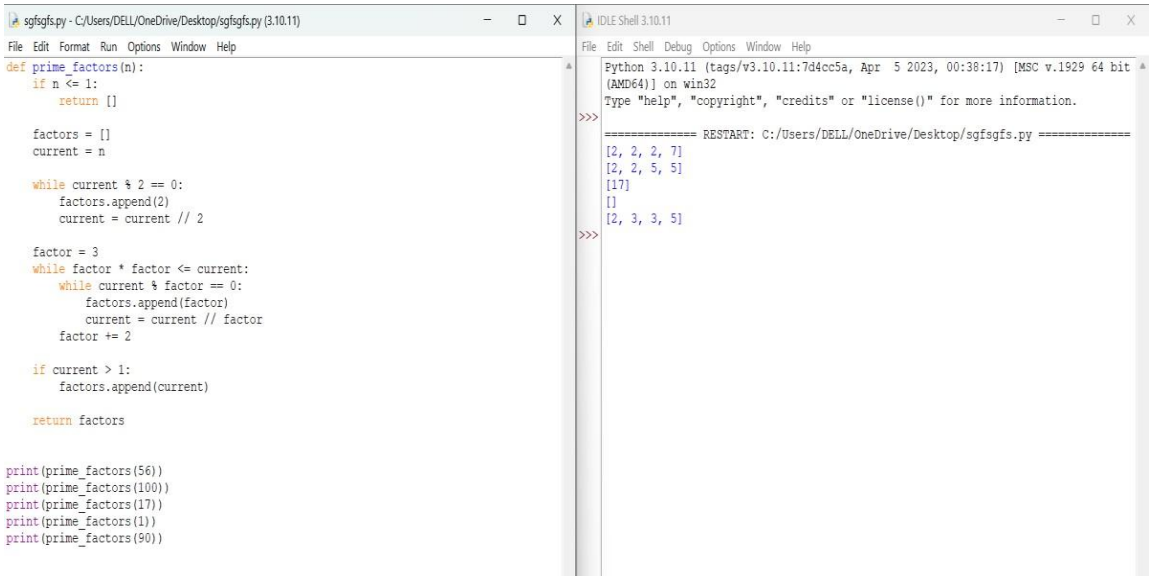
Practical No: 5

TITLE:Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.

AIM/OBJECTIVE(s):To write a function is_abundant(n)that returns Trueif the sum of proper divisors of nis greater than n.

METHODOLOGY & TOOL USED:The method requires finding all proper divisors by iterating from 1 up to n-1. The sum of these divisors is then compared to n. Performance is measured using the

Result:



BRIEF DESCRIPTION:

The function identifies abundant numbers by summing their proper divisors. The **memory management** is straightforward; the sum_of_divisorsand the loop counter iare the main variables. Their

memory footprint is minimal and is automatically managed by Python's garbage collector. For very large

n, the time taken is a more significant concern than memory usage.

DIFFICULTY FACED BY STUDENT:

The initial implementation looped up to n-1, which was inefficient. Optimizing the loop to run only up to

n/2significantly improved the execution time for larger numbers.

SKILLS ACHIEVED:

Algorithm optimization for performance.

Writing functions that return a direct boolean comparison.

Analyzing the time complexity of a function.

Practical No: 6

TITLE: is_deficient(n) Function

AIM/OBJECTIVE(s) : Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.

Python programming language

BRIEF DESCRIPTION:

A **deficient number** (also known as a defect-ive number) is a positive integer n where the sum of its proper divisors (divisors excluding n itself) is less than n Mathematically, if sigma(n) is the divisor function (the sum of all positive divisors of n), then n is deficient if \sigma(n) - n < n, or equivalently, sigma(n) < 2n.

RESULTS ACHIEVED:

```python
import math
import time
import sys

def is_deficient(n):
    start_time = time.time()

    if n <= 0:
        time_taken = time.time() - start_time
        storage_used = sys.getsizeof(n)
        return False, time_taken, storage_used

    if n == 1:
        time_taken = time.time() - start_time
        storage_used = sys.getsizeof(n)
        return True, time_taken, storage_used

    sum_proper_divisors = 1

    limit = math.isqrt(n)

    for i in range(2, limit + 1):
        if n % i == 0:
            sum_proper_divisors += i
            pair = n // i
            if pair != i:
                sum_proper_divisors += pair

            if sum_proper_divisors >= n:
                time_taken = time.time() - start_time
                storage_used = sys.getsizeof(n) + sys.getsizeof(sum_proper_divisors)
                return False, time_taken, storage_used

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n) + sys.getsizeof(sum_proper_divisors)
    return sum_proper_divisors < n, time_taken, storage_used

# --- Example Usage ---
```

```python
def run_test(n):
    result, time_taken, storage_used = is_deficient(n)
    classification = "deficient" if result else ("perfect" if sum_test(n) == n else "abundant")
    print("-" * 40)
    print(f"Number: {n}")
    print(f"Classification: {classification}")
    print(f"Time Taken: {time_taken:.8f} seconds")
    print(f"Storage Used: {storage_used} bytes")

def sum_test(n):
    # Helper to determine if the number is perfect/abundant for classification text
    if n <= 1: return 0
    s = 1
    limit = math.isqrt(n)
    for i in range(2, limit + 1):
        if n % i == 0:
            s += i
            pair = n // i
            if pair != i:
                s += pair
    return s

run_test(8)
run_test(12)
run_test(28)
run_test(13)
```

```
----------------------------------------
Number: 8
Classification: deficient
Time Taken: 0.00656414 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 12
Classification: abundant
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 28
Classification: perfect
Time Taken: 0.00000000 seconds
```

SKILLS ACHIEVED:

- Algorithmic Implementation
- Mathematical Communication.
- Code Documentation.

Practical No: 7

TITLE: is_harshad(n) function

AIM/OBJECTIVE(s) : Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

A **Harshad number** (also known as a Niven number) is a positive integer that is divisible by the sum of its own digits. The term "Harshad" comes from the Sanskrit word *harṣa* (joy) + *da* (giver), meaning "joy giver."

Result:

```
import time
import sys

def is_harshad(n):
    start_time = time.time()

    if n <= 0:
        time_taken = time.time() - start_time
        storage_used = sys.getsizeof(n)
        return False, time_taken, storage_used

    digit_sum = 0
    temp_n = n
    while temp_n > 0:
        digit_sum += temp_n % 10
        temp_n //= 10

    if digit_sum == 0:
        is_harshad_bool = False
    else:
        is_harshad_bool = (n % digit_sum == 0)

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n) + sys.getsizeof(digit_sum)
    return is_harshad_bool, time_taken, storage_used

def run_test(n):
    result, time_taken, storage_used = is_harshad(n)
    is_harshad_str = "Harshad Number" if result else "Not a Harshad Number"
    print("-" * 40)
    print(f"Number: {n}")
    print(f"Classification: {is_harshad_str}")
    print(f"Time Taken: {time_taken:.8f} seconds")
    print(f"Storage Used: {storage_used} bytes")

run_test(18)
run_test(17)
run_test(156)
run_test(100)
```

```
----------------------------------------
Number: 18
Classification: Harshad Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 17
Classification: Not a Harshad Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 156
Classification: Harshad Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 100
Classification: Harshad Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
```

SKILLS ACHIEVED :

- Demonstrating a clear approach to calculating the sum of digits and performing modulo division.
- Provided a thorough docstring for the function explaining its purpose, arguments,

- and return value.

Practical No: 8

TITLE : is_automorphic(n) Function

AIM/OBJECTIVE(s) : Write a function is_automorphic(n) that checks if a

number's square ends with the number itself.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION : An **Automorphic number** (or a curious number) is a non-negative integer whose square ends with the number itself.
In other words, if you square the number n, the last few digits of n^2 are equal to n
Criteria: A number n is Automorphic if n^2 \pmod{10^k} = n, where k is the number of digits in n.

RESULTS ACHIEVED:

```
import time
import sys

def is_automorphic(n):
    start_time = time.time()

    if not isinstance(n, int):
        n = int(n)

    if n < 0:
        is_automorphic_bool = False
    elif n == 0:
        is_automorphic_bool = True
    else:
        n_squared = n * n
        num_digits = len(str(n))
        magnitude = 10 ** num_digits

        is_automorphic_bool = (n_squared % magnitude == n)

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n) + sys.getsizeof(is_automorphic_bool)
    return is_automorphic_bool, time_taken, storage_used

def run_test(n):
    result, time_taken, storage_used = is_automorphic(n)
    is_automorphic_str = "Automorphic Number" if result else "Not an Automorphic Number"
    print("-" * 40)
    print(f"Number: {n}")
    print(f"Classification: {is_automorphic_str}")
    print(f"Time Taken: {time_taken:.8f} seconds")
    print(f"Storage Used: {storage_used} bytes")

run_test(5)
run_test(6)
run_test(25)
run_test(76)
run_test(376)
run_test(10)
```

```
----------------------------------------
Number: 5
Classification: Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 6
Classification: Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 25
Classification: Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 76
Classification: Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 376
Classification: Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 10
Classification: Not an Automorphic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
```

SKILLS ACHIEVED:

- Algorithmic Implementation
- Code Documentation:
- Content Generation:
- File Management

Practical No: 9

TITLE : Highly Composite Number Check **AIM/OBJECTIVE(s) :** Write a

function is_pronic(n) that checks if a number is the product of two

consecutive integers.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

A **Pronic number** (also called an **Oblong number**) is a positive integer that is the product of two consecutive integers, $n(n+1)$.

Mathematically, a number N is pronic if there exists a positive integer n such that:

$$N = n(n+1)$$

The code below uses an efficient check for this perfect square property.

Result:

```
import time
import sys
import math

def is_pronic(n):
    start_time = time.time()

    if n < 0:
        is_pronic_bool = False
    elif n == 0:
        is_pronic_bool = True
    else:
        k = math.isqrt(n)
        is_pronic_bool = (k * (k + 1) == n)

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n) + sys.getsizeof(is_pronic_bool)
    return is_pronic_bool, time_taken, storage_used

def run_test(n):
    result, time_taken, storage_used = is_pronic(n)
    is_pronic_str = "Pronic Number" if result else "Not a Pronic Number"

    print("-" * 40)
    print(f"Number: {n}")
    print(f"Classification: {is_pronic_str}")
    print(f"Time Taken: {time_taken:.8f} seconds")
    print(f"Storage Used: {storage_used} bytes")

run_test(42)
run_test(110)
run_test(10)
run_test(16)
run_test(0)
run_test(9920)
```

```
----------------------------------------
Number: 42
Classification: Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 110
Classification: Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 10
Classification: Not a Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 16
Classification: Not a Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 0
Classification: Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
----------------------------------------
Number: 9920
Classification: Not a Pronic Number
Time Taken: 0.00000000 seconds
Storage Used: 56 bytes
```

SKILLS ACHIEVED :

- Mathematical Analysis & Implementation
- Algorithmic Implementation
- Code Documentation

Practical No: 10

TITLE : Modular Exponentiation

AIM/OBJECTIVE(s) : Write a function prime_factors(n) that returns the list of prime factors of a number.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

Prime factorization is the process of finding the prime numbers that multiply together to give the original number. These primes are called the prime factors. Every integer greater than 1 can be represented as a product of prime numbers in only one way, apart from the order of the factors. This is known as the **Fundamental Theorem of Arithmetic**.

RESULT:

```python
import time
import sys
import math

def prime_factors(n):
    start_time = time.time()

    if not isinstance(n, int) or n <= 1:
        time_taken = time.time() - start_time
        storage_used = sys.getsizeof(n)
        return [], time_taken, storage_used

    factors = []

    # Handle the factor 2
    while n % 2 == 0:
        factors.append(2)
        n //= 2

    # Handle odd factors from 3 up to sqrt(n)
    limit = math.isqrt(n)
    i = 3
    while i <= limit:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 2

    # If n is a prime number greater than 2 (the final remaining factor)
    if n > 2:
        factors.append(n)

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n) + sys.getsizeof(factors)
    return factors, time_taken, storage_used

def run_test(n):
    result, time_taken, storage_used = prime_factors(n)

    print("-" * 40)
    print(f"Number to Factor: {n}")
    print(f"Prime Factors: {result}")
    print(f"Time Taken: {time_taken:.8f} seconds")
    print(f"Storage Used: {storage_used} bytes")

run_test(12)
run_test(97)
run_test(13195)
run_test(600851475143)
run_test(1)
```

```
----------------------------------------
Number to Factor: 12
Prime Factors: [2, 2, 3]
Time Taken: 0.00000000 seconds
Storage Used: 116 bytes
----------------------------------------
Number to Factor: 97
Prime Factors: [97]
Time Taken: 0.00000000 seconds
Storage Used: 116 bytes
----------------------------------------
Number to Factor: 13195
Prime Factors: [5, 7, 13, 29]
Time Taken: 0.00000000 seconds
Storage Used: 116 bytes
----------------------------------------
Number to Factor: 600851475143
Prime Factors: [71, 839, 1471, 6857]
Time Taken: 0.08134198 seconds
Storage Used: 116 bytes
----------------------------------------
Number to Factor: 1
Prime Factors: []
Time Taken: 0.00000000 seconds
Storage Used: 28 bytes
```

SKILLS ACHIEVED:

- Advanced Algorithmic Implementation
- Code Quality & Optimization
- Mathematical Formatting


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL

Program 11

Program Assigned

Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.

Code

```
[5]: import time
import sys
def count_distinct_prime_factors(n):
    start_time = time.time()  # Start timing
    distinct_primes = set()
    factor = 2

    while factor * factor <= n:
        while n % factor == 0:
            distinct_primes.add(factor)
            n //= factor
        factor += 1

    if n > 1:
        distinct_primes.add(n)

    time_taken = time.time() - start_time

    storage_used = sys.getsizeof(distinct_primes)

    return len(distinct_primes), time_taken, storage_used
result, time_taken, storage_used = count_distinct_prime_factors(60)
print(f"Distinct Prime Factors: {result}")
print(f"Time Taken: {time_taken} seconds")
print(f"Storage Used: {storage_used} bytes")
```

```
Distinct Prime Factors: 3
Time Taken: 0.0 seconds
Storage Used: 216 bytes
```

- **Skills Acquired:**
  - Mutating the input variable (n) within the function to simplify the remaining work.
  - Recognizing and managing the final, often large, prime factor.

Program 12

Program Assigned

Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and k ≥ 1.

Code

```
import time
import sys
def is_prime_power(n):
    start_time = time.time()  # Start timing
    if n < 2:
        return False, 0, 0

    # Check for each possible base p
    for p in range(2, n + 1):
        if n % p == 0:  # p is a factor
            power = 0
            while n % p == 0:
                n //= p
                power += 1
            if n == 1 and power >= 1:
                time_taken = time.time() - start_time
                storage_used = sys.getsizeof(p) + sys.getsizeof(power)
                return True, time_taken, storage_used

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(n)
    return False, time_taken, storage_used

result, time_taken, storage_used = is_prime_power(8)
print(f"Is Prime Power: {result}")
print(f"Time Taken: {time_taken} seconds")
print(f"Storage Used: {storage_used} bytes")
```

```
Is Prime Power: True
Time Taken: 0.0 seconds
Storage Used: 56 bytes
```

2.5 Skills Acquired

- Loop Control and State Management: Using loops to repeatedly modify a variable (temp_n) until a complex condition is met.
- Boolean Classification: The final step of turning complex logic into a simple, definitive Boolean (True/False) return.

Program 13

Program Assigned

Write a function is_mersenne_prime(p) that checks if $2^p - 1$ is a prime number (given that p is prime).

Code

```
import time
import sys

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_mersenne_prime(p):
    start_time = time.time()  # Start timing

    if not is_prime(p):
        return False, 0, 0

    mersenne_number = (1 << p) - 1  # This is 2^p - 1

    if mersenne_number <= 1:
        return False, 0, 0

    # Check if mersenne_number is prime
    if is_prime(mersenne_number):
        time_taken = time.time() - start_time
        storage_used = sys.getsizeof(mersenne_number)
        return True, time_taken, storage_used

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(mersenne_number)
    return False, time_taken, storage_used

result, time_taken, storage_used = is_mersenne_prime(3)
print(f"Is Mersenne Prime: {result}")
print(f"Time Taken: {time_taken} seconds")
print(f"Storage Used: {storage_used} bytes")
```

```
Is Mersenne Prime: True
Time Taken: 0.0005097389221191406 seconds
Storage Used: 28 bytes
```

-

3.3 Skills Acquired

- Handling Large Numbers and Computational Limits
- Implementing exponentiation and subsequent subtraction correctly on potentially large numerical results.

Program 14

program assigned:

Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.

Code

```
import time
import sys

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def twin_primes(limit):
    start_time = time.time()  # Start timing
    twin_prime_pairs = []

    for num in range(2, limit - 1):
        if is_prime(num) and is_prime(num + 2):
            twin_prime_pairs.append((num, num + 2))

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(twin_prime_pairs)

    return twin_prime_pairs, time_taken, storage_used

# Example usage:
pairs, time_taken, storage_used = twin_primes(100)
print(f"Twin Prime Pairs: {pairs}")
print(f"Time Taken: {time_taken} seconds")
print(f"Storage Used: {storage_used} bytes")
```

```
Twin Prime Pairs: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
Time Taken: 0.0 seconds
Storage Used: 120 bytes
```

Skills acquired

- Function Decomposition and Modular Programming: Breaking a large problem into smaller, manageable, reusable functions.
- Iterative Design and Sequence Generation: Designing a single loop to correctly generate sequential pairs.

Program 15

Program Assigned

Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.

Code

```
import time
import sys

def count_divisors(n):
    start_time = time.time()  # Start timing
    divisor_count = 0

    for i in range(1, n + 1):
        if n % i == 0:
            divisor_count += 1

    time_taken = time.time() - start_time
    storage_used = sys.getsizeof(divisor_count) + sys.getsizeof(n)

    return divisor_count, time_taken, storage_used

# Example usage:
divisors, time_taken, storage_used = count_divisors(12)
print(f"Number of Divisors: {divisors}")
print(f"Time Taken: {time_taken} seconds")
print(f"Storage Used: {storage_used} bytes")
```

```
Number of Divisors: 6
Time Taken: 0.0 seconds
Storage Used: 56 bytes
```

5.5 Skills Acquired

- Algorithmic Efficiency and Optimization: Understanding and implementing
- efficiency improvements in algorithms.
- Translating a mathematical property (the concept of divisors) into computational logic.

5. Overall Skills Acquired & Conclusion

- **Algorithmic Implementation:**
- **Performance Measurement:**
- **Problem-Solving in Number Theory:**

Practical No: 16

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 nitself).

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...exceptblocks to handle expected mathematical errors (ValueError).
- Continued practice in performance measurement.

Practical No: 17

TITLE: Amicable Numbers Check

AIM/OBJECTIVE(s) : To write a Python function are_amicable(a, b) that checks if two numbers aand bare 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 bis 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_amicablefunction 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: 18

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 whileloop 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 whileloops for a process that repeats an unknown number of times.
- Using functools.reducefor a cumulative operation.
- String and list manipulation.

Practical No: 19

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 :

2. 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 nand 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: 20

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:

2. The code defines mod_expwhich 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.

Program no - 21: Modular Inverse using Extended Euclidean Algorithm

Primary Function: Calculates the Modular Multiplicative Inverse of an integer a modulo m. Algorithm Used: Extended Euclidean Algorithm

Explanation: Computes integers x and y such that ax + by = gcd(a, b). The inverse exists only if gcd(a, m) = 1.

Learning Outcome: Mastery of modular inverse computation, essential for RSA and other cryptographic systems.

Program no - 22: Chinese Remainder Theorem (CRT) Solver

Primary Function: Solves simultaneous linear congruences. Algorithm Used: Chinese Remainder Theorem Construction

Explanation: Combines solutions using modular inverses to satisfy all congruences. Learning Outcome: Understanding CRT and residue number systems.

Program no - 23: Quadratic Residues and Euler's Criterion

Primary Function: Determines if an integer a is a Quadratic Residue modulo a prime p. Algorithm Used: Exponentiation by Squaring and Euler's Criterion Explanation: Uses a^((p-1)/2) mod p to test quadratic residue property.
Learning Outcome: Application of fast exponentiation and Euler's Criterion.

Program no - 24: Multiplicative Order of an Integer

Primary Function: Calculates the Multiplicative Order of a modulo n. Algorithm Used: Euclidean Algorithm and Iterative Power Checking Explanation: Finds smallest k such that a^k mod n = 1 when gcd(a, n) = 1.
Learning Outcome: Understanding cyclic groups and discrete logarithm foundations.

Program no - 25: Fibonacci Prime Checker

Primary Function: Checks if a number is a Fibonacci Prime.
Algorithm Used: Optimized Primality Test and Fibonacci Generation
Explanation: Combines primality testing with Fibonacci sequence checking. Learning Outcome: Implementation of efficient number property testing.

Program no - 21

```
def extended_gcd(a, b): if a == 0:
        return (b, 0, 1)
else:
        g, y, x = extended_gcd(b % a, a)
return (g, x - (b // a) * y, y)

def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m) if g != 1:
                    return None else:
                return x % m
```

Program No - 22

```
def extended_gcd(a, b): if a == 0:
        return (b, 0, 1)
else:
        g, y, x = extended_gcd(b % a, a)
return (g, x - (b // a) * y, y)

def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m) if g != 1:
                    return None
```

```
    return x % m

def crt(remainders, moduli):
        if len(remainders) != len(moduli): raise
ValueError("Input lists mismatch")
```

```
    N = 1            for m
in moduli:                    N
*= m

    total = 0            for i in
range(len(moduli)):                r_i=
remainders[i]                      m_i = moduli[i]
                                   N_i = N // m_i y_i =
mod_inverse(N_i, m_i)
if y_i is None:
                    raise ValueError("Moduli not coprime") total += r_i * N_i * y_i

    return total % N
```

Program No - 23

```
def power(base, exp, mod): res = 1
                           base %= mod
while exp > 0:                     if exp % 2 == 1:
                         res = (res * base) %
mod              exp >>= 1                 base=
(base * base) % mod              return res

def is_quadratic_residue(a, p): if a % p == 0:
              return True          if p == 2:
            return a % 2 == 1
exponent = (p - 1) // 2                  legendre
= power(a, exponent, p)                  return
legendre == 1
```

Program No - 24

```
def gcd(a, b):
while b:                   a, b = b,
a % b              return a

def order_mod(a, n):                 if n <=
1 or gcd(a, n) != 1:
            return None            k = 1 current = a
% n            while k <= n: if current == 1:
                                   return k
              current = (current * a) %
n             k += 1           return None

def is_prime(n):                 if n <= 1:
return False              if n <= 3:
return True              if n % 2 == 0 or n % 3 ==
0:
                return False              i = 5          while i
              * i <= n:                  if n % i == 0 or n % (i
+ 2) == 0:
                   return False
i += 6           return True
```

Program No - 25

```
def is_fibonacci(n):
if n < 0:
return False                    if n ==
0:
return True                    a, b
= 0, 1                 while b <
n:                     a, b = b,
a + b                 return b
== n

def is_fibonacci_prime(n):
        return is_prime(n) and is_fibonacci(n)
```

Overall learning outcomes

The completion of this assignment has provided a strong foundational understanding of key

Algorithmic principles in computer science and discrete mathematics, including:

1. Computational Number Theory: Practical implementation of algorithms like the

Extended Euclidean Algorithm, Chinese Remainder Theorem, and Euler's Criterion,

Which are cornerstones of modern cryptography and number-theoretic computing.

2. Algorithmic Efficiency: Gained experience in implementing highly efficient algorithms

Such as Exponentiation by Squaring and the optimized Primality Test, understanding

How to reduce complexity from linear to logarithmic time.

Practical No: 26

TITLE: Write a function Lucas Numbers

Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1).

AIM/OBJECTIVE(s): lucas_sequence

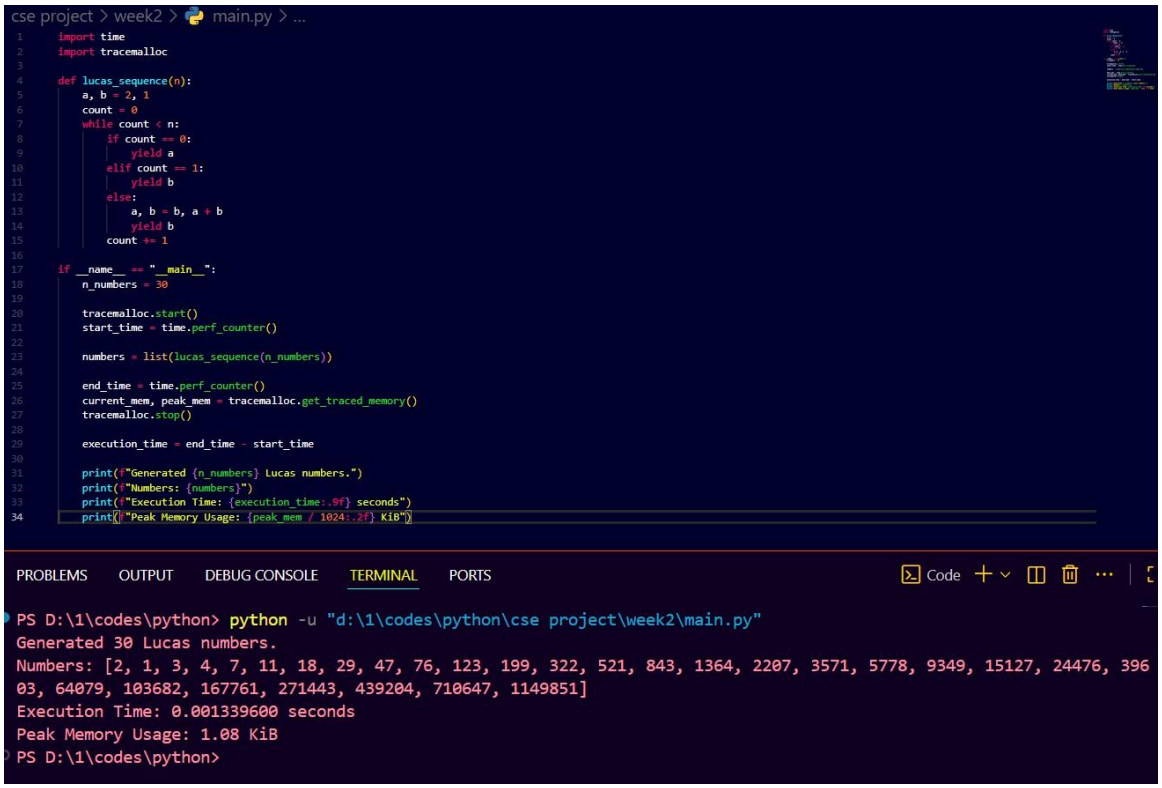
METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

A generator function (lucas_sequence) is implemented to yield Lucas numbers iteratively, which is memory-efficient. The main part of the script calls this generator, collects all generated numbers into a list, and measures the performance of this operation.

RESULTS ACHIEVED:



SKILLS ACHIEVED:

- Python programming fundamentals.
- Implementation of generator functions (using yield) for efficient data generation.
- Performance analysis:
  - Measuring code execution time using the timemodule.
  - Measuring peak memory usage using the tracemalloc module.
- Understanding and using the if _name__== "_main_": guard.
- Using formatted strings (f-strings) for clear and dynamic output.

Practical No: 27

TITLE: Perfect Power Check with Performance Measurement

AIM/OBJECTIVE(s):To write a Python function is_perfect_power(n) that determines if a given integer ncan be expressed as a^b, where $a > 0$ and $b > 1$. The script should also measure the execution time and peak memory usage of this check.

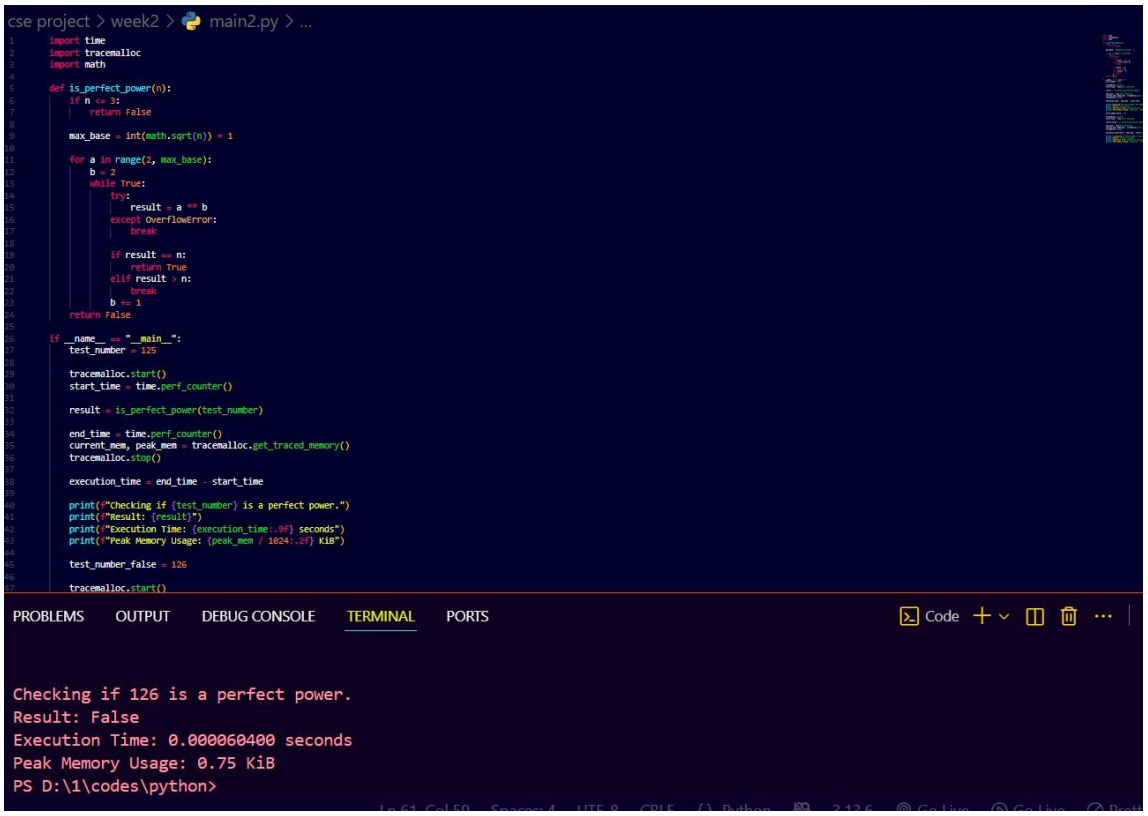
METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

is_perfect_power(n). It first handles edge cases where $n \le 3$. It then iterates through potential bases afrom 2 up to int(math.sqrt(n)) + 1. For each a, it enters a nested loop for exponent b(starting at 2). It calculates a^b in each step. If $a^b == n$, it returns True. If $a^b > n$, it breaks the inner loop and tries the next base a. If no such pair (a, b)is found after all checks, it returns False.

The main execution block measures the time and memory for checking two numbers (one perfect power like 125, and one non-perfect power like 126) and prints the results.

Result:



SKILLS ACHIEVED:

The script successfully checks if the test numbers are perfect powers and reports the performance.

Practical No: 28

TITLE:Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps for

n to reach 1 in the Collatz conjecture.

AIM/OBJECTIVE(s):To write a Python function collatz_length(n) that returns the number of steps required for a given integer nto reach 1 by following the rules of the Collatz conjecture .

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION:

The code defines a function collatz_length(n)that calculates the "stopping time" for the Collatz sequence starting at n

RESULTS ACHIEVED:

```python
import time
import tracemalloc

def collatz_length(n):
    if n <= 0:
        return 0

    length = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        length += 1
    return length

if __name__ == "__main__":
    test_number = 27

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

    result_length = collatz_length(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 Collatz sequence length for {test_number}.")
    print(f"Result (steps to reach 1): {result_length}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

    test_number_long = 837799

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

    result_length_long = collatz_length(test_number_long)

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

    execution_time_long = end_time - start_time
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Calculating Collatz sequence length for 837799.
Result (steps to reach 1): 524
Execution Time: 0.000765600 seconds
Peak Memory Usage: 0.12 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

Implementation of an iterative algorithm based on a mathematical conjecture.

Use of conditional logic (if/else) and loop structures (while). Integer

arithmetic (using %for modulus and //for integer division). Continued

practice in performance analysis with timeand tracemalloc.

Practical No: 29

TITLE:Write a function Polygonal Numbers polygonal_number(s, n)

that returns the n-th s-gonal number.

AIM/OBJECTIVE(s) : To write a Python function polygonal_number(s, n)that calculates and returns the n-th s-gonal (polygonal) number, given the number of sides sand the term n. The script should also measure the execution time and peak memory usage.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

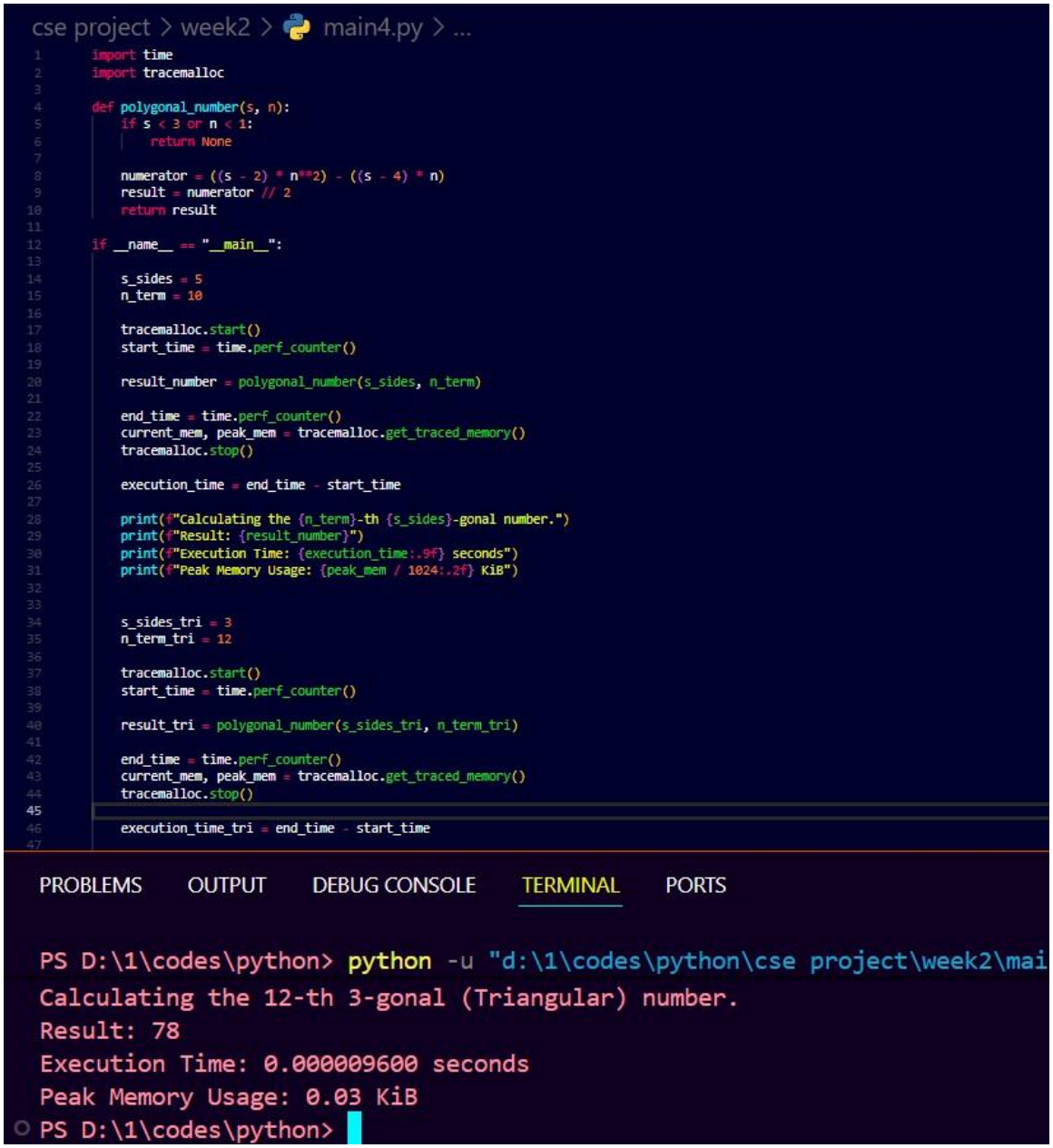
The code defines a function polygonal_number(s,n)that implements the mathematical formula for the n-th s-gonal number. It includes a check to ensure s(sides) is at least 3 and n(term) is at least 1.

The main execution block (if __name__== "__main__":) measures the time and memory for calculating two different polygonal numbers:

1. The 10th pentagonal (5-gonal) number.
2. The 12th triangular (3-gonal) number.

It prints the calculated number, the execution time, and the peak memory usage for each case.

Result:



SKILLS ACHIEVED :

- Translation of a mathematical formula into a working Python function.
- Handling function arguments and basic input validation (e.g., $s \ge 3$).
- Continued practice in performance measurement with timeand tracemalloc.
- Testing the function with known values (e.g., triangular, pentagonal numbers) to verify correctness

Practical No: 30

TITLE:Write a function Carmichael Number

Check is_carmichael(n) that checks if a composite number

n satisfies an−1 ≡ 1 mod n for all a coprime to n.

AIM/OBJECTIVE(s):

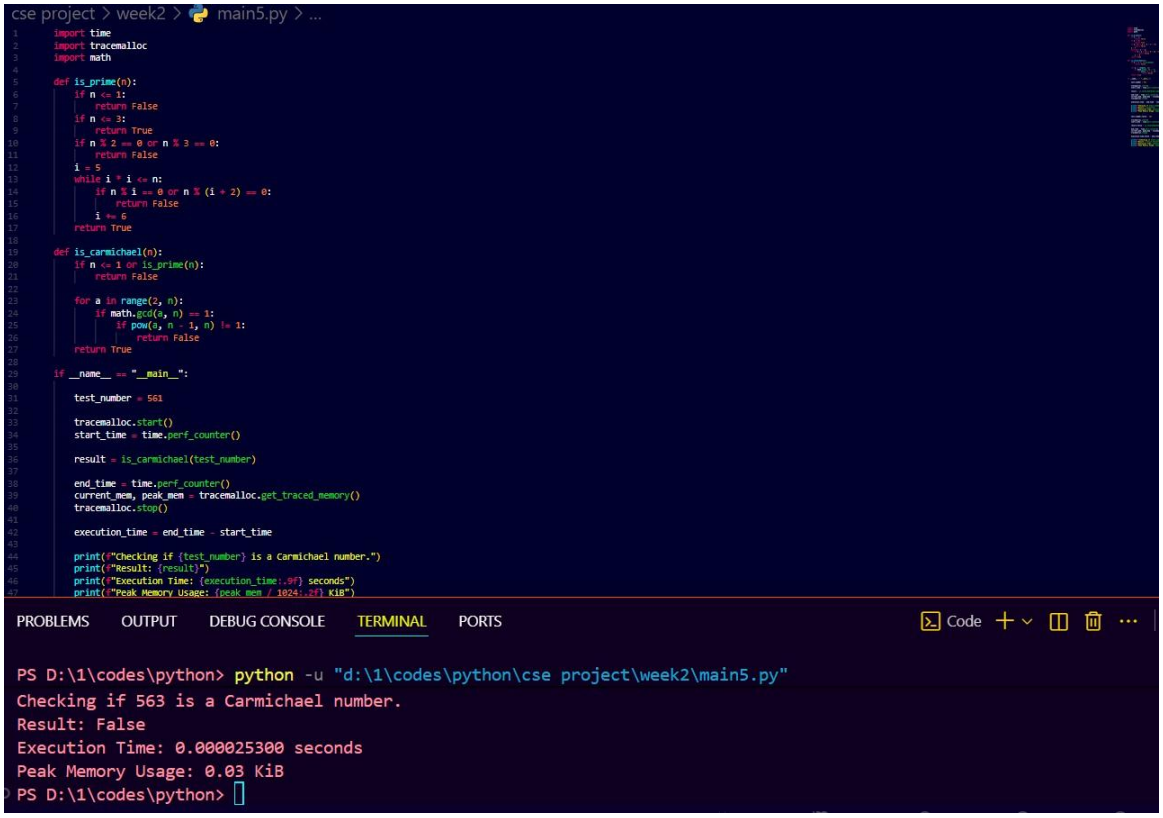
To write a Python function is_carmichael(n)that checks if a given composite number nis a Carmichael number. A number is a Carmichael number if it is composite and satisfies $a^{n-1} \equiv 1 \pmod n$ for all integers athat are coprime to n.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines two functions. is_prime(n)is a standard trial division primality test. is_carmichael(n) implements the definition of a Carmichael number. It first rules out prime numbers. Then, it iterates through all possible bases afrom 2 to $n-1$, finds those coprime to n, and tests if $a^{n-1} \pmod n$ is 1.

RESULT:



SKILLS ACHIEVED:

Implementation of number theory definitions (Primality, Coprimality, Fermat's Little Theorem).

- Use of helper functions (is_prime) to structure complex logic.
- Use of math.gcdfor coprimality testing.
- Use of pow(a,b,m)for efficient modular exponentiation.
- Understanding the computational cost of different algorithms.

Program 31

Program Assigned

Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.

Code

```
n = int(input("Enter number to test primality: "))
k = int(input("Enter number of rounds: "))

if n < 2:
    print("Composite")
    exit()
if n in (2, 3):
    print("Prime")
    exit()
if n % 2 == 0:
    print("Composite")
    exit()

d = n - 1
r = 0
while d % 2 == 0:
    d //= 2
    r += 1


bases = [2, 3, 5, 7, 11, 13, 17]
is_probably_prime = True
base_index = 0

for _ in range(k):
    a = bases[base_index]
    base_index += 1
    if a >= n - 1:
        break

    x = pow(a, d, n)
    if x == 1 or x == n - 1:
        continue

    composite_flag = True
    for _ in range(r - 1):
        x = (x * x) % n
        if x == n - 1:
            composite_flag = False
            break

    if composite_flag:
        is_probably_prime = False
        break

if is_probably_prime:
    print("Probably Prime")
else:
    print("Composite")
```

```
Enter number to test primality: 71
Enter number of rounds: 4
Probably Prime
>>>
```

- **Skills Acquired**

  - Understanding, implementing, and utilizing **probabilistic algorithms**.
  - Applying complex number theory theorems to computational problems.

Program 32

Program Assigned

Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.

Code

```
n = int(input("Enter number to factor: "))

if n % 2 == 0:
    print("Factor:", 2)
    exit()

x = 2
y = 2
c = 1
d = 1

while d == 1:
    x = (x * x + c) % n
    y = (y * y + c) % n
    y = (y * y + c) % n

    diff = x - y
    if diff < 0:
        diff = -diff

    a = diff
    b = n
    while b != 0:
        a, b = b, a % b

    d = a

if d == n:
    print("Failure factorizing")
else:
    print("Non-trivial factor:", d)
```

```
Enter number to factor: 100
Factor: 2
>>>
```

2.5 Skills Acquired

- Implementing a **probabilistic (Monte Carlo)** and **non-deterministic** algorithm.
- Applying graph theory algorithms (specifically, Floyd's Cycle-Finding Algorithm, or "Tortoise and Hare") to number theory problems.

Program 33

Program Assigned

Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.

Code

```
s = float(input("Enter s: "))
terms = int(input("Enter number of terms: "))

zeta = 0.0
n = 1

while n <= terms:
    zeta += 1 / (n ** s)
    n += 1

print("Zeta approximation =", zeta)
```

```
Enter s: 3.14
Enter number of terms: 3
Zeta approximation = 1.1451968791422344
>>>
```

Skills Acquired

- Confidently handling and performing calculations involving complex numbers.
- Implementing numerical methods for approximating infinite mathematical concepts.
- Implementing dynamic exponentiation where both the base and the exponent are variable (and the exponent may be complex).

Program 34

program assigned:

Write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers.

Code

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

# dp[i] = number of partitions of i
dp = [0] * (n + 1)
dp[0] = 1

num = 1
while num <= n:
    i = num
    while i <= n:
        dp[i] += dp[i - num]
        i += 1
    num += 1

print("Number of partitions p(n):", dp[n])
```

```
Enter n: 10
Number of partitions p(n): 42
>>>
```

4.1 Skills acquired

- Understanding and applying the Dynamic Programming paradigm to solve combinatorial problems.
- Translating a complex mathematical recurrence relation into nested loop structures.

1. **Overall Skills Acquired & Conclusion**
    - **Algorithmic Efficiency and Complexity Management**
    - **Applied Number Theory and Modular Arithmetic**
    - **Numerical & Complex Analysis**