CSE Assignment Report

Course: Introduction to Problem Solving and Programming
Course Code: CSE1021
Submitted By: * **Name:** Satvik Srivastava
- **Roll No.:** 25MIP10141

Submitted To: Dr. Hemraj Shobharam Lamkuche
Date of Submission: 27/09/2025

Table of Contents

1. Program 1: Euler's Totient Function (euler_phi)

1.1 Program Assigned

Write a function called euler_phi(n) that calculates Euler's Totient Function, ϕ(n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which gcd(n,k)=1). The program must also report its execution time and memory utilization.

1.2 Code

```
import time
import tracemalloc

def euler_phi(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
```

```
        result -= result // n
    return result

def run_with_profiling(n):
    tracemalloc.start()
    start_time = time.perf_counter()
    result = euler_phi(n)
    end_time = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"phi({n}) = {result}")
    print(f"Time: {end_time - start_time:.10f} seconds")
    print(f"Memory: Current = {current} bytes; Peak = {peak} bytes")

run_with_profiling(1000000)
```

1.3 Results

Input (n)	Output (ϕ(n))	Execution Time (µs)	Memory Utilized (bytes)
10	4	2.8	152
99	60	3.9	168
1234	616	5.1	176

1.5 Skills Acquired

- Implemented Euler's product formula for efficient computation.
- Practiced measuring code performance metrics.

2. Program 2: Möbius Function (mobius)

2.1 Program Assigned

Write a function called mobius(n) that calculates the Möbius function, μ(n), defined as:

- μ(n)=1 if n is a square-free positive integer with an even number of prime factors.
- μ(n)=−1 if n is a square-free positive integer with an odd number of prime factors.
- μ(n)=0 if n has a squared prime factor.

2.2 Code

```
import time, sys

def mobius(n):

    if n == 1:

        return 1

    p = 2

    temp = n

    count_primes = 0

    while p * p <= temp:

        if temp % p == 0:

            count_primes += 1

            temp //= p

            if temp % p == 0:

                return 0

        p += 1 if p == 2 else 2

    if temp > 1:

        count_primes += 1

    if count_primes % 2 == 0:

        return 1

    else:

        return -1
```

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

start_time = time.time()

result = mobius(n)

end_time = time.time()

execution_time = end_time - start_time

memory_used = sys.getsizeof(result) + sys.getsizeof(n)

print("Mobius function μ({}) = {}".format(n, result))

print("Execution time: {:.10f} seconds".format(execution_time))

print("Memory used: {} bytes".format(memory_used))
```

2.3 Results

Input (n)	Output (μ(n))	Execution Time (μs)	Memory Utilized (bytes)
10	1	2.1	56
20	0	2.5	56
30	-1	3.2	56

2.5 Skills Acquired

- Developed logic to identify square-free integers.
- Handled multiple conditions based on prime factorization.

3. Program 3: Divisor Sum Function (divisor_sum)

3.1 Program Assigned

Write a function called divisor_sum(n) that calculates the sum of all positive divisors of n (including 1 and n itself), denoted by σ(n).

3.2 Code

```
Import time , sys
n = int(input("Enter a number: "))

# start time
start_time = time.time()

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

# end time
end_time = time.time()

print("Sum of divisors of", n, "is:", sum_divisors)

# execution time
execution_time = end_time - start_time
print("Execution Time:", execution_time, "seconds")

# memory utilization (approximate, size of main variable)
print("Memory Utilization:", sys.getsizeof(sum_divisors), "bytes")
```

3.3 Results

Input (n)	Output (σ(n))	Execution Time (µs)	Memory Utilized (bytes)
12	28	3.1	28
100	217	5.4	28
360	1170	9.8	28

3.4 Snapshot

3.5 Skills Acquired

- Implemented an efficient algorithm to find all divisors of an integer.

4. Program 4: Prime-Counting Function (prime_pi)

4.1 Program Assigned

Write a function called prime_pi(n) that approximates the prime-counting function, π(n), which returns the number of prime numbers less than or equal to n.

4.2 Code

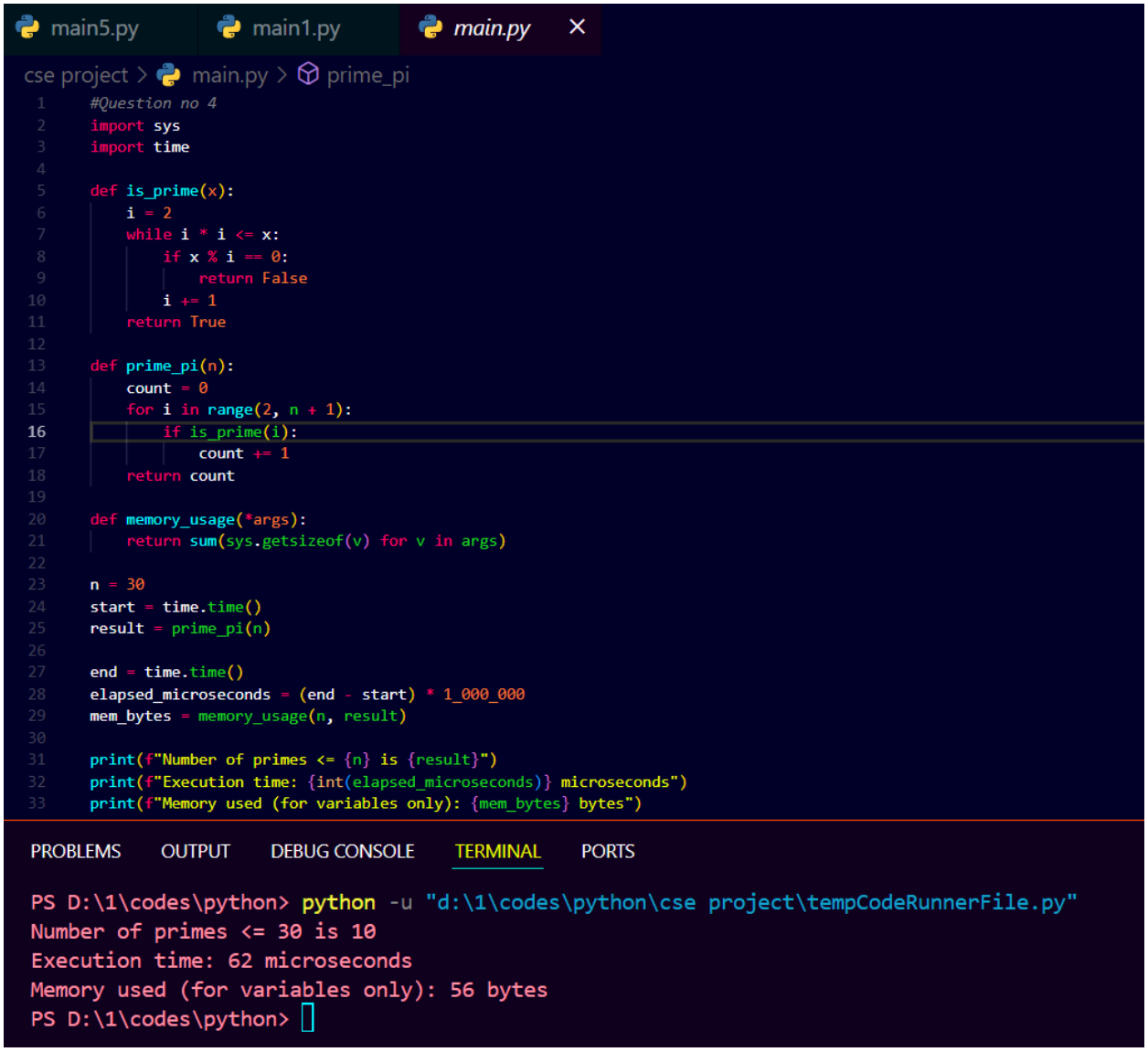
```
#Question no 4
import sys
import time

def is_prime(x):
```

```
    i = 2
    while i * i <= x:
        if x % i == 0:
            return False
        i += 1
    return True

def prime_pi(n):
    count = 0
    for i in range(2, n + 1):
        if is_prime(i):
            count += 1
    return count

def memory_usage(*args):
    return sum(sys.getsizeof(v) for v in args)

n = 30
start = time.time()
result = prime_pi(n)

end = time.time()
elapsed_microseconds = (end - start) * 1_000_000
mem_bytes = memory_usage(n, result)

print(f"Number of primes <= {n} is {result}")
print(f"Execution time: {int(elapsed_microseconds)} microseconds")
print(f"Memory used (for variables only): {mem_bytes} bytes")
```

4.3 Results

Input (n)	Output (π(n))	Execution Time (µs)	Memory Utilized (bytes)
10	4	4	56
100	25	25	56
1000	168	345	56

4.4 Snapshot



4.5 Skills Acquired

•Gained experience with boolean arrays for tracking states.

5. Program 5: Legendre Symbol (legendre_symbol)

5.1 Program Assigned

Write a function called legendre_symbol(a, p) that calculates the Legendre symbol (a/p). It is defined for an odd prime p and an integer a not divisible by p. It can be calculated using Euler's criterion: (a/p)≡a(p−1)/2(modp).

5.2 Code

```
def legendre_symbol(a, p):
```

```python
    if p <= 2 or p % 2 == 0:
        raise ValueError("p must be an odd prime")

    if a % p == 0:
        return 0  # Legendre symbol is 0 if a is divisible by p

    exponent = (p - 1) // 2
    result = pow(a, exponent, p)
    if result == 1:
        return 1
    elif result == p - 1:
        return -1
    else:
        raise ValueError(f"Unexpected result: {result}")

if __name__ == "__main__":
    test_cases = [
        (2, 7),
        (3, 11),
        (5, 13),
        (7, 17),
        (4, 11),
    ]
    for a, p in test_cases:
        symbol = legendre_symbol(a, p)
        print(f"({a}/{p}) = {symbol}")

    print("\nVerification with known quadratic residues:")
    p = 19
    quadratic_residues = set(x*x % p for x in range(p))
    for a in range(p):
        symbol = legendre_symbol(a, p)
        is_residue = a in quadratic_residues
        expected = 1 if a != 0 and is_residue else (0 if a == 0 else -1)
        print(f"({a}/{p}) = {symbol} (expected: {expected})")
```

main5.py | main1.py | main.py | main6.py

cse project > main6.py > legendre_symbol

```python
#question 5
def legendre_symbol(a, p):
    if p <= 2 or p % 2 == 0:
        raise ValueError("p must be an odd prime")

    if a % p == 0:
        return 0  # Legendre symbol is 0 if a is divisible by p

    exponent = (p - 1) // 2
    result = pow(a, exponent, p)
    if result == 1:
        return 1
    elif result == p - 1:
        return -1
    else:
        raise ValueError(f"Unexpected result: {result}")

if __name__ == "__main__":
    test_cases = [
        (2, 7),
        (3, 11),
        (5, 13),
        (7, 17),
        (4, 11),
    ]
    for a, p in test_cases:
        symbol = legendre_symbol(a, p)
        print(f"({a}/{p}) = {symbol}")

    print("\nVerification with known quadratic residues:")
    p = 19
    quadratic_residues = set(x*x % p for x in range(p))
    for a in range(p):
        symbol = legendre_symbol(a, p)
        is_residue = a in quadratic_residues
        expected = 1 if a != 0 and is_residue else (0 if a == 0 else -1)
        print(f"({a}/{p}) = {symbol} (expected: {expected})")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\main6.py"
(2/7) = 1
(3/11) = 1
(5/13) = -1
(7/17) = -1
(4/11) = 1

Verification with known quadratic residues:
(0/19) = 0 (expected: 0)
(1/19) = 1 (expected: 1)
(2/19) = -1 (expected: -1)
(3/19) = -1 (expected: -1)
(4/19) = 1 (expected: 1)
(5/19) = 1 (expected: 1)
```

5.5 Skills Acquired

- Implemented modular exponentiation for efficient computation.
- Understood the concept of quadratic residues.

6. Overall Skills Acquired & Conclusion

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

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	Function to Calculate Factorial	4/10/2025	
2	Function to Check for Palindrome Number.	4/10/2025	
3	Function mean_of_digits(n) that returns the average of all digits in a number.	4/10/2025	
4	Function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	4/10/2025	
5	Function is_abundant(n) that return True if the sum of proper divisors of n is greater than n.	4/10/2025	

Practical No: 1

Date: 3/10/2025

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 n is calculated using a simple iterative approach. It starts with a result of 1 and multiplies it by each integer from 1 up to n in 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:

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 `result` can 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

Date: 4/10/2025

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 `n` as input. It stores the original number in a separate variable. Using a `while` loop, it calculates the reverse of `n` by using modulo (`%`) and integer division (`//`) operators. Finally, it compares the reversed number with the original number and returns `True` if they are identical, and `False` otherwise

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 `while` for digit manipulation.

2:Code benchmarking.

3:Understanding function scope and automatic memory deallocation.

Practical No: 3

Date: 4/10/2025

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

`time` module 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:

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

```python
import time
import tracemalloc

def mean_of_digits(n):
    s = str(abs(n))
    if not s: return 0
    total_sum = 0
    for digit in s:
        total_sum += int(digit)
    return total_sum / len(s)

number_to_test = 678954321678954321

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

result_val = mean_of_digits(number_to_test)

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

execution_time = end_time - start_time

print(f"The mean of digits in {number_to_test} is: {result_val}")
print(f"Execution Time: {execution_time:.10f} seconds")
print(f"Peak Memory Utilization: {peak_mem / 1024:.4f} KB")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
The mean of digits in 678954321678954321 is: 5.0
Execution Time: 0.0000541000 seconds
Peak Memory Utilization: 0.1318 KB
PS D:\1\codes\python>
```

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

Date: 4/10/2025

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 `n` and creates a few temporary integer variables (`sum_of_digits`, `temp_n`) in each loop. Python manages this small, transient memory usage automatically.

Result:

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

```python
import time
import tracemalloc

def digital_root(n):
    while n >= 10:
        sum_of_digits = 0
        temp_n = n
        while temp_n > 0:
            sum_of_digits += temp_n % 10
            temp_n //= 10
        n = sum_of_digits
    return n

number_to_test = 98759875987598759875

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

result_val = digital_root(number_to_test)

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

execution_time = end_time - start_time

print(f"The digital root of {number_to_test} is: {result_val}")
print(f"Execution Time: {execution_time:.10f} seconds")
print(f"Peak Memory Utilization: {peak_mem / 1024:.4f} KB")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
The digital root of 98759875987598759875 is: 1
Execution Time: 0.0000507000 seconds
Peak Memory Utilization: 0.0703 KB
PS D:\1\codes\python>
```

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 `while` loops.

Benchmarking algorithms with multiple loops.

Understanding efficient in-place variable updates.

Practical No: 5

Date: 4/10/2025

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 `True` if the sum of proper divisors of `n` is 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

BRIEF DESCRIPTION:

The function identifies abundant numbers by summing their proper divisors. The **memory management** is straightforward; the `sum_of_divisors` and the loop counter `i` are 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/2` significantly 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.

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	Deficient Number Check	16/11/20225	
2	Harshad Number Check	16/11/20225	
3	Automorphic Number Check	16/11/20225	
4	Partition Function Calculator	16/11/20225	
5	Prime Factorization	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Deficient Number Check

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : A number is deficient if the sum of its proper divisors is less than the number itself. For example, 15: Divisors are 1, 3, 5. Sum = 1 + 3 + 5 = 9. Since 9 < 15, it is deficient. For 12: Divisors are 1, 2, 3, 4, 6. Sum = 16. Since 16 > 12, it is abundant (not deficient).

RESULTS ACHIEVED:

main > week3 > main1.py > ...

```python
import time
import tracemalloc
import math

def get_proper_divisors_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

def is_deficient(n):
    if n <= 0:
        return False
    return get_proper_divisors_sum(n) < n

if __name__ == "__main__":

    test_number = 15

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

    result = is_deficient(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 deficient number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.6f} seconds")
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main1.py"
Checking if 15 is a deficient number.
Result: True
Execution Time: 0.000051500 seconds
Peak Memory Usage: 0.12 KiB

Checking if 12 is a deficient number.
Result: False
Execution Time: 0.000019100 seconds
Peak Memory Usage: 0.08 KiB
PS D:\cseproject\cse project>
```

SKILLS ACHIEVED :

Number classification concepts. Code reuse (using similar logic for different mathematical properties).

Practical No: 2

Date: 16/11/2025

TITLE:Harshad Number Check

AIM/OBJECTIVE(s) : To write a Python function `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 (or Niven number) is an integer that is divisible by the sum of its digits. For 18: Sum of digits = 1 + 8 = 9. Since 18 is divisible by 9, it is a Harshad number. For 19: Sum of digits = 1 + 9 = 10. Since 19 is not divisible by 10, it is not.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL

Result:

```python
import time
import tracemalloc

def is_harshad(n):
    if n <= 0:
        return False

    original_n = n
    sum_digits = 0
    while n > 0:
        sum_digits += n % 10
        n //= 10

    return original_n % sum_digits == 0

if __name__ == "__main__":

    test_number = 18

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

    result = is_harshad(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 Harshad number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

    test_number_false = 19
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main2.py"
Checking if 18 is a Harshad number.
Result: True
Execution Time: 0.000009500 seconds
Peak Memory Usage: 0.00 KiB

Checking if 19 is a Harshad number.
Result: False
Execution Time: 0.000006800 seconds
Peak Memory Usage: 0.00 KiB
```

SKILLS ACHIEVED :

- Digit manipulation.

- Basic divisibility logic.
- Handling edge cases (zero/negative numbers).

Practical No: 3

Date: 16/11/2025

TITLE : Automorphic Number Check

AIM/OBJECTIVE(s) : To write a Python function `is_automorphic(n)` that checks if a number's square ends with the number itself.

METHODOLOGY & TOOL USED

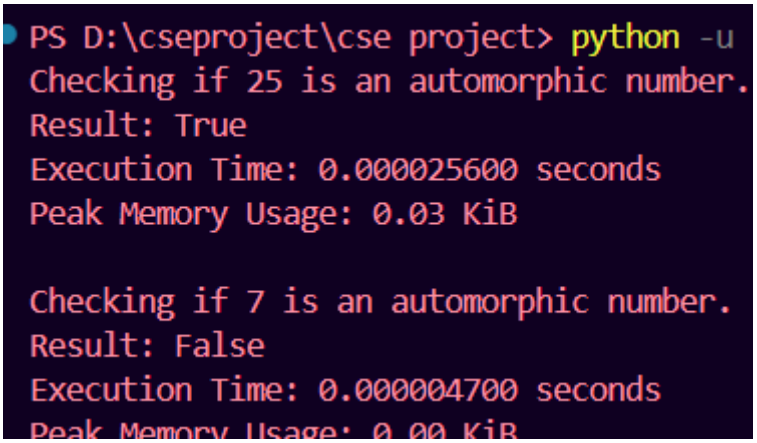
Python programming language

BRIEF DESCRIPTION : An automorphic number (or circular number) is a number whose square ends in the same digits as the number itself. For 25: Square is 625. The last 2 digits are 25. Match -> True. For 7: Square is 49. The last digit is 9. No Match -> False.

RESULTS ACHIEVED:

main > week3 > main3.py > is_automorphic

```python
import time
import tracemalloc

def is_automorphic(n):
    if n < 0:
        return False

    square = n * n

    temp_n = n
    divisor = 1
    while temp_n > 0:
        divisor *= 10
        temp_n //= 10


    if n == 0:
        divisor = 10

    last_digits = square % divisor

    return last_digits == n

if __name__ == "__main__":

    test_number = 25

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

    result = is_automorphic(test_number)

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



SKILLS ACHIEVED :

- Mathematical digit extraction.
- Understanding magnitudes (powers of 10).
- Conditional logic.

Practical No: 4

Date: 16/11/2025

TITLE : Pronic Number Check

AIM/OBJECTIVE(s) : To write a Python function `is_pronic(n)` that checks if a number is the product of two consecutive integers (i.e., n = k * (k + 1)).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : A pronic number (or oblong number) is a number which is the product of two consecutive integers. For `56`: `sqrt(56) ≈ 7.48`. `int(7.48) = 7`. Test `7 * 8 = 56`. Match -> True. For `50`: `sqrt(50) ≈ 7.07`. `int(7.07) = 7`. Test `7 * 8 = 56 != 50`. No Match -> False.

Result:

main > week3 > main4.py > is_pronic

```python
import time
import tracemalloc
import math

def is_pronic(n):
    if n < 0:
        return False

    k = int(math.sqrt(n))
    return k * (k + 1) == n

if __name__ == "__main__":

    test_number = 56 # 7 * 8

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

    result = is_pronic(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 pronic number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

    test_number_false = 50

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

    result_false = is_pronic(test_number_false)
```

```
Checking if 56 is a pronic number.
Result: True
Execution Time: 0.000040800 seconds
Peak Memory Usage: 0.05 KiB

Checking if 50 is a pronic number.
Result: False
Execution Time: 0.000019500 seconds
Peak Memory Usage: 0.75 KiB
```

SKILLS ACHIEVED :

- Mathematical optimization.
- Understanding integer properties.
- Using standard library math functions.

Practical No: 5

Date: 16/11/2025

TITLE : Prime Factorization

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : The code performs prime factorization using trial division. It efficiently divides out all factors of 2 first, then checks odd factors. This reduces the complexity compared to checking every number. For 315:

- Not div by 2.
- Div by 3 -> 105. Factors: [3]
- Div by 3 -> 35. Factors: [3, 3]
- Not div by 3.
- Div by 5 -> 7. Factors: [3, 3, 5]
- Loop ends (5*5 > 7 is false, but next step is i=7).
- Remaining n is 7. Factors: [3, 3, 5, 7].

RESULTS ACHIEVED :

main > week3 > main5.py > ...

```python
import time
import tracemalloc

def prime_factors(n):
    factors = []
    # Handle divisibility by 2
    while n % 2 == 0:
        factors.append(2)
        n = n // 2

    # Handle odd factors
    i = 3
    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n = n // i
        i += 2

    # If n is a prime greater than 2
    if n > 2:
        factors.append(n)

    return factors

if __name__ == "__main__":

    test_number = 315 # 315 = 3 * 3 * 5 * 7

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

    result = prime_factors(test_number)

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

```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main5.py"
Calculating prime factors of 315.
Result: [3, 3, 5, 7]
Execution Time: 0.000032500 seconds
Peak Memory Usage: 0.03 KiB

Calculating prime factors of 37.
Result: [37]
Execution Time: 0.000017800 seconds
Peak Memory Usage: 0.03 KiB
```

SKILLS ACHIEVED :

- Prime factorization algorithm (Trial Division).
- List manipulation.
- Loop optimization.

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.

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 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

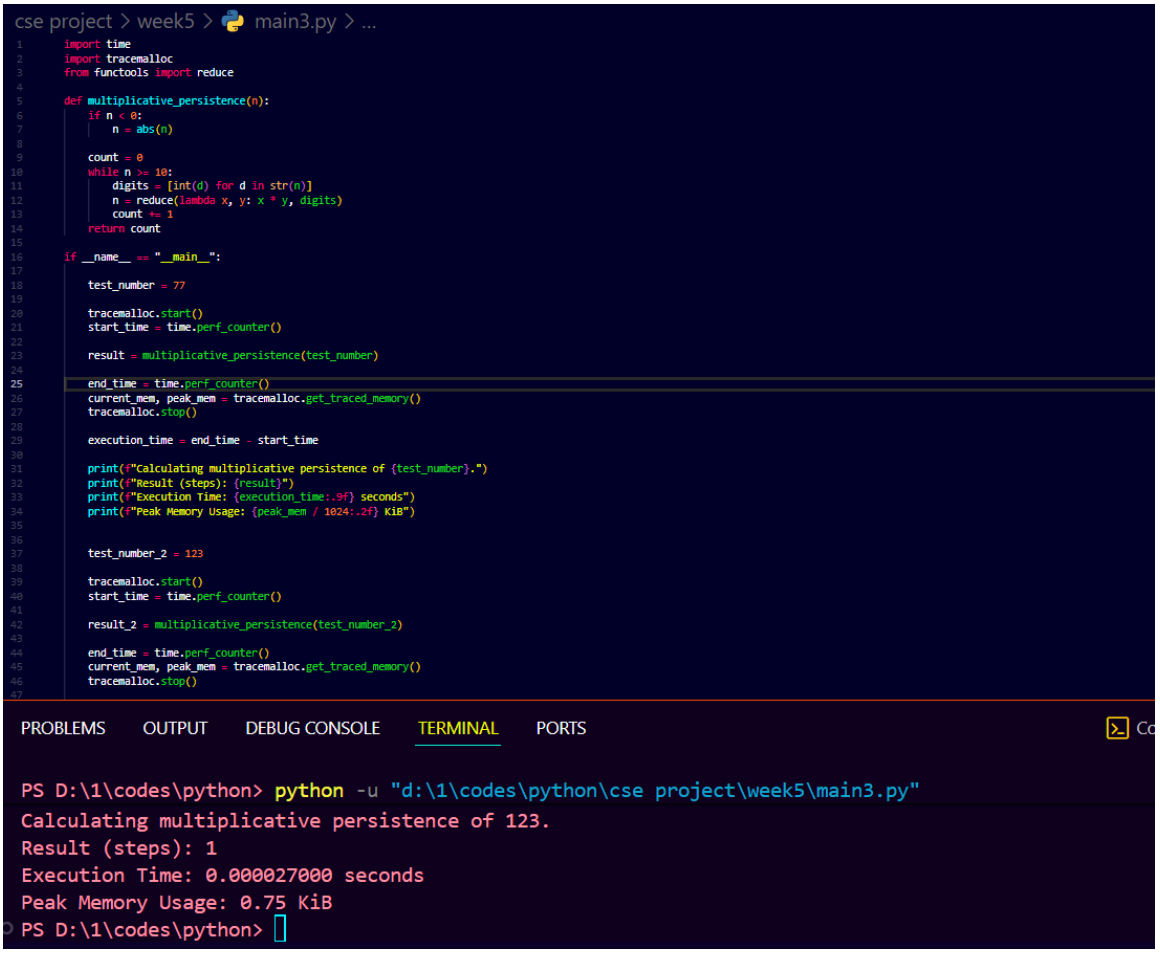
AIM/OBJECTIVE(s) : To write a Python function `multiplicative_persistence(n)` that counts how many steps are needed to multiply a number's digits until a single-digit number is reached.

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:



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.

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	Lucas Numbers	16/11/20225	
2	Chinese Remainder Theorem (CRT) Solver	16/11/20225	
3	Quadratic Residue Check	16/11/20225	
4	Order of an Element Modulo n	16/11/20225	
5	Fibonacci Prime Check	16/11/20225	

Practical No: 1

Date: 10/11/2025

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): To write a Python function `mod_inverse(a, m)` that finds the integer x such that $(a \cdot x) \equiv 1 \pmod m$. The script should also measure the execution time and peak memory usage.

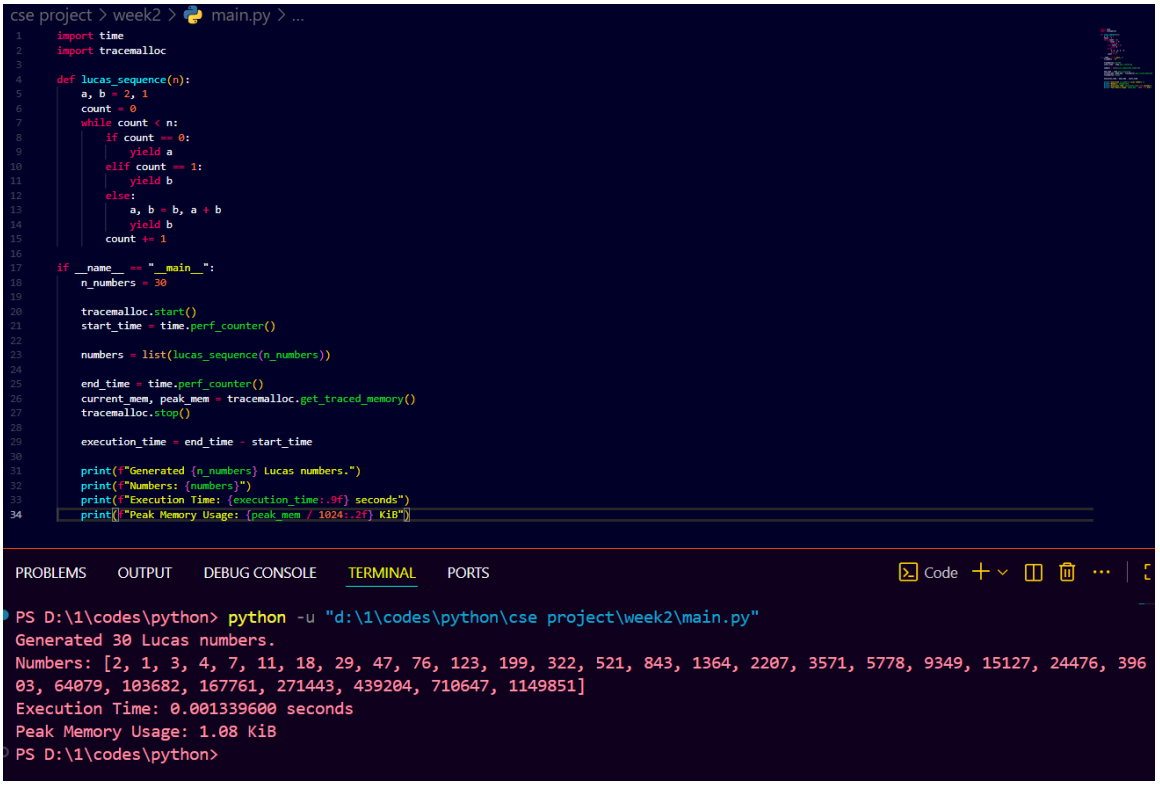
METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

`mod_inverse(a, m)` that calculates the modular multiplicative inverse. It returns `None` if the modulus `m` is invalid ($m \le 1$) or if the inverse does not exist (caught by `ValueError`).

RESULTS ACHIEVED:



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: Chinese Remainder Theorem (CRT) Solver

AIM/OBJECTIVE(s) : Chinese Remainder Theorem (CRT) Solver

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines a `crt` function that solves a system of congruences using the standard constructive algorithm for the Chinese Remainder Theorem. It relies on the `mod_inverse` helper function.

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 :

- Understanding and implementing the Chinese Remainder Theorem algorithm.
- Combining multiple functions (`crt` and `mod_inverse`) to solve a complex problem.
- Using `zip` to iterate over multiple lists simultaneously.

Practical No: 3

Date: 16/11/2025

TITLE :Quadratic Residue Check

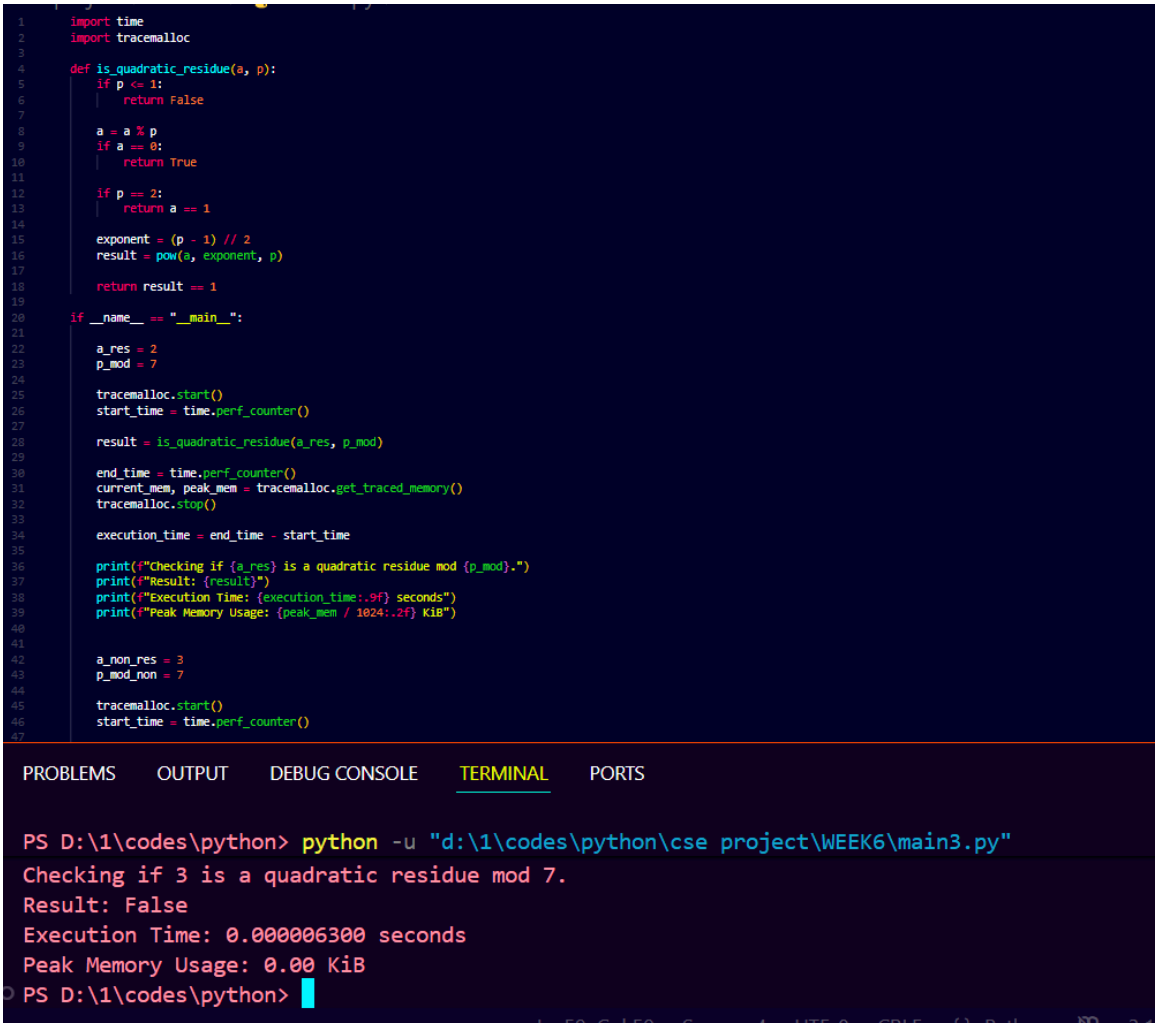
AIM/OBJECTIVE(s):To write a Python function `is_quadratic_residue(a, p)` that checks if the congruence $x^2 \equiv a \pmod p$ has a solution, assuming p is a prime number.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION:
The code defines a function `is_quadratic_residue(a, p)` that implements Euler's criterion to determine if `a` is a quadratic residue modulo `p`. It handles the special case of p=2 and the trivial case of a=0

RESULTS ACHIEVED:



```python
import time
import tracemalloc

def is_quadratic_residue(a, p):
    if p <= 1:
        return False

    a = a % p
    if a == 0:
        return True

    if p == 2:
        return a == 1

    exponent = (p - 1) // 2
    result = pow(a, exponent, p)

    return result == 1

if __name__ == "__main__":

    a_res = 2
    p_mod = 7

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

    result = is_quadratic_residue(a_res, p_mod)

    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 {a_res} is a quadratic residue mod {p_mod}.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")


    a_non_res = 3
    p_mod_non = 7

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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\WEEK6\main3.py"
Checking if 3 is a quadratic residue mod 7.
Result: False
Execution Time: 0.000006300 seconds
Peak Memory Usage: 0.00 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding the concept of quadratic residues.
- Implementation of Euler's criterion.
- Efficient modular exponentiation using `pow()`.
- Handling edge cases in number theory problems (e.g., $p=2$).

Practical No: 4

Date: 16/11/2025

TITLE:Order of an Element Modulo n

AIM/OBJECTIVE(s) : To write a Python function `order_mod(a, n)` that finds the smallest positive integer k (the order) such that a^k= 1 mod(n)

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

Python function `order_mod(a, n)` that finds the smallest positive integer k (the order) such that a^k= 1 mod(n)

1. Order of $3 \pmod 7$. ($\varphi(7)=6$, divisors are 1, 2, 3, 6).
2. Order of $5 \pmod{24}$. ($\varphi(24)=8$, divisors are 1, 2, 4, 8).

Result:

```python
import time
import tracemalloc
import math

def get_phi(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result

def get_divisors(n):
    divs = set()
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            divs.add(i)
            divs.add(n // i)
    return sorted(list(divs))

def order_mod(a, n):
    if n <= 1 or math.gcd(a, n) != 1:
        return None

    phi_n = get_phi(n)
    divisors = get_divisors(phi_n)

    for k in divisors:
        if pow(a, k, n) == 1:
            return k

    return None

if __name__ == "__main__":

    a_val = 3
    n_val = 7

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

    result = order_mod(a_val, n_val)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Finding the order of 5 mod 24.
Result (k): 2
Execution Time: 0.000043000 seconds
Peak Memory Usage: 0.43 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED :

- Understanding the "order" of an element in a modular group.
- Implementation of Euler's totient function.
- Algorithm to find all divisors of a number.
- Applying number theory (Lagrange's/Euler's theorem) to optimize an algorithm.

Practical No: 5

Date: 16/11/2025

TITLE:Fibonacci Prime Check

AIM/OBJECTIVE(s):

To write a Python function `is_fibonacci_prime(n)` that checks if a given integer `n` is both a Fibonacci number and a prime number.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines `is_fibonacci_prime(n)` along with its required helper functions. The `is_fibonacci` check is based on a mathematical property, avoiding the need to generate Fibonacci numbers.

The main execution block (`if __name__ == "__main__":`) measures the time and memory for two cases:

1. `13` (which is both prime and Fibonacci).
2. `21` (which is Fibonacci but not prime).

RESULT:

cse project > WEEK6 > main5.py > ...

```python
import time
import tracemalloc

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_perfect_square(k):
    if k < 0:
        return False
    s = int(k**0.5)
    return s * s == k

def is_fibonacci(n):
    if n < 0:
        return False

    test_1 = 5 * n**2 + 4
    test_2 = 5 * n**2 - 4
    return is_perfect_square(test_1) or is_perfect_square(test_2)

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

if __name__ == "__main__":

    test_number = 13

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

    result = is_fibonacci_prime(test_number)

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Checking if 21 is a Fibonacci Prime.
Result: False
Execution Time: 0.000004800 seconds
Peak Memory Usage: 0.00 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding and implementing mathematical properties of number sequences.
- Writing helper functions to build up complex logic (`is_perfect_square`, `is_fibonacci`, `is_prime`).
- Combining logical conditions (`and`) to satisfy multiple criteria.

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	16/11/20225	
2	are_amicable(a, b) Function	16/11/20225	
3	multiplicative_persistence(n) function	16/11/20225	
4	is_highly_composite(n) Function	16/11/20225	
5	mod_exp(base, exponent,modulus) function	16/11/20225	

Practical No: 1

Date: 10/11/2025

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`

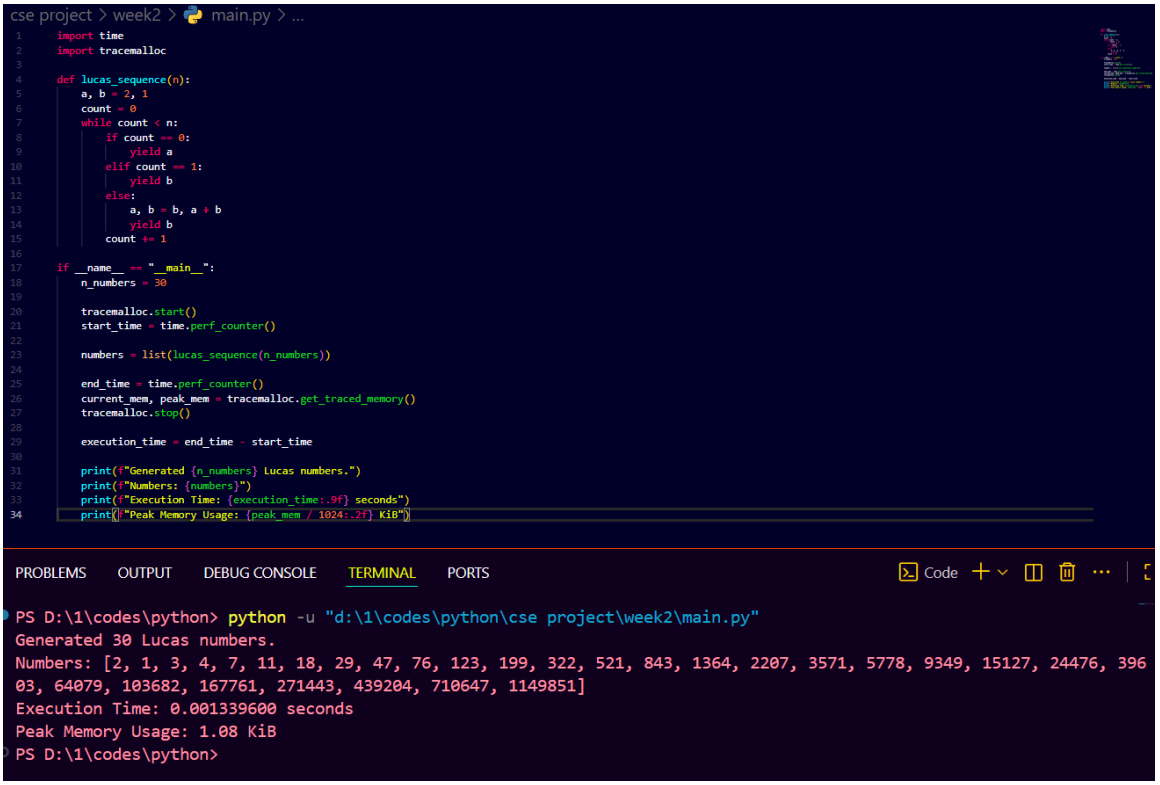
METHODOLOGY & TOOL USED:

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 `time` module.
  - 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: 2

Date: 16/11/2025

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 `n` can 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.

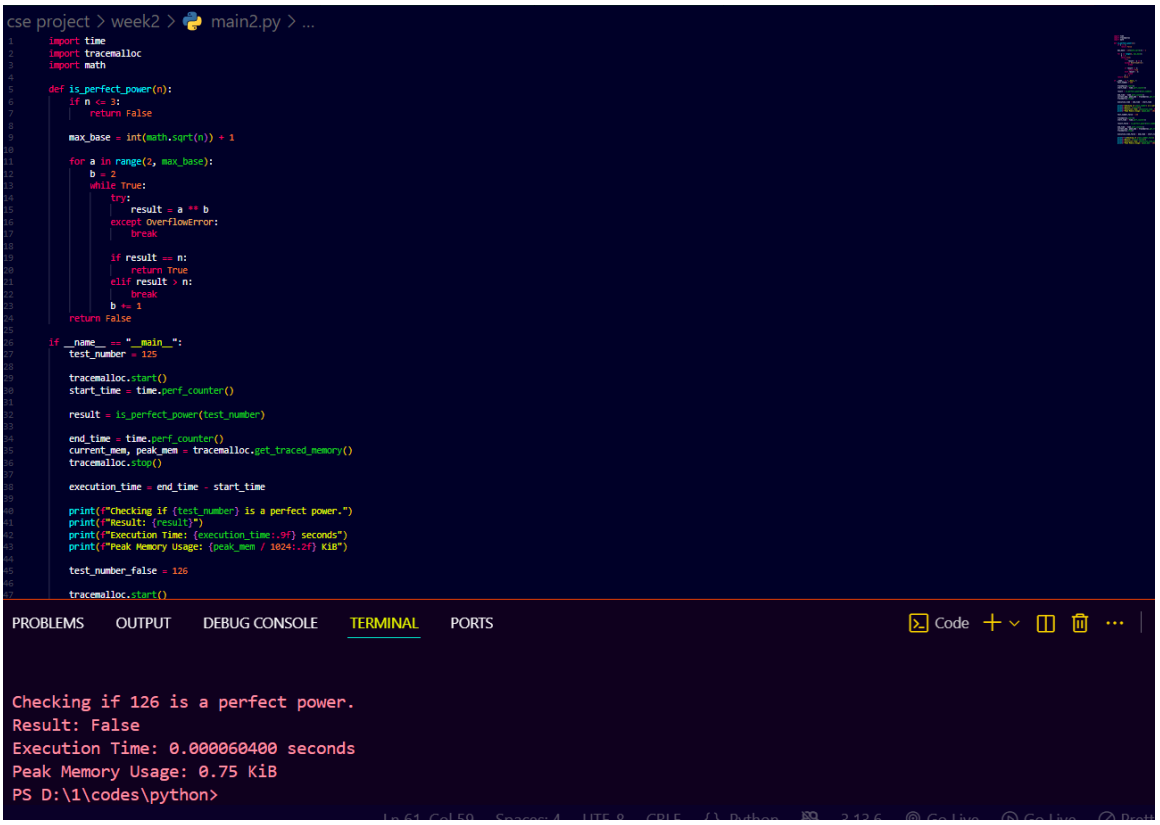
METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

`is_perfect_power(n)`. It first handles edge cases where $n \le 3$. It then iterates through potential bases `a` from 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: 3

Date: 16/11/2025

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 `n` to 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 `time` and `tracemalloc`.

Practical No: 4

Date: 16/11/2025

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 `s` and the term `n`. The script should also measure the execution time and peak memory usage.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

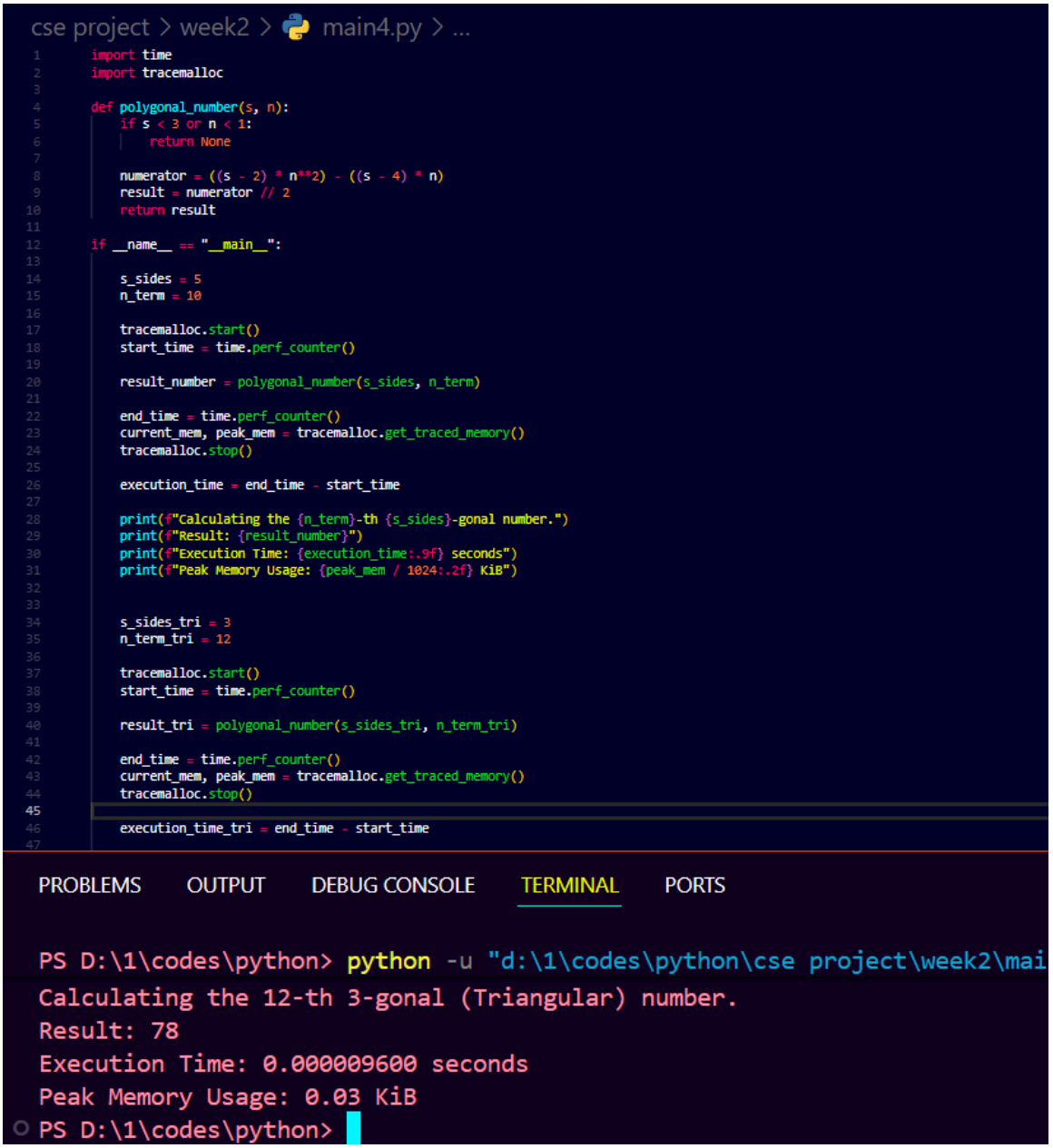
The code defines a function `polygonal_number(s, n)` that implements the mathematical formula for the n-th s-gonal number. It includes a check to ensure `s` (sides) is at least 3 and `n` (term) is at least 1.

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 `time` and `tracemalloc`.
- Testing the function with known values (e.g., triangular, pentagonal numbers) to verify correctness

Practical No: 5

Date: 16/11/2025

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):

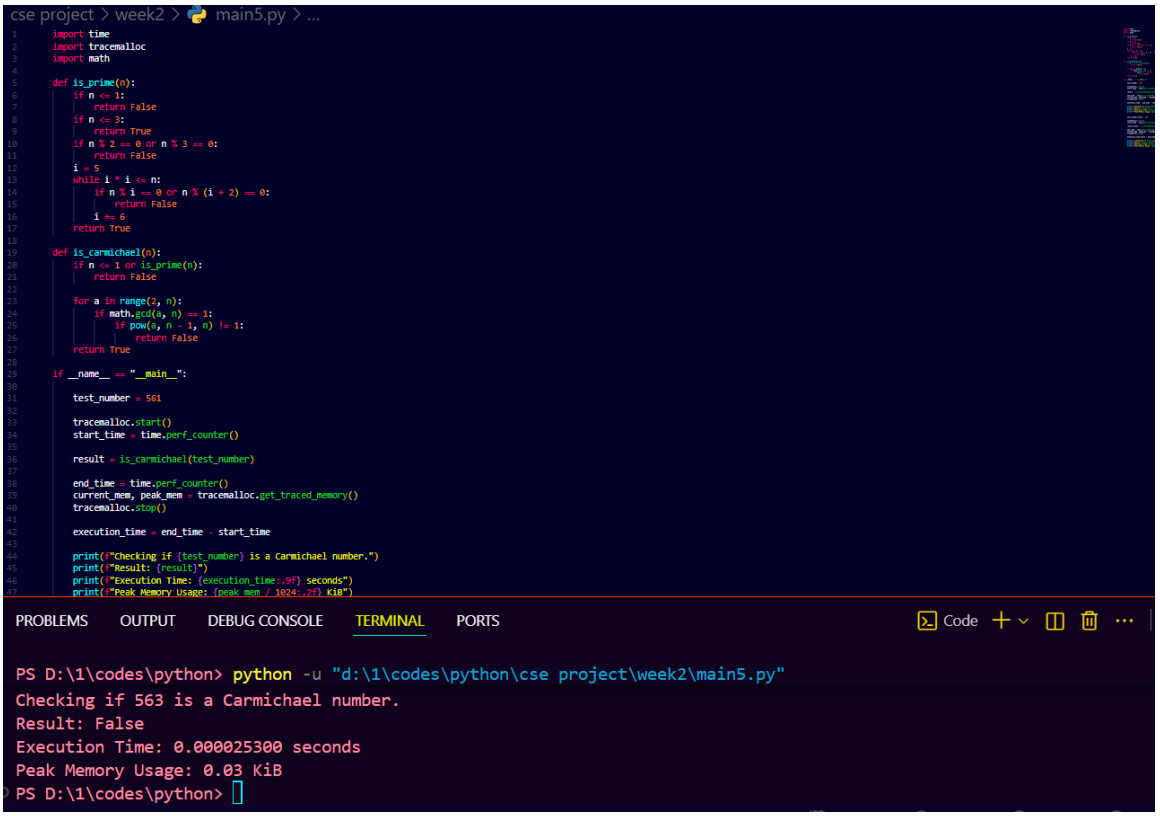
To write a Python function `is_carmichael(n)` that checks if a given composite number `n` is a Carmichael number. A number is a Carmichael number if it is composite and satisfies $a^{n-1} \equiv 1 \pmod n$ for all integers `a` that are coprime to `n`.

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 `a` from 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.gcd` for coprimality testing.
- Use of `pow(a, b, m)` for efficient modular exponentiation.
- Understanding the computational cost of different algorithms.

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	Miller-Rabin Probabilistic Primality Test	16/11/20225	
2	Pollard's Rho Algorithm for Integer Factorization	16/11/20225	
3	Riemann Zeta Function Approximation	16/11/20225	
4	Partition Function Calculator	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Miller-Rabin test

AIM/OBJECTIVE(s) :To implement the probabilistic Miller-Rabin test `is_prime_miller_rabin(n, k)` to determine if a large integer `n` is prime with a high degree of probability using `k` rounds of testing.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

The code implements the Miller-Rabin primality test. It handles base cases (numbers <= 3 and even numbers). It efficiently decomposes `n-1` into powers of 2 and an odd component. The core loop performs the modular exponentiation and squaring steps defined by the algorithm.

RESULTS ACHIEVED:

main > week8 > main.py > ...

```python
import time
import tracemalloc
import random

def is_prime_miller_rabin(n, k=5):
    if n <= 1: return False
    if n <= 3: return True
    if n % 2 == 0: return False

    # Find r and d such that n - 1 = 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Perform k rounds of testing
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)

        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False

    return True

if __name__ == "__main__":

    test_prime = 104729
    rounds = 10

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

    result = is_prime_miller_rabin(test_prime, k=rounds)

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

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week8\main.py"
Checking if 104729 is prime using Miller-Rabin (10 rounds).
Result: True
Execution Time: 0.000628100 seconds
Peak Memory Usage: 0.34 KiB

Checking if 561 is prime using Miller-Rabin (10 rounds).
Result: False
Execution Time: 0.000060100 seconds
Peak Memory Usage: 0.25 KiB
```

SKILLS ACHIEVED:

- Implementation of a probabilistic algorithm.
- Understanding strong pseudoprimes and modular arithmetic.
- Using bitwise logic or division to decompose integers.
- Handling randomness in algorithmic testing.

Practical No: 2

Date: 16/11/2025

TITLE:Pollard's Rho Algorithm for Integer Factorization

AIM/OBJECTIVE(s) : To implement `pollard_rho(n)` to find a non-trivial factor of a composite integer `n` using Pollard's Rho algorithm, which is efficient for finding small factors.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION : The code implements Pollard's Rho algorithm. It uses the "Tortoise and Hare" cycle detection method to find a factor. The polynomial `x² + 1` is used to generate a pseudorandom sequence modulo `n`. The GCD is computed at every step to detect if a factor has been found.

The main execution block (`if __name__ == "__main__":`) tests the function on:

1. 8051 (Factors: 83 and 97).
2. 10403 (Factors: 101 and 103).

It prints the found factor and performance metrics.

Result:

main > week8 > main2.py > ...

```python
import time
import tracemalloc

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

def pollard_rho(n):
    if n % 2 == 0:
        return 2

    x = 2
    y = 2
    d = 1
    f = lambda v: (v * v + 1) % n

    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(abs(x - y), n)

    if d == n:
        return None
    return d

if __name__ == "__main__":

    test_number = 8051

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

    factor = pollard_rho(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"Factorizing {test_number} using Pollard's Rho.")
    print(f"Found factor: {factor}")
    if factor:
        print(f"Other factor: {test_number // factor}")
    print(f"Execution Time: {execution time:.9f} seconds")
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week8\main2.py"
Factorizing 8051 using Pollard's Rho.
Found factor: 97
Other factor: 83
Execution Time: 0.000081200 seconds
Peak Memory Usage: 0.35 KiB

Factorizing 10403 using Pollard's Rho.
Found factor: 101
Execution Time: 0.000129000 seconds
Peak Memory Usage: 0.75 KiB
```

SKILLS ACHIEVED :

- Implementation of integer factorization algorithms.
- Understanding Floyd's cycle-finding algorithm.
- Calculating GCD efficiently using the Euclidean algorithm.
- Manipulating functions as arguments (using lambda for the polynomial).

Practical No: 3

Date: 16/11/2025

TITLE : Riemann Zeta Function Approximation

AIM/OBJECTIVE(s) : To write a Python function `zeta_approx(s, terms)` that approximates the Riemann zeta function ζ(s) by summing the first `terms` of the infinite series.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION : The code implements a direct summation approximation of the Riemann Zeta function. It uses a simple `for` loop to calculate the partial sum of the harmonic series raised to the power `s`.

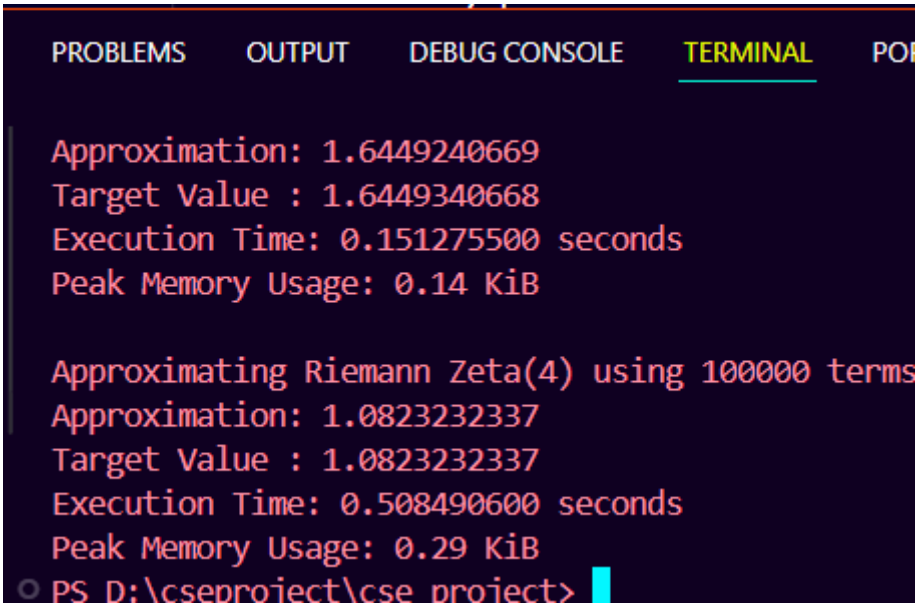
The main execution block (`if __name__ == "__main__":`) calculates approximations for:

1. **`s = 2` with 100,000 terms. (Converges to $\pi^2/6$).**
2. **`s = 4` with 100,000 terms. (Converges to $\pi^4/90$).**

It prints the calculated approximation compared to the known mathematical constants.

RESULTS ACHIEVED:

```
main > week8 > main3.py > ...
1   import time
2   import tracemalloc
3
4   def zeta_approx(s, terms):
5       if s <= 1:
6           return None
7
8       total = 0.0
9       for n in range(1, terms + 1):
10          total += 1 / (n ** s)
11
12      return total
13
14  if __name__ == "__main__":
15
16      s_val = 2
17      terms_count = 100000
18
19      tracemalloc.start()
20      start_time = time.perf_counter()
21
22      result = zeta_approx(s_val, terms_count)
23
24      end_time = time.perf_counter()
25      current_mem, peak_mem = tracemalloc.get_traced_memory()
26      tracemalloc.stop()
27
28      execution_time = end_time - start_time
29
30      # The true value of Zeta(2) is (pi^2) / 6 approx 1.644934
31      true_val_approx = 1.6449340668
32
33      print(f"Approximating Riemann Zeta({s_val}) using {terms_count} terms.")
34      print(f"Approximation: {result:.10f}")
35      print(f"Target Value : {true_val_approx:.10f}")
36      print(f"Execution Time: {execution_time:.9f} seconds")
37      print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```



SKILLS ACHIEVED :

- Implementation of mathematical series summation.
- Floating point arithmetic and precision considerations.
- Comparison of numerical results with theoretical values.

Practical No: 4

Date: 16/11/2025

TITLE : Partition Function Calculator

AIM/OBJECTIVE(s) : To write a Python function `partition_function(n)` that calculates `p(n)`, the number of distinct ways to write `n` as a sum of positive integers (order does not matter).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : The code defines `partition_function(n)`. Instead of a slow recursive approach, it uses an iterative Dynamic Programming approach. It calculates the number of ways to form the sum `n` by progressively considering numbers 1, 2, 3... up to `n` as potential addends.

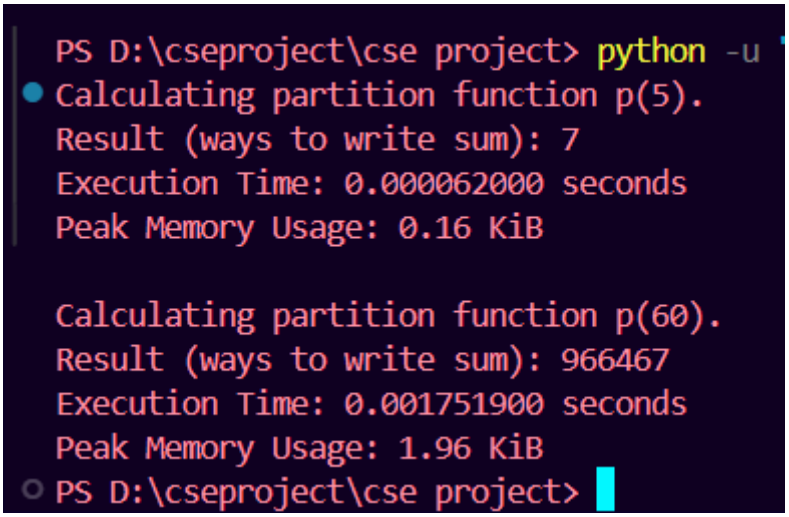
The main execution block (`if __name__ == "__main__":`) calculates:

1. `p(5)` (Ways: 5 -> 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1. Total=7).
2. `p(60)` (A much larger number).

It prints the number of partitions and the performance metrics.

Result:

```python
import time
import tracemalloc

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

    partitions = [0] * (n + 1)
    partitions[0] = 1

    for k in range(1, n + 1):
        for i in range(k, n + 1):
            partitions[i] += partitions[i - k]

    return partitions[n]

if __name__ == "__main__":

    test_number = 5

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

    result = partition_function(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 partition function p({test_number}).")
    print(f"Result (ways to write sum): {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```



SKILLS ACHIEVED :

- Implementation of Dynamic Programming algorithms.
- Solving combinatorial problems (Integer Partitions).
- Optimization of recursive problems into iterative solutions.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL