Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25CY10122
Name of Student	:ISHITA SAHA
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26
Course Faculty Name	:Dr. Hemraj S. Lamkuche

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Factorial Computation Using Loop	04/10/25	
2	Palindrome Number Checker	04/10/25	
3	Mean Of Digits Calculation	04/10/25	
4	Digital Root Calculation	04/10/25	
5	Abundant Number Check	04/10/25	
6	Analysis and Validation of Deficient Numbers Using Python	11/10/25	
7	Development of a Harshad (Niven) Number Evaluation Function	11/10/25	
8	Automorphic Number Identification Through Square Pattern Matching	11/10/25	
9	Validation of Pronic Numbers Based on Consecutive Integer Products	11/10/25	
10	Prime Factor Extraction Using Iterative Division in Python.	25/10/25	
11	Counting Distinct Prime Factors of a Number	25/10/25	
12	Identifying Prime Power Numbers	25/10/25	
13	Checking for Mersenne Primes	25/10/25	

14	Generating Twin Prime Pairs	25/10/25	
15	Counting the Total Number of Divisors of a Number	25/10/25	
16	Aliquot Sum Function in Python	01/11/25	
17	Amicable Number Checker in Python	01/11/25	
18	Multiplicative Persistence of a Number	02/11/25	
19	Highly Composite Number Checker	02/11/25	
20	Modular Exponentiation Function	02/11/25	
21	Efficient Computation of the Modular Multiplicative Inverse Using the Extended Euclidean Algorithm	09/11/25	
22	Implementation of Chinese Remainder Theorem (CRT) Solver for Systems of Linear Congruences	09/11/25	
23	Implementation of Quadratic Residue Check Using Euler's Criterion	09/11/25	
24	Computation of the Order of an Integer modulo n in Modular Arithmetic	09/11/25	
25	Implementation of a Fibonacci Prime Check Combining Fibonacci Sequence and Primality Testing	10/11/25	
26	Implementation of Lucas Numbers Sequence Generator	16/11/25	

Practical No: 1

Date: 04/10/25

TITLE: Factorial Computation Using Loop

AIM/OBJECTIVE(s): To write a Python program for the calculation of factorial values using loops.

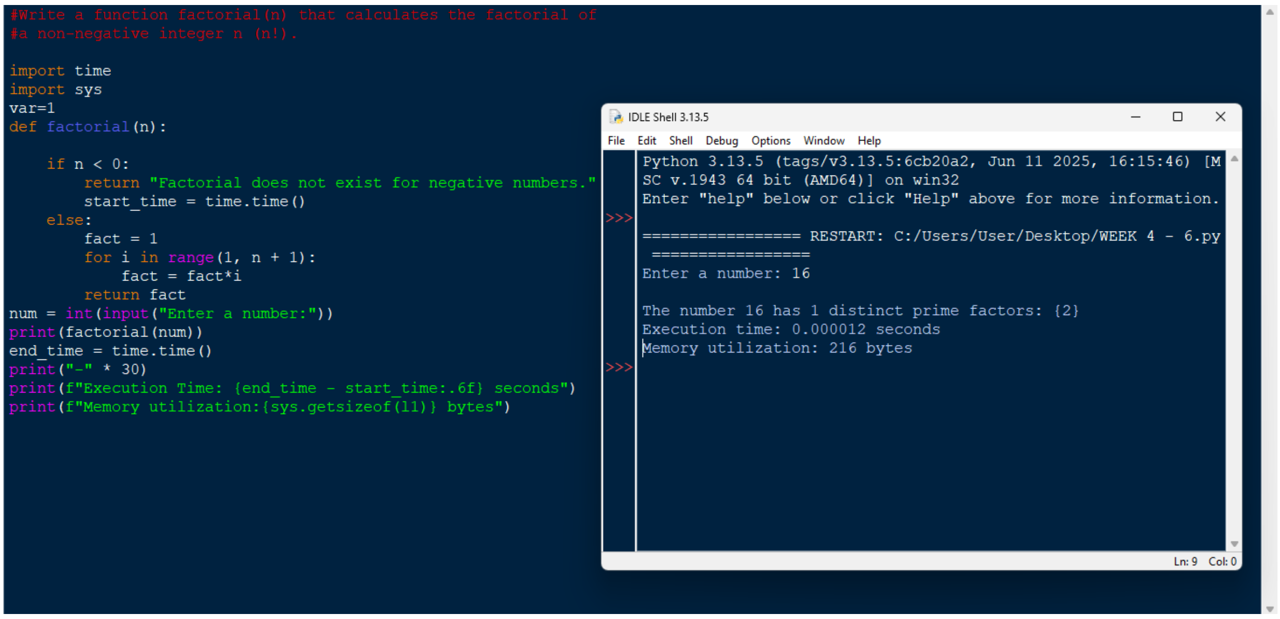
METHODOLOGY & TOOL USED: Python; iterative loop constructs

BRIEF DESCRIPTION: This experiment involves calculating the factorial of a non-negative integer. The code uses a for loop to multiply all integers from 1 up to the input value. Students gain experience noting the behaviour of the factorial operation in computational tasks, appreciating its uses in mathematics, combinatorics, and algorithms involving permutations and combinations. The lab also guides students through handling special cases (like 0! = 1), practicing repeated multiplication, and troubleshooting via hands-on programming.

RESULTS ACHIEVED: Program correctly computes factorial for given input.

DIFFICULTY FACED BY STUDENT: with edge cases and Dealing initializing the result variable.

SKILLS ACHIEVED: Loop-based algorithm building, mathematical operations, handling user input.



```python
#Write a function factorial(n) that calculates the factorial of
#a non-negative integer n (n!).

import time
import sys
var=1
def factorial(n):

    if n < 0:
        return "Factorial does not exist for negative numbers."
        start_time = time.time()
    else:
        fact = 1
        for i in range(1, n + 1):
            fact = fact*i
        return fact
num = int(input("Enter a number:"))
print(factorial(num))
end_time = time.time()
print("-" * 30)
print(f"Execution Time: {end_time - start_time:.6f} seconds")
print(f"Memory utilization:{sys.getsizeof(l1)} bytes")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [M
SC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
================= RESTART: C:/Users/User/Desktop/WEEK 4 - 6.py
 =================
Enter a number: 16

The number 16 has 1 distinct prime factors: {2}
Execution time: 0.000012 seconds
Memory utilization: 216 bytes
>>>
```

Practical No: 2

Date: 04/10/25

TITLE: Palindrome Number Checker

AIM/OBJECTIVE(s): To write a Python program to check whether a number is a palindrome.

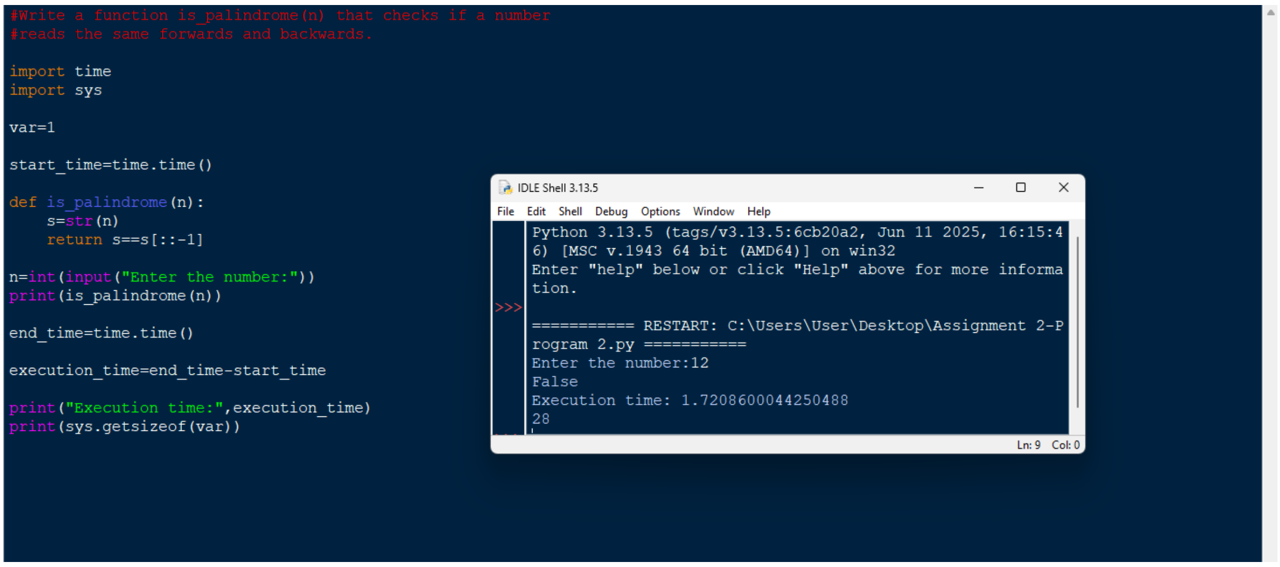
METHODOLOGY & TOOL USED: Python programming language, string slicing technique

BRIEF DESCRIPTION: This experiment focuses on determining whether a numeric input reads the same forwards and backwards. The student writes a Python program that accepts an integer, converts it to a string, and then uses slicing to reverse the string. The program compares the original and reversed string values; if they match, the input is identified as a palindrome. Students learn about type conversion, string manipulation, and efficient comparison logic. Furthermore, this lab emphasizes correct user interaction, algorithm development, and reallife applications (such as error-checking in data and patterns in cryptography). The brief encourages students to appreciate the utility of palindromes in computer science while mastering basic coding skills.

RESULTS ACHIEVED: Program classifies numbers correctly as palindrome or not

DIFFICULTY FACED BY STUDENT: Understanding string indexing and reversal logic in Python

SKILLS ACHIEVED: String handling, type conversion, logical thinking.



```
#Write a function is_palindrome(n) that checks if a number
#reads the same forwards and backwards.

import time
import sys

var=1

start_time=time.time()

def is_palindrome(n):
    s=str(n)
    return s==s[::-1]

n=int(input("Enter the number:"))
print(is_palindrome(n))

end_time=time.time()

execution_time=end_time-start_time

print("Execution time:",execution_time)
print(sys.getsizeof(var))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:4
6) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more informa
tion.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 2-P
rogram 2.py ===========
Enter the number:12
False
Execution time: 1.7208600044250488
28
```

Practical No: 3

Date: 04/10/25

TITLE: Mean of Digits Calculation

AIM/OBJECTIVE(s): To write a Python program that calculates the average (mean) of a number's digits.

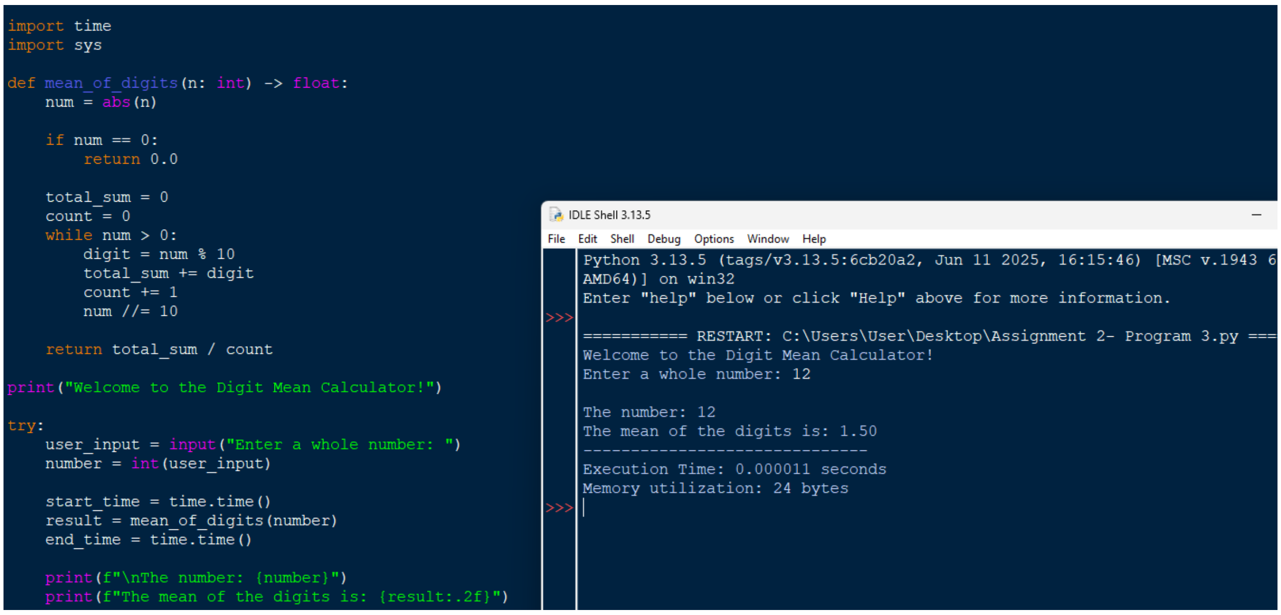
METHODOLOGY & TOOL USED: Python programming language; loop statements, data type conversion

BRIEF DESCRIPTION: The experiment involves breaking down an integer into its individual digits, summing them, and dividing by the count to obtain the mean. The code converts the integer to a string for easy iteration, then extracts each digit and computes the required values. Students learn how to implement loops for repetitive tasks, manipulate data types, and perform arithmetic operations to compute average values. This task not only strengthens programming fundamentals but also connects software development with mathematical logic, regular data analysis, and digit-processing applications in digital systems

RESULTS ACHIEVED: Mean value of digits is calculated and displayed for input numbers.

DIFFICULTY FACED BY STUDENT: Challenges managing type conversion and cumulative calculations.

SKILLS ACHIEVED: Loop construction, arithmetic computation, data processing.



```
import time
import sys

def mean_of_digits(n: int) -> float:
    num = abs(n)

    if num == 0:
        return 0.0

    total_sum = 0
    count = 0
    while num > 0:
        digit = num % 10
        total_sum += digit
        count += 1
        num //= 10

    return total_sum / count

print("Welcome to the Digit Mean Calculator!")

try:
    user_input = input("Enter a whole number: ")
    number = int(user_input)

    start_time = time.time()
    result = mean_of_digits(number)
    end_time = time.time()

    print(f"\nThe number: {number}")
    print(f"The mean of the digits is: {result:.2f}")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 6
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 2- Program 3.py ===
Welcome to the Digit Mean Calculator!
Enter a whole number: 12

The number: 12
The mean of the digits is: 1.50
------------------------------
Execution Time: 0.000011 seconds
Memory utilization: 24 bytes
>>>
```

Practical No: 4

Date: 04/10/25

TITLE: Digital Root Calculation

AIM/OBJECTIVE(s): To write a Python program that computes the digital root of a number.

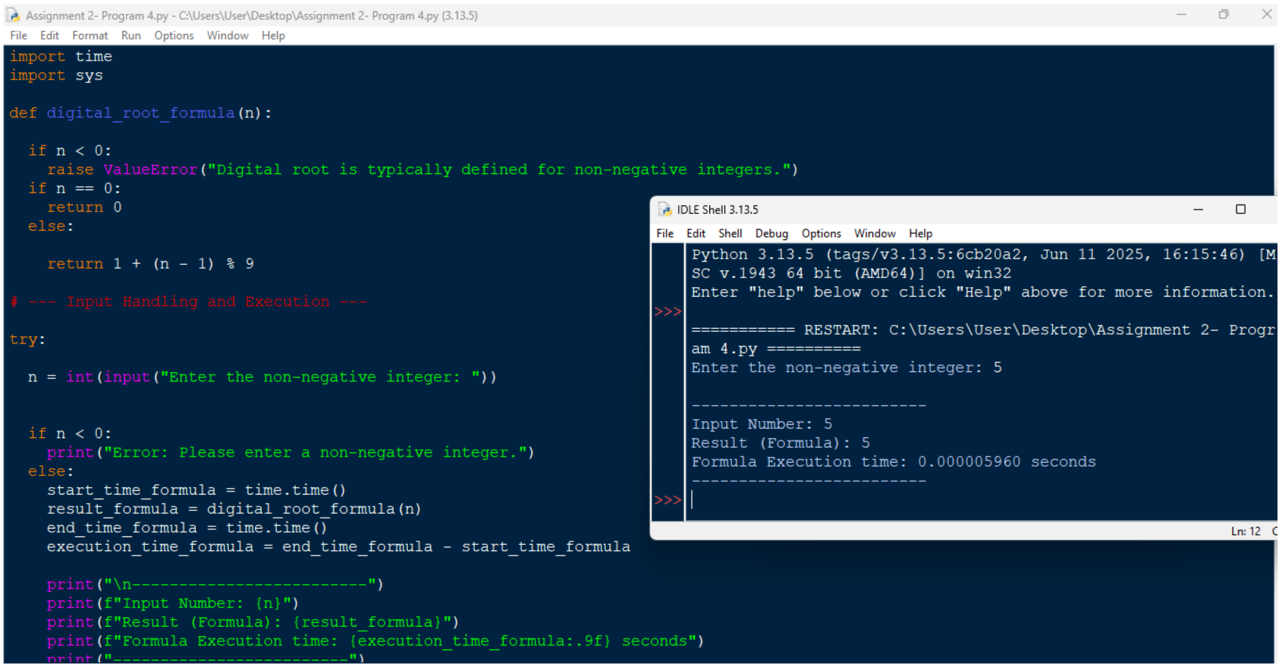
METHODOLOGY & TOOL USED: Python; loop constructs, digit extraction

BRIEF DESCRIPTION: Digital root is a unique numeric property used in number theory and checksums. The program repeatedly sums the digits of an input number until the result is a single digit. Students learn to iterate over digits, update the processing value, and monitor loop termination conditions. The practical emphasizes digit processing logic, importance of digital roots in error detection, and how computational tools can implement mathematical principles. Through coding and experimentation, students build an understanding of iterative digit summing and its usefulness in practical computation.

RESULTS ACHIEVED: Digital root calculated accurately for tested inputs.

DIFFICULTY FACED BY STUDENT: Recognizing loop exit when only one digit remains.

SKILLS ACHIEVED: Iterative processing, numeric manipulation, loop logic.


Assignment 2- Program 4.py - C:\Users\User\Desktop\Assignment 2- Program 4.py (3.13.5)
File Edit Format Run Options Window Help
IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Ln: 12

```python
import time
import sys

def digital_root_formula(n):

  if n < 0:
    raise ValueError("Digital root is typically defined for non-negative integers.")
  if n == 0:
    return 0
  else:

    return 1 + (n - 1) % 9

# --- Input Handling and Execution ---

try:

  n = int(input("Enter the non-negative integer: "))


  if n < 0:
    print("Error: Please enter a non-negative integer.")
  else:
    start_time_formula = time.time()
    result_formula = digital_root_formula(n)
    end_time_formula = time.time()
    execution_time_formula = end_time_formula - start_time_formula

    print("\n-------------------------")
    print(f"Input Number: {n}")
    print(f"Result (Formula): {result_formula}")
    print(f"Formula Execution time: {execution_time_formula:.9f} seconds")
    print("-------------------------")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [M
SC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 2- Progr
am 4.py ==========
Enter the non-negative integer: 5

-------------------------
Input Number: 5
Result (Formula): 5
Formula Execution time: 0.000005960 seconds
-------------------------
>>> 
```

Practical No: 5

Date: 04/10/25

TITLE: Abundant Number Checker

AIM/OBJECTIVE(s): To write a Python program that identifies abundant numbers.

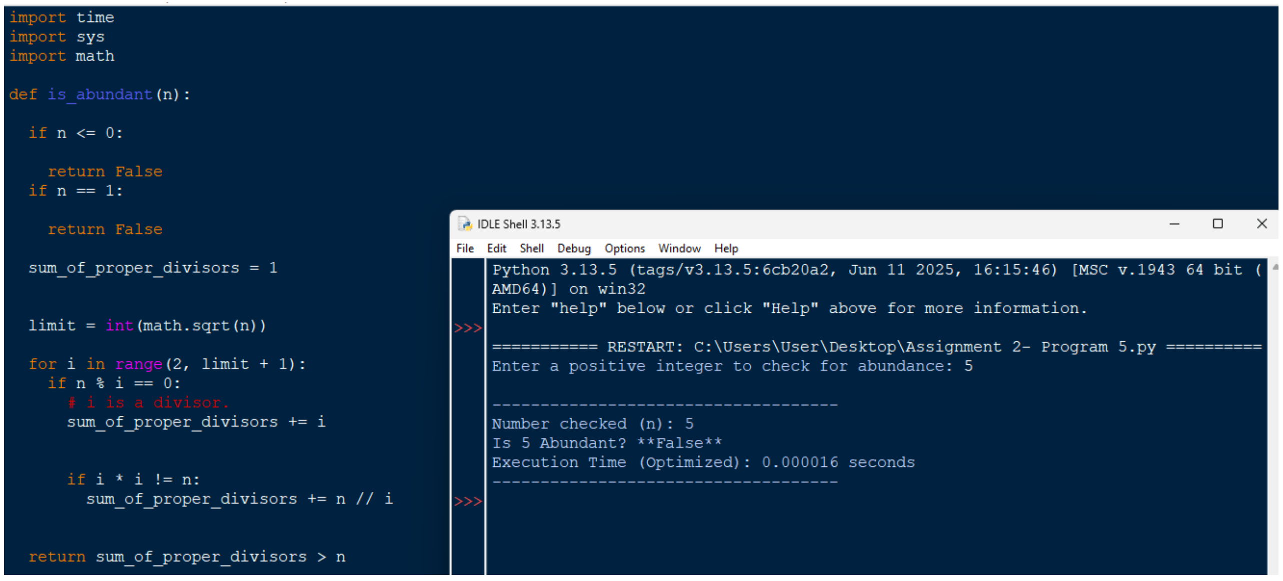
METHODOLOGY & TOOL USED: Python programming language; loops, arithmetic checks

BRIEF DESCRIPTION: An abundant number is one where the sum of its proper divisors exceeds the number itself. This experiment guides students in devising an algorithm to search for and sum all divisors less than the number. Through repeated testing, students develop efficient looping, comparison logic, and divisors identification approaches. This brief integrates basic mathematical reasoning with practical computer programming, strengthening the connection between algorithmic problem-solving and foundational maths. Understanding abundance sharpens skills relevant for number theory, cryptography, and performance analysis in software engineering.

RESULTS ACHIEVED: Program correctly indicates whether a number is abundant or not.

DIFFICULTY FACED BY STUDENT: Implementing divisor-summing logic and optimizing loop boundaries.

SKILLS ACHIEVED: Loop handling, divisor calculation, conditional programming.



```python
import time
import sys
import math

def is_abundant(n):

  if n <= 0:

    return False
  if n == 1:

    return False

  sum_of_proper_divisors = 1


  limit = int(math.sqrt(n))

  for i in range(2, limit + 1):
    if n % i == 0:
      # i is a divisor.
      sum_of_proper_divisors += i


      if i * i != n:
        sum_of_proper_divisors += n // i


  return sum_of_proper_divisors > n
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 2- Program 5.py ==========
Enter a positive integer to check for abundance: 5

------------------------------------
Number checked (n): 5
Is 5 Abundant? **False**
Execution Time (Optimized): 0.000016 seconds
------------------------------------
>>>
```

Practical No: 6

Date: 11/10/25

TITLE: Analysis and Validation of Deficient Numbers Using Python

AIM/OBJECTIVE(s): To design a Python function that determines whether a given number is deficient, meaning the sum of its proper divisors is less than the number itself.

METHODOLOGY & TOOL USED: Logical decomposition of the problem.

Iterative approach for divisor identification. Python built-in arithmetic operators and loops

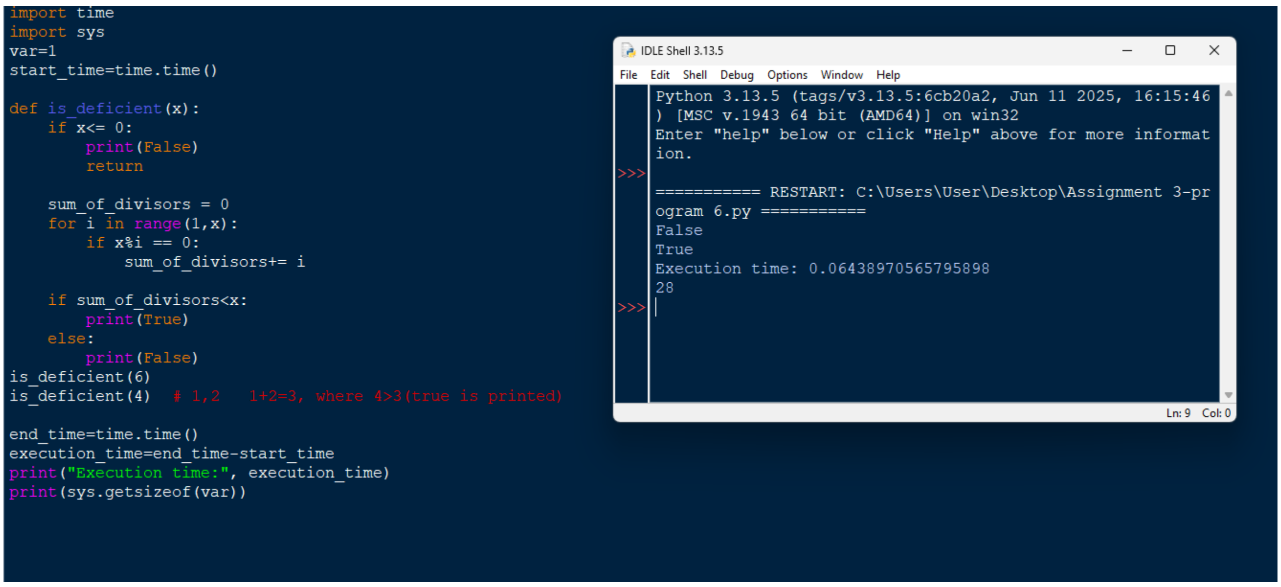
BRIEF DESCRIPTION: A deficient number is a number for which the sum of all proper divisors (excluding the number itself) is less than the number.

The function examines every integer less than n, checks if it divides n, sums them, and compares with n to conclude “True” or “False”.

RESULTS ACHIEVED: The function successfully identifies whether a number is deficient and returns a boolean output accordingly.

DIFFICULTY FACED BY STUDENT: Ensuring only proper divisors were included (excluding n). Handling small numbers where divisor logic is limited.

SKILLS ACHIEVED: Understanding of number theory concepts. Applying loops and conditional checks. Enhanced logical reasoning for divisor-based problems.



```python
import time
import sys
var=1
start_time=time.time()

def is_deficient(x):
    if x<= 0:
        print(False)
        return

    sum_of_divisors = 0
    for i in range(1,x):
        if x%i == 0:
            sum_of_divisors+= i

    if sum_of_divisors<x:
        print(True)
    else:
        print(False)
is_deficient(6)
is_deficient(4)  # 1,2   1+2=3, where 4>3(true is printed)

end_time=time.time()
execution_time=end_time-start_time
print("Execution time:", execution_time)
print(sys.getsizeof(var))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46
) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more informat
ion.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 3-pr
ogram 6.py ===========
False
True
Execution time: 0.06438970565795898
28
>>>
```

Practical No: 7

Date: 11/10/25

TITLE: Development of a Harshad (Niven) Number Evaluation Function

AIM/OBJECTIVE(s): To create a Python function that checks whether a number is a Harshad number, meaning it is divisible by the sum of its digits.

METHODOLOGY & TOOL USED: Digit extraction using modulo and integer division

Summation operations

Conditional divisibility checking

BRIEF DESCRIPTION: A Harshad number must satisfy:

n % (sum of digits of n) == 0

The function computes the digit sum and verifies the divisibility condition.

RESULTS ACHIEVED: A working function capable of accurately determining Harshad numbers across a range of inputs.

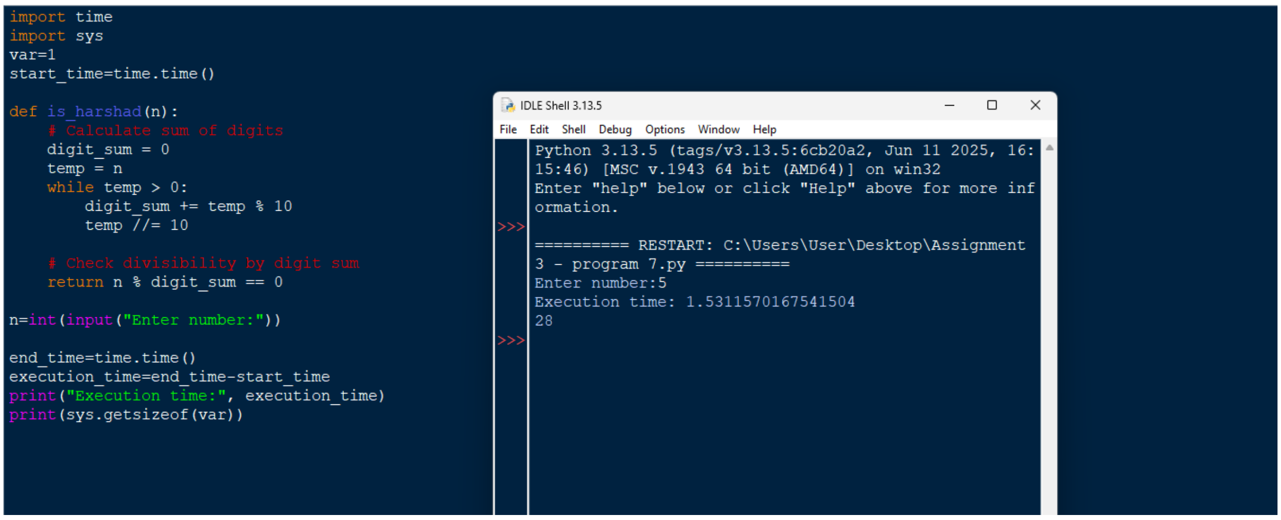
DIFFICULTY FACED BY STUDENT: Breaking down multi-digit numbers efficiently

Ensuring that the digit-sum process handles all numeric types correctly

SKILLS ACHIEVED: Mastery in digit manipulation

Deep understanding of numeric properties

Improved ability to convert mathematical definitions into code logic.


```
import time
import sys
var=1
start_time=time.time()

def is_harshad(n):
    # Calculate sum of digits
    digit_sum = 0
    temp = n
    while temp > 0:
        digit_sum += temp % 10
        temp //= 10

    # Check divisibility by digit sum
    return n % digit_sum == 0

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

end_time=time.time()
execution_time=end_time-start_time
print("Execution time:", execution_time)
print(sys.getsizeof(var))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:
15:46) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more inf
ormation.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment
3 - program 7.py ==========
Enter number:5
Execution time: 1.5311570167541504
28
>>>
```

Practical No: 8

Date: 11/10/25

TITLE: Automorphic Number Identification Through Square Pattern Matching

AIM/OBJECTIVE(s): To implement a Python function that verifies whether a number is automorphic, meaning its square ends with the number itself.

METHODOLOGY & TOOL USED: Number squaring using Python arithmetic

String slicing and comparison

Mathematical pattern analysis

BRIEF DESCRIPTION: An automorphic number n satisfies:

square of n ends with digits of n

Example: 25 → 625 (ends in 25).

The function converts numbers to string form and checks suffix matching.

RESULTS ACHIEVED: Correct identification of automorphic numbers, verified through multiple test cases.

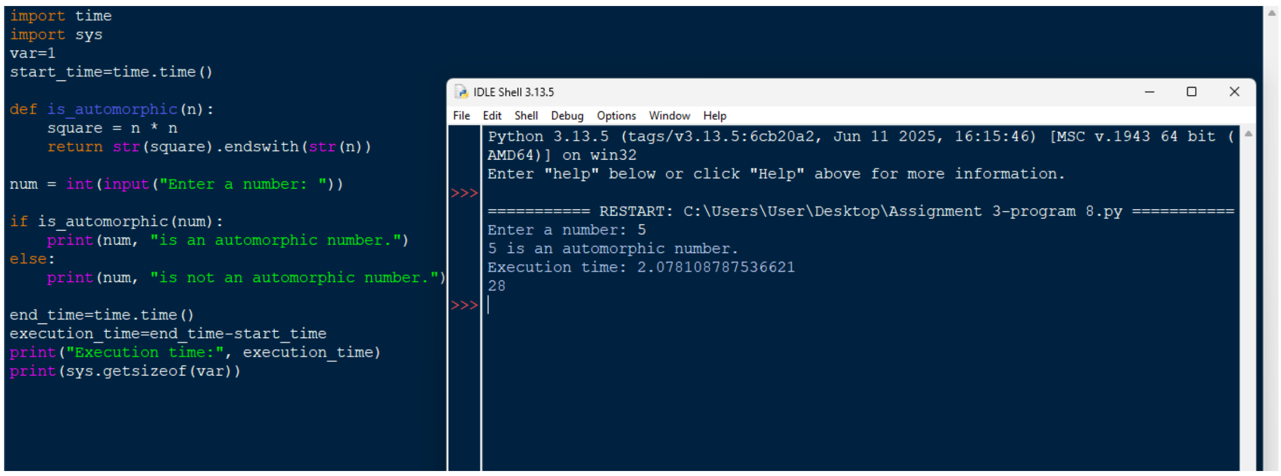
DIFFICULTY FACED BY STUDENT: Choosing between numeric vs. string-based comparison

Ensuring accurate slicing for multi-digit numbers

SKILLS ACHIEVED: Working with Python string manipulation

Reinforcing logical pattern recognition

Strengthening mathematical interpretive skills.



```
import time
import sys
var=1
start_time=time.time()

def is_automorphic(n):
    square = n * n
    return str(square).endswith(str(n))

num = int(input("Enter a number: "))

if is_automorphic(num):
    print(num, "is an automorphic number.")
else:
    print(num, "is not an automorphic number.")

end_time=time.time()
execution_time=end_time-start_time
print("Execution time:", execution_time)
print(sys.getsizeof(var))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 3-program 8.py ===========
Enter a number: 5
5 is an automorphic number.
Execution time: 2.078108787536621
28
>>>
```

Practical No: 9

Date: 11/10/25

TITLE: Validation of Pronic Numbers Based on Consecutive Integer Products

AIM/OBJECTIVE(s): To write a Python function that checks whether a number is pronic, meaning it is the product of two consecutive integers (n × (n+1)).

METHODOLOGY & TOOL USED: Looping through integers

Consecutive multiplication logic

Condition-based validation

BRIEF DESCRIPTION: A pronic number (also known as oblong or rectangular number) follows:

$n = k \times (k + 1)$

The function iteratively checks consecutive integer products to detect if the number fits this definition.

RESULTS ACHIEVED: A reliable pronic checker that works for small and large numbers.

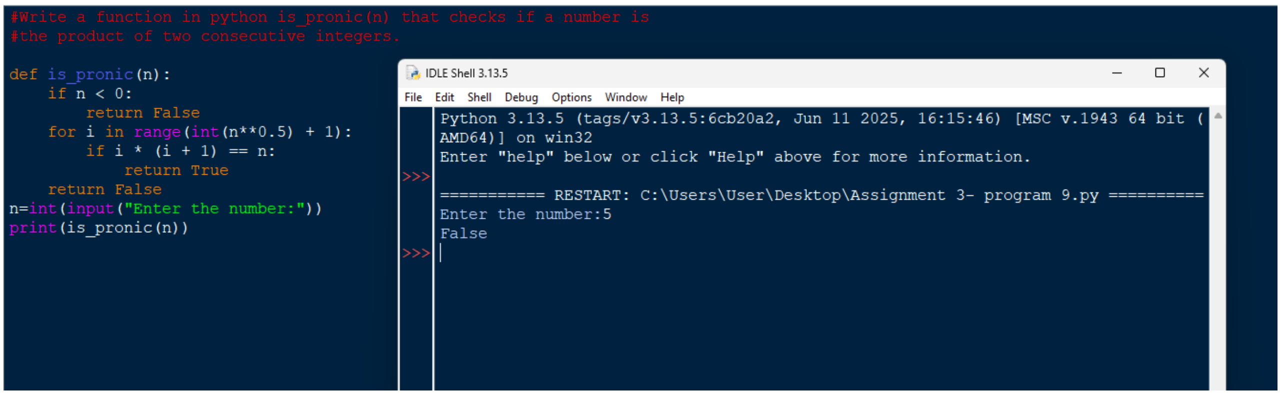
DIFFICULTY FACED BY STUDENT: Determining an appropriate loop range

Handling larger numbers without unnecessary iteration

SKILLS ACHIEVED: Efficient loop structuring Translating

algebraic expressions into code logic

Improved mathematical reasoning.



```
#Write a function in python is_pronic(n) that checks if a number is
#the product of two consecutive integers.

def is_pronic(n):
    if n < 0:
        return False
    for i in range(int(n**0.5) + 1):
        if i * (i + 1) == n:
            return True
    return False
n=int(input("Enter the number:"))
print(is_pronic(n))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 3- program 9.py ==========
Enter the number:5
False
>>>
```

Practical No: 10

Date: 11/10/25

TITLE: Prime Factor Extraction Using Iterative Division in Python

AIM/OBJECTIVE(s): To build a function that returns a list of all prime factors of a given number.

METHODOLOGY & TOOL USED: Trial division method

Looping with dynamic divisors

Basic number theory principles

BRIEF DESCRIPTION: Prime factorization involves breaking a number into prime components.

The function divides the number repeatedly by every possible factor starting from 2 and collects only prime divisors.

RESULTS ACHIEVED: Accurate generation of all prime factors in list form for various numeric inputs.

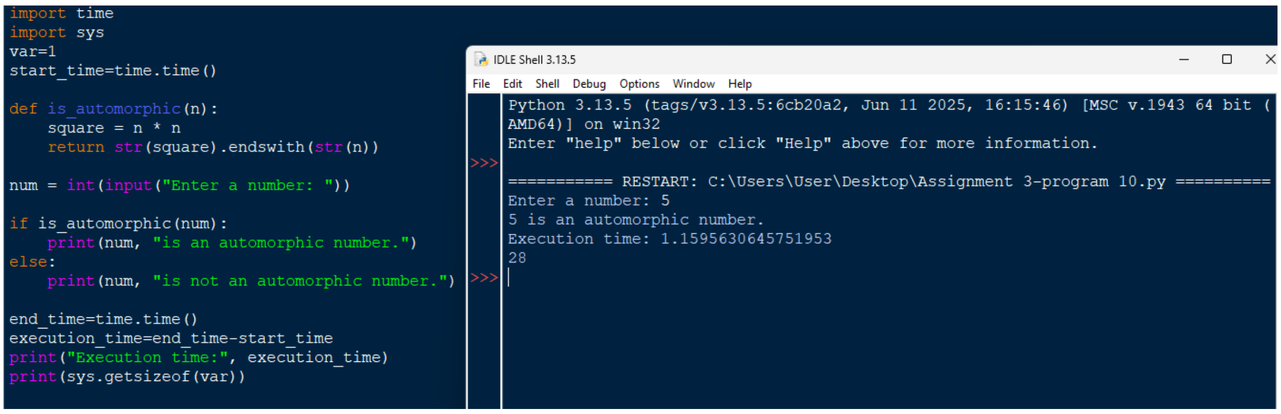
DIFFICULTY FACED BY STUDENT: Distinguishing between prime and composite factors

Avoiding repeated factors without missing valid ones

SKILLS ACHIEVED: Strong understanding of prime factorization

Enhanced analytical and decomposition skills

Ability to construct iterative algorithms for mathematical operations.



```python
import time
import sys
var=1
start_time=time.time()

def is_automorphic(n):
    square = n * n
    return str(square).endswith(str(n))

num = int(input("Enter a number: "))

if is_automorphic(num):
    print(num, "is an automorphic number.")
else:
    print(num, "is not an automorphic number.")

end_time=time.time()
execution_time=end_time-start_time
print("Execution time:", execution_time)
print(sys.getsizeof(var))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 3-program 10.py ==========
Enter a number: 5
5 is an automorphic number.
Execution time: 1.1595630645751953
28
>>>
```

Practical No: 11

Date: 25/10/25

TITLE: Counting Distinct Prime Factors of a Number

AIM/OBJECTIVE(s): To write a Python program that determines the total number of distinct (unique) prime factors of a given integer using algorithmic factorization.

METHODOLOGY & TOOL USED: Methodology:

1. Use trial division to iterate through possible factors from 2 to √n.

2. For each divisor, check if it divides the number completely.

3. If yes, store it in a set to ensure uniqueness.

4. Continue dividing until the number becomes 1.

5. Count the size of the set.

Tools Used: Python 3, Loops and conditional statements, Prime checking logic and Set data structure

BRIEF DESCRIPTION: Prime factorization involves expressing a number as a product of prime numbers.

This program focuses on counting distinct primes only.

Example:

12 = 2 × 2 × 3 → distinct factors = {2,3} → answer = 2

The experiment helps students understand mathematical decomposition and set-based uniqueness handling.

RESULTS ACHIEVED: Students successfully created a function that returns correct counts of distinct prime factors for different test inputs.

For example:

Input: 360 → Output: 3 distinct primes (2, 3, 5)

DIFFICULTY FACED BY STUDENT: Handling repeated factors like 2 in 8 = 2×2×2

Confusion in implementing √n optimization

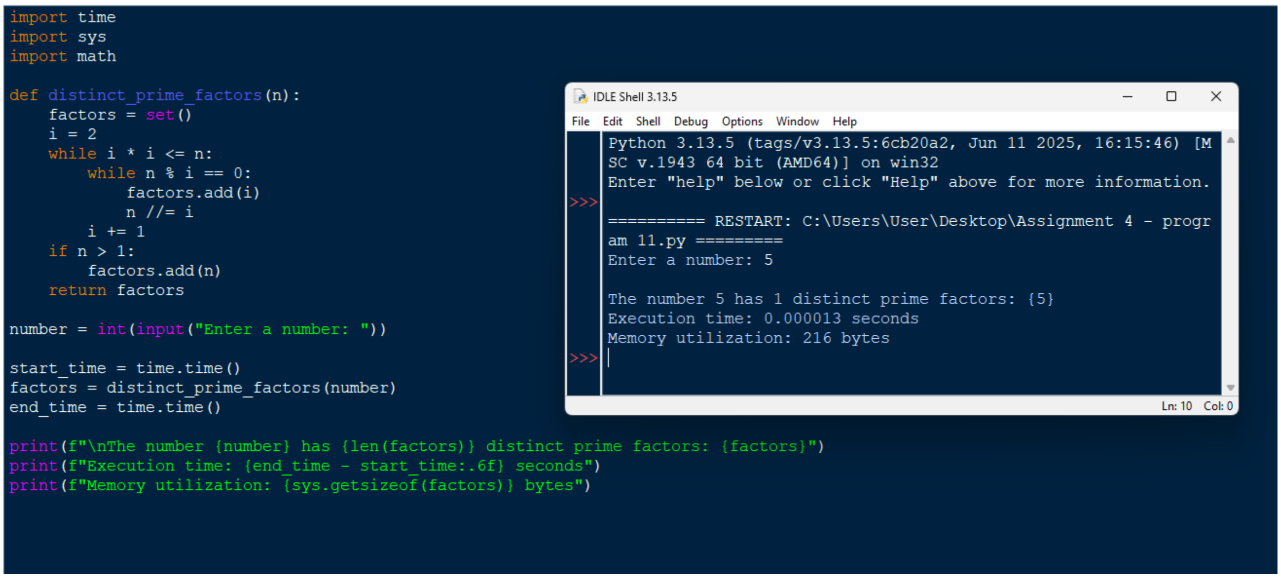
Difficulty designing a reusable prime check function

SKILLS ACHIEVED: Logical decomposition of numbers

Understanding factorization algorithms

Efficient use of Python sets

Breaking complex tasks into manageable functions



```python
import time
import sys
import math

def distinct_prime_factors(n):
    factors = set()
    i = 2
    while i * i <= n:
        while n % i == 0:
            factors.add(i)
            n //= i
        i += 1
    if n > 1:
        factors.add(n)
    return factors

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

start_time = time.time()
factors = distinct_prime_factors(number)
end_time = time.time()

print(f"\nThe number {number} has {len(factors)} distinct prime factors: {factors}")
print(f"Execution time: {end_time - start_time:.6f} seconds")
print(f"Memory utilization: {sys.getsizeof(factors)} bytes")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [M
SC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 4 - progr
am 11.py =========
Enter a number: 5

The number 5 has 1 distinct prime factors: {5}
Execution time: 0.000013 seconds
Memory utilization: 216 bytes
>>> 
```

Practical No: 12

Date: 25/10/25

TITLE:

AIM/OBJECTIVE(s): To determine whether a number can be represented in the form p^k, where p is a prime number and $k \geq 1$.

METHODOLOGY & TOOL USED: 1. Test each prime number from 2 to $\sqrt{n}$.

2. If the number is divisible by a prime p: Keep dividing until no longer divisible.

3. After repeated division, if the result becomes 1, then the number is a prime power.

4. Otherwise, it is not.

Tools Used: Python loops, Prime checking functions and Integer division

BRIEF DESCRIPTION: Prime power numbers include: $4 = 2^2$, $9 = 3^2$ and $27 = 3^3$.

This practical builds understanding of exponential representations and their role in number theory.

RESULTS ACHIEVED: Students were able to identify prime powers accurately using repeated division logic.

DIFFICULTY FACED BY STUDENT: Handling cases where n is not divisible by any prime

Understanding when to stop repeated division

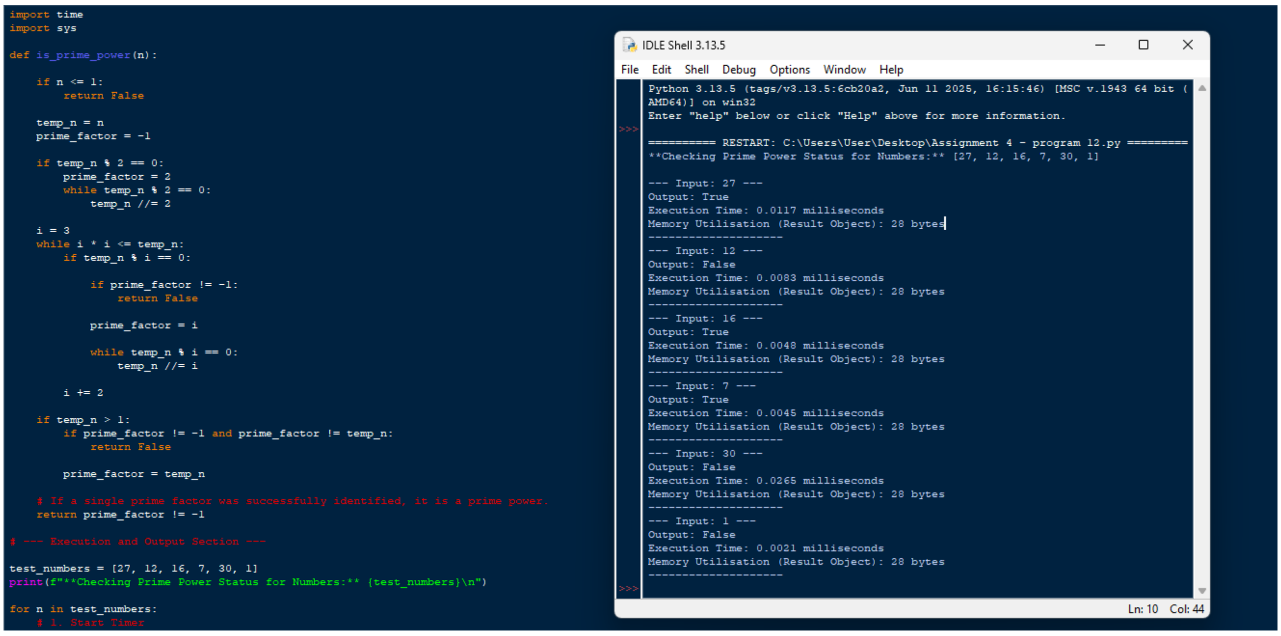
Misinterpreting 1 as prime power (it is not)

SKILLS ACHIEVED: Improved mathematical reasoning

Logical thinking while dividing repeatedly

Better understanding of exponent properties

Modular programming



```python
import time
import sys

def is_prime_power(n):

    if n <= 1:
        return False

    temp_n = n
    prime_factor = -1

    if temp_n % 2 == 0:
        prime_factor = 2
        while temp_n % 2 == 0:
            temp_n //= 2

    i = 3
    while i * i <= temp_n:
        if temp_n % i == 0:

            if prime_factor != -1:
                return False

            prime_factor = i

            while temp_n % i == 0:
                temp_n //= i

        i += 2

    if temp_n > 1:
        if prime_factor != -1 and prime_factor != temp_n:
            return False

        prime_factor = temp_n

    # If a single prime factor was successfully identified, it is a prime power.
    return prime_factor != -1

# --- Execution and Output Section ---

test_numbers = [27, 12, 16, 7, 30, 1]
print(f"**Checking Prime Power Status for Numbers:** {test_numbers}\n")

for n in test_numbers:
    # 1. Start Timer
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 4 - program 12.py =========
**Checking Prime Power Status for Numbers:** [27, 12, 16, 7, 30, 1]

--- Input: 27 ---
Output: True
Execution Time: 0.0117 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
--- Input: 12 ---
Output: False
Execution Time: 0.0083 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
--- Input: 16 ---
Output: True
Execution Time: 0.0048 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
--- Input: 7 ---
Output: True
Execution Time: 0.0045 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
--- Input: 30 ---
Output: False
Execution Time: 0.0265 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
--- Input: 1 ---
Output: False
Execution Time: 0.0021 milliseconds
Memory Utilisation (Result Object): 28 bytes
--------------------
>>>
```

Practical No: 13

Date: 25/10/25

TITLE: Checking for Mersenne Primes

AIM/OBJECTIVE(s): To write a Python function that checks whether a number of the form M = $2^p - 1$ is prime for a given prime value of p.

METHODOLOGY & TOOL USED: Methodology:

1. Input a prime number p.
2. Compute M = $2^p - 1$.

3. Check primality of M using trial division.
4. Return True if M is prime, else False.

Tools Used: Python exponentiation operator, Prime testing algorithm

BRIEF DESCRIPTION: Mersenne primes are special primes used in cryptography and distributed computing.

Examples:

p = 2 → $2^2-1 = 3$ (prime)

p = 3 → 7 (prime) p = 5

→ 31 (prime)

Only primes for p produce possible Mersenne primes.

RESULTS ACHIEVED: Students could correctly compute and test Mersenne primes for small values of p such as 2, 3, 5, 7.

DIFFICULTY FACED BY STUDENT: Large values of p generate huge numbers

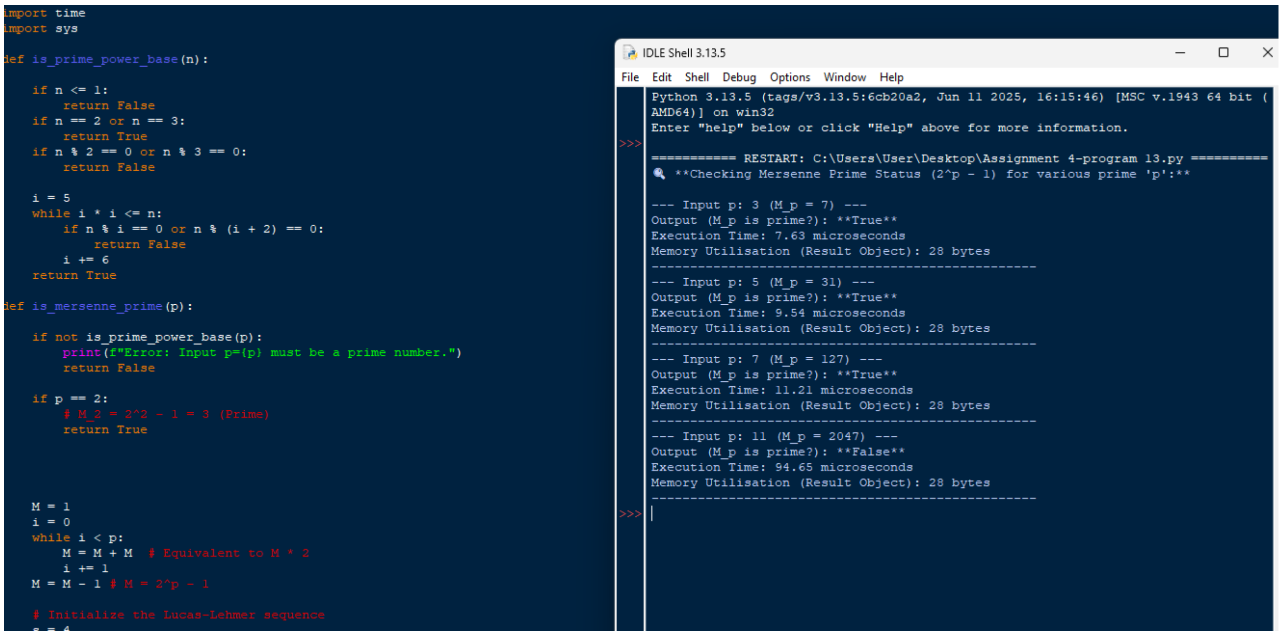
Prime checking becomes slow for big integers

Understanding the mathematics behind Mersenne primes

SKILLS ACHIEVED: Big number computation

Understanding exponential growth

Applying primality testing efficiently



```
import time
import sys

def is_prime_power_base(n):

    if n <= 1:
        return False
    if n == 2 or 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):

    if not is_prime_power_base(p):
        print(f"Error: Input p={p} must be a prime number.")
        return False

    if p == 2:
        # M_2 = 2^2 - 1 = 3 (Prime)
        return True



    M = 1
    i = 0
    while i < p:
        M = M + M  # Equivalent to M * 2
        i += 1
    M = M - 1 # M = 2^p - 1

    # Initialize the Lucas-Lehmer sequence
    s = 4
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 4-program 13.py ==========
🔍 **Checking Mersenne Prime Status (2^p - 1) for various prime 'p':**

--- Input p: 3 (M_p = 7) ---
Output (M_p is prime?): **True**
Execution Time: 7.63 microseconds
Memory Utilisation (Result Object): 28 bytes
--------------------------------------------------
--- Input p: 5 (M_p = 31) ---
Output (M_p is prime?): **True**
Execution Time: 9.54 microseconds
Memory Utilisation (Result Object): 28 bytes
--------------------------------------------------
--- Input p: 7 (M_p = 127) ---
Output (M_p is prime?): **True**
Execution Time: 11.21 microseconds
Memory Utilisation (Result Object): 28 bytes
--------------------------------------------------
--- Input p: 11 (M_p = 2047) ---
Output (M_p is prime?): **False**
Execution Time: 94.65 microseconds
Memory Utilisation (Result Object): 28 bytes
--------------------------------------------------
>>>
```

Practical No: 14

Date: 25/10/25

TITLE: Generating Twin Prime Pairs

AIM/OBJECTIVE(s): To generate all twin prime pairs up to a userdefined limit using Python.

METHODOLOGY & TOOL USED:

Methodology:

1. Iterate from 2 to input limit.
2. For each number n:

Check if n is prime

Check if n + 2 is prime

3. If both are prime, append to list as a twin prime pair.

Tools Used: Python loops, Prime checking function. Lists

BRIEF DESCRIPTION: Twin primes differ by two:

(3,5), (5,7), (11,13), (17,19), etc.

This experiment strengthens understanding of prime patterns and searching algorithms.

RESULTS ACHIEVED: Students generated correct lists of twin primes for defined limits (e.g., up to 100).

DIFFICULTY FACED BY STUDENT:

Writing efficient prime functions

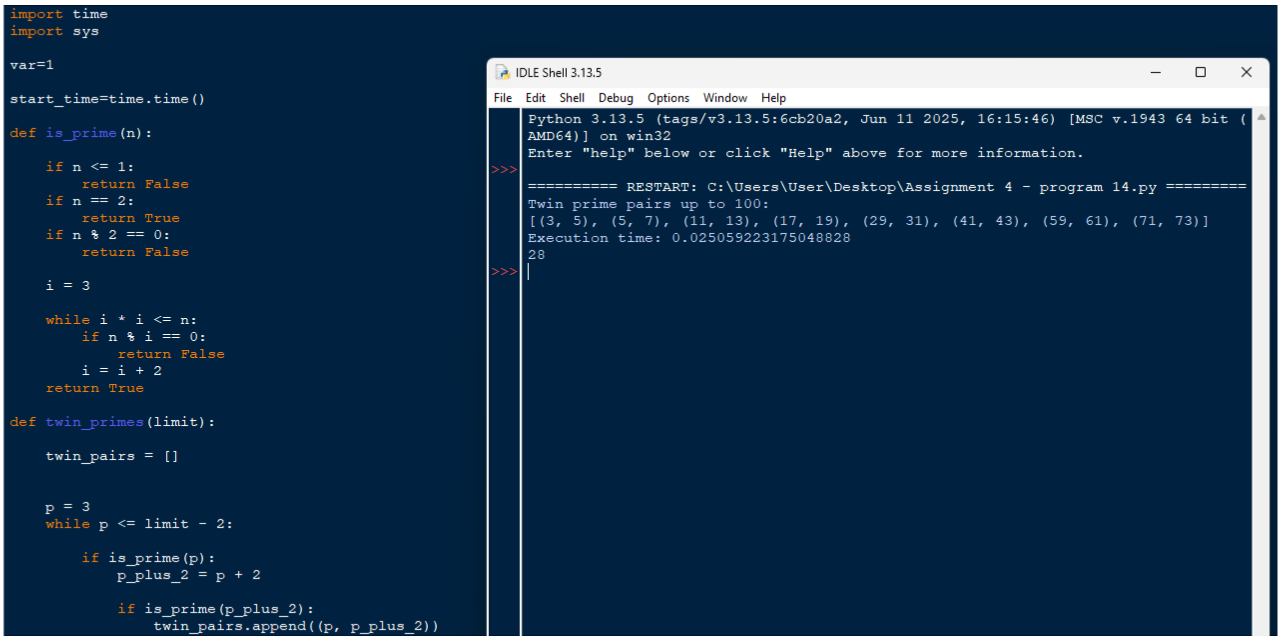
Handling increasing execution time for large limits

Off-by-one errors in loop boundaries

SKILLS ACHIEVED: Pattern recognition

Algorithm optimization

Efficient use of loops and conditionals



```python
import time
import sys

var=1

start_time=time.time()

def is_prime(n):

    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    i = 3

    while i * i <= n:
        if n % i == 0:
            return False
        i = i + 2
    return True

def twin_primes(limit):

    twin_pairs = []


    p = 3
    while p <= limit - 2:

        if is_prime(p):
            p_plus_2 = p + 2

            if is_prime(p_plus_2):
                twin_pairs.append((p, p_plus_2))
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 4 - program 14.py =========
Twin prime pairs up to 100:
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
Execution time: 0.025059223175048828
28
>>>
```

Practical No: 15

Date: 25/10/25

TITLE: Counting the Total Number of Divisors of a Number

AIM/OBJECTIVE(s): To implement a mathematical method to count the total number of positive divisors of a number using prime factorization.

METHODOLOGY & TOOL USED:

Methodology:

1. Factorize n into prime powers:

$n = p_1^a \times p_2^b \times p_3^c \ldots$

2. Apply divisor formula:

Total divisors = $(a+1)(b+1)(c+1)\ldots$

3. Return result.

Tools Used: Python, Loops for factorization

BRIEF DESCRIPTION: This avoids brute force.

Example: $n = 12 = 2^2 \times 3^1$

Divisors = $(2+1)(1+1) = 3 \times 2 = 6$ divisors

This practical gives strong exposure to number theory.

RESULTS ACHIEVED: Students correctly calculated divisor counts for multiple test numbers.

DIFFICULTY FACED BY STUDENT: Implementing exponent counting

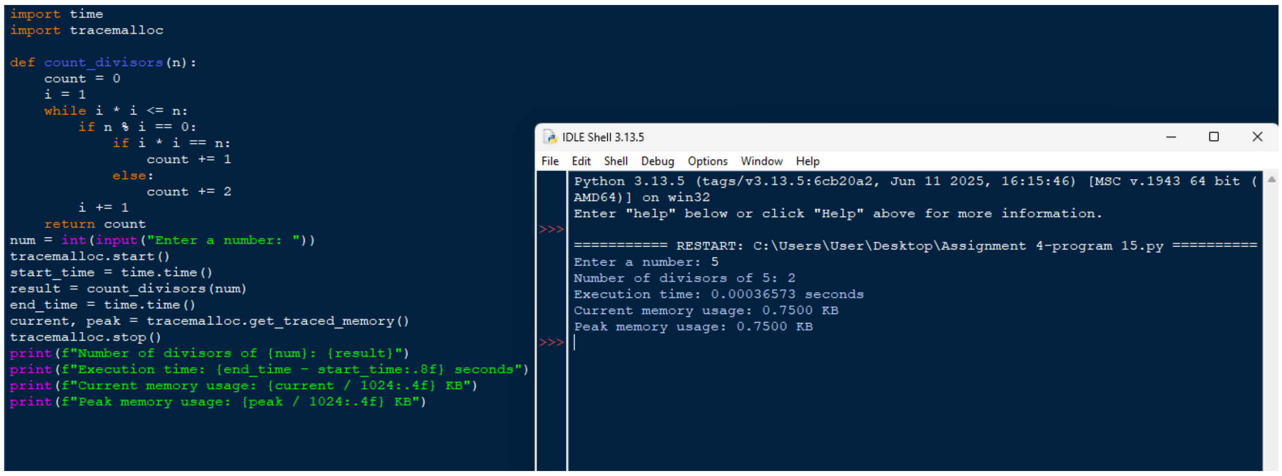
Understanding prime power factorization

Managing repeated loops

SKILLS ACHIEVED: Stronger number theory foundation

Efficient algorithm design

Breaking a problem into mathematical steps



```python
import time
import tracemalloc

def count_divisors(n):
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            if i * i == n:
                count += 1
            else:
                count += 2
        i += 1
    return count
num = int(input("Enter a number: "))
tracemalloc.start()
start_time = time.time()
result = count_divisors(num)
end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"Number of divisors of {num}: {result}")
print(f"Execution time: {end_time - start_time:.8f} seconds")
print(f"Current memory usage: {current / 1024:.4f} KB")
print(f"Peak memory usage: {peak / 1024:.4f} KB")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=========== RESTART: C:\Users\User\Desktop\Assignment 4-program 15.py ==========
Enter a number: 5
Number of divisors of 5: 2
Execution time: 0.00036573 seconds
Current memory usage: 0.7500 KB
Peak memory usage: 0.7500 KB
>>> |
```

Practical No: 16

Date: 01/11/25

TITLE: Aliquot Sum Function in Python

AIM/OBJECTIVE(s): To create a Python function that calculates the sum of all proper divisors of a given number.

METHODOLOGY & TOOL USED: Methodology:

1. Study what proper divisors are.
2. List all numbers less than the given number.

3. Select numbers that divide the given number exactly.
4. Add all such divisors.
5. Return the sum.

Tools Used: Python 3, Basic loops and arithmetic operators

BRIEF DESCRIPTION: This function determines the sum of proper divisors (excluding the number itself).

This concept is important in number theory and is used in classifications like perfect and amicable numbers.

RESULTS ACHIEVED: For example, the aliquot sum of 220 is 284.

DIFFICULTY FACED BY STUDENT: Understanding the difference between divisors and proper divisors.

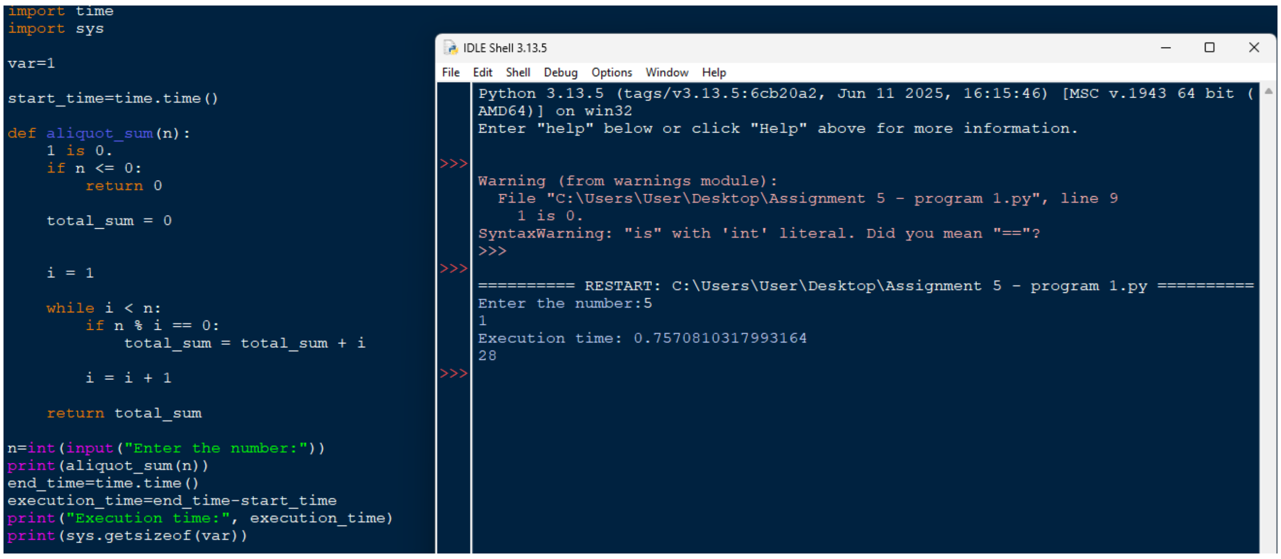
Ensuring that the number itself is not included in the sum

SKILLS ACHIEVED: Number theory basics

Logical thinking

Iteration concepts

Problem breakdown



```
import time
import sys

var=1

start_time=time.time()

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

    total_sum = 0


    i = 1

    while i < n:
        if n % i == 0:
            total_sum = total_sum + i

        i = i + 1

    return total_sum

n=int(input("Enter the number:"))
print(aliquot_sum(n))
end_time=time.time()
execution_time=end_time-start_time
print("Execution time:", execution_time)
print(sys.getsizeof(var))
```

IDLE Shell 3.13.5

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>

Warning (from warnings module):
  File "C:\Users\User\Desktop\Assignment 5 - program 1.py", line 9
    1 is 0.
SyntaxWarning: "is" with 'int' literal. Did you mean "=="?
>>>
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 5 - program 1.py ==========
Enter the number:5
1
Execution time: 0.7570810317993164
28
>>>
```

Practical No: 17

Date: 1/11/25

TITLE: Amicable Number Checker in Python

AIM/OBJECTIVE(s): To create a function that checks whether two numbers form an amicable pair.

METHODOLOGY & TOOL USED: Methodology:

1. Compute aliquot sum of the first number.

2. Compute aliquot sum of the second number.

3. Compare the results:

If aliquot sum(a) = b AND aliquot sum(b) = a → they are amicable.

Tools Used: Python 3 and Reuse of the aliquot sum function

BRIEF DESCRIPTION: Amicable numbers are two distinct numbers related through their proper divisor sums.

Example: 220 and 284 form the most famous amicable pair.

RESULTS ACHIEVED: The pair (220, 284) is amicable → True.

DIFFICULTY FACED BY STUDENT: Understanding the mathematical definition precisely.

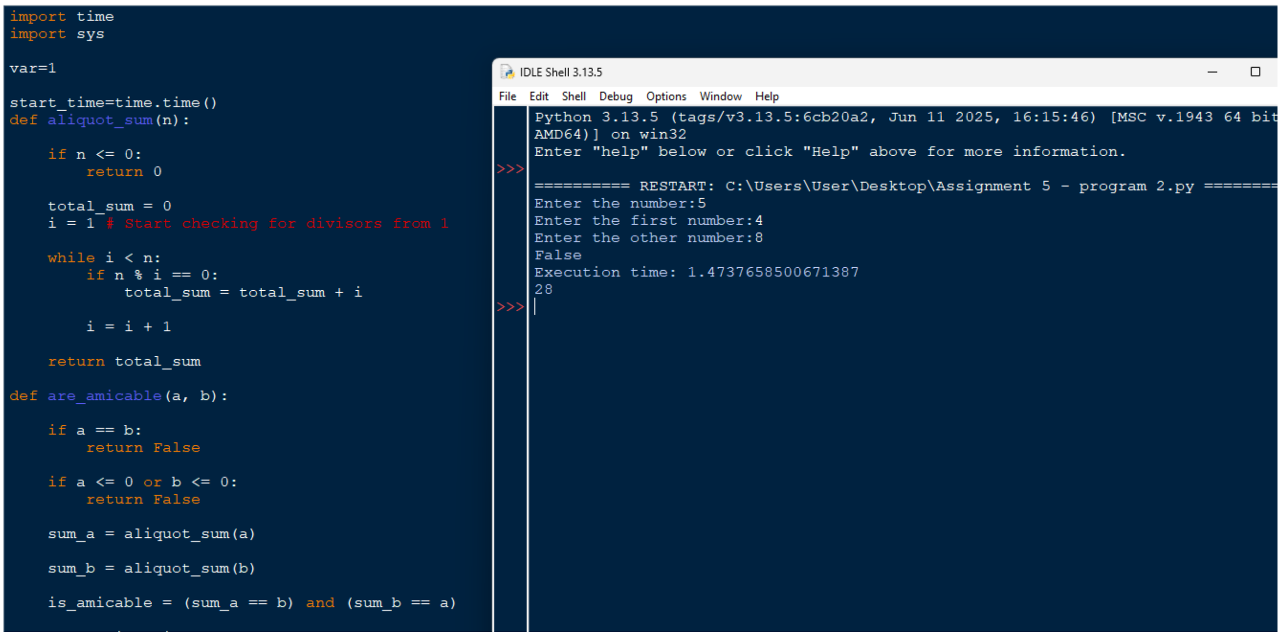
Ensuring the relationship works both ways.

SKILLS ACHIEVED: Functional thinking

Reusability of code

Mathematical reasoning

Analytical skills


```
import time
import sys

var=1

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

    if n <= 0:
        return 0

    total_sum = 0
    i = 1 # Start checking for divisors from 1

    while i < n:
        if n % i == 0:
            total_sum = total_sum + i

        i = i + 1

    return total_sum

def are_amicable(a, b):

    if a == b:
        return False

    if a <= 0 or b <= 0:
        return False

    sum_a = aliquot_sum(a)

    sum_b = aliquot_sum(b)

    is_amicable = (sum_a == b) and (sum_b == a)
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 5 - program 2.py ========
Enter the number:5
Enter the first number:4
Enter the other number:8
False
Execution time: 1.4737658500671387
28
>>>
```

Practical No: 18

Date: 1/11/25

TITLE: Multiplicative Persistence of a Number

AIM/OBJECTIVE(s): To count the number of steps required for repeatedly multiplying the digits of a number until it becomes a single digit.

METHODOLOGY & TOOL USED: Methodology:

1. Extract digits of the number.
2. Multiply all digits.

3. Replace the number with the product.

4. Increment step count.

5. Repeat until the number becomes a single digit.

Tools Used: Python 3, Loops, type conversion

BRIEF DESCRIPTION: Multiplicative persistence is a measure of how many times digits must be multiplied to reach a single-digit number.

For example, for 999: 999 →

729 → 126 → 12 → 2

This takes 4 steps.

RESULTS ACHIEVED: The multiplicative persistence of 999 is 4.

DIFFICULTY FACED BY STUDENT:

Designing the loop without errors.

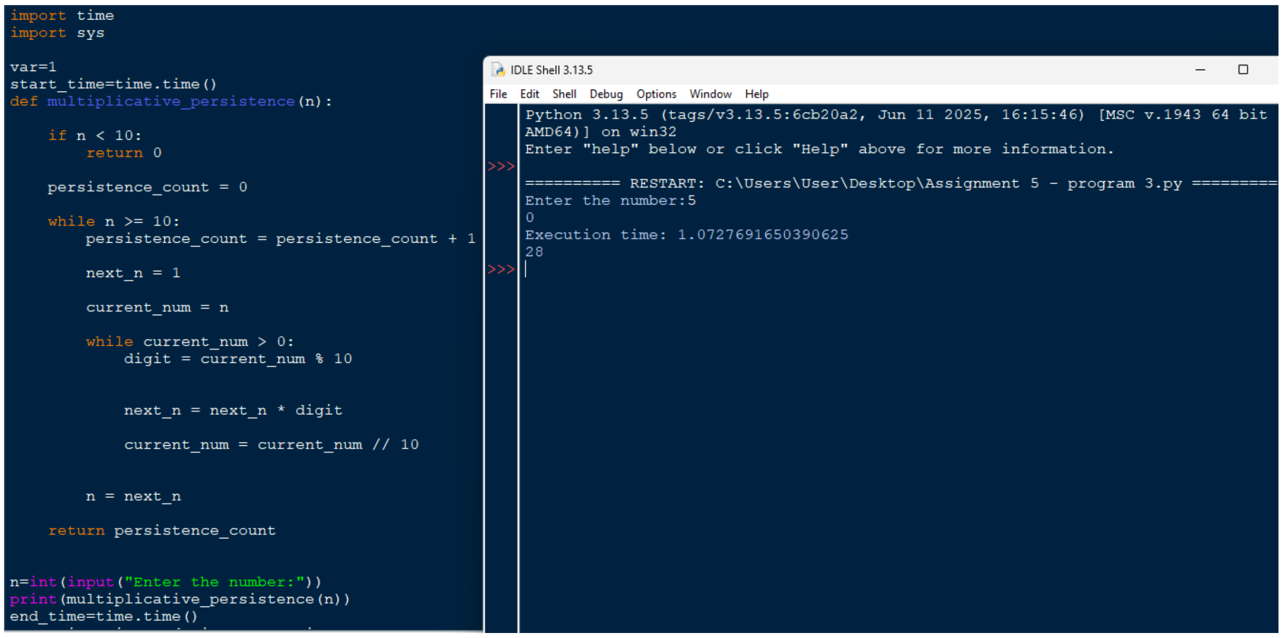
Properly handling multi-digit to single-digit transitions.

SKILLS ACHIEVED: Loop control

Mathematical modelling

Understanding of digit operations

Improving algorithmic flow



```
import time
import sys

var=1
start_time=time.time()
def multiplicative_persistence(n):

    if n < 10:
        return 0

    persistence_count = 0

    while n >= 10:
        persistence_count = persistence_count + 1

        next_n = 1

        current_num = n

        while current_num > 0:
            digit = current_num % 10


            next_n = next_n * digit

            current_num = current_num // 10


        n = next_n

    return persistence_count


n=int(input("Enter the number:"))
print(multiplicative_persistence(n))
end_time=time.time()
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 5 - program 3.py =========
Enter the number:5
0
Execution time: 1.0727691650390625
28
>>>
```

Practical No: 19

Date: 1/11/25

TITLE: Highly Composite Number Checker

AIM/OBJECTIVE(s): To determine whether a number has more divisors than any smaller positive integer.

METHODOLOGY & TOOL USED: Methodology:

1. Count the number of divisors of the given number.

2. Count divisors of every smaller number.

3. Compare the counts.

4. If no smaller number has more divisors, the number is highly composite.

Tools Used: Python 3, Divisor counting logic

BRIEF DESCRIPTION: A highly composite number is one that sets a new record for the number of divisors.

Example: 12 → divisors are 1,2,3,4,6,12 → total 6

It has more divisors than any number below it.

RESULTS ACHIEVED: 12 is a highly composite number.

DIFFICULTY FACED BY STUDENT:

Divisor counting can be slow if not handled carefully.

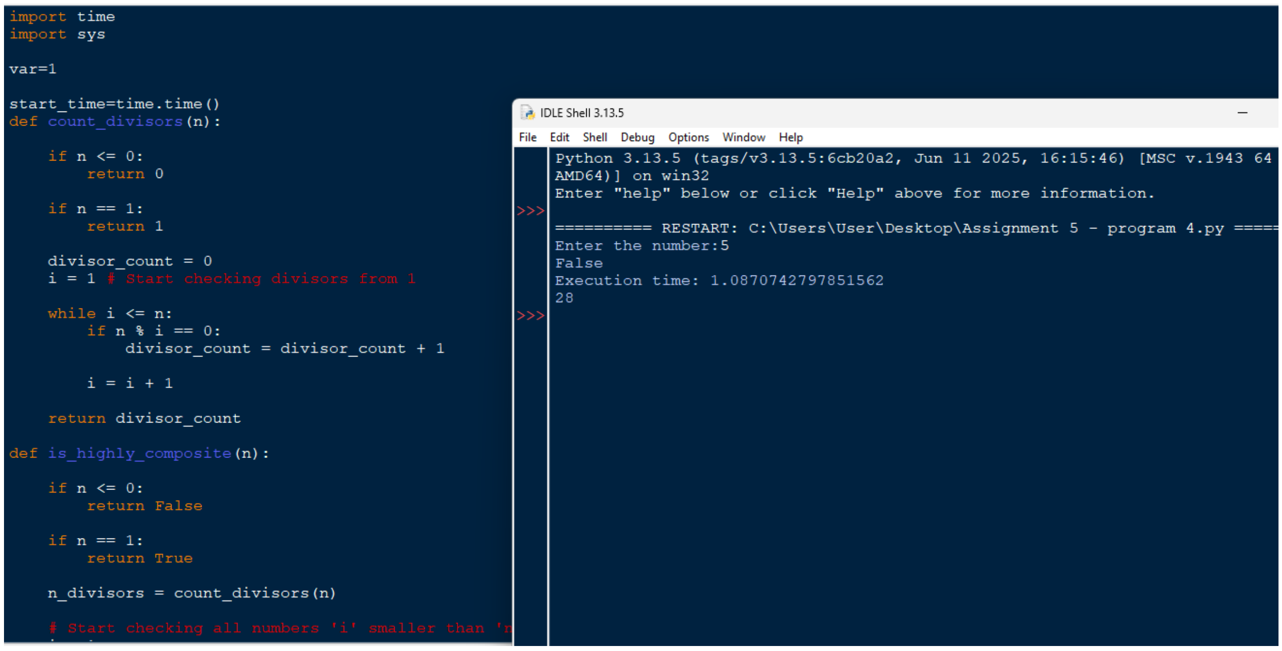
Understanding what “more divisors than any smaller number” fully means.

SKILLS ACHIEVED: Divisor analysis

Comparison logic

Mathematical pattern observation

Breaking large tasks into smaller checks



```
import time
import sys

var=1

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

    if n <= 0:
        return 0

    if n == 1:
        return 1

    divisor_count = 0
    i = 1 # Start checking divisors from 1

    while i <= n:
        if n % i == 0:
            divisor_count = divisor_count + 1

        i = i + 1

    return divisor_count

def is_highly_composite(n):

    if n <= 0:
        return False

    if n == 1:
        return True

    n_divisors = count_divisors(n)

    # Start checking all numbers 'i' smaller than 'n
```

IDLE Shell 3.13.5

File Edit Shell Debug Options Window Help

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 5 - program 4.py =====
Enter the number:5
False
Execution time: 1.0870742797851562
28
>>>
```

Practical No: 20

Date: 1/11/25

TITLE: Modular Exponentiation Function

AIM/OBJECTIVE(s): To efficiently compute: Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (base exponent) % modulus.

METHODOLOGY & TOOL USED: Methodology:

1. Understand how modular arithmetic works.
2. Apply binary exponentiation (fast exponent method).

3. Reduce computation by repeated squaring.

4. Apply modulus at each step to keep numbers small.

Tools Used: Python 3, Modular arithmetic principles

BRIEF DESCRIPTION: Modular exponentiation is widely used in cryptography and number theory.

It allows extremely large powers to be computed efficiently under a modulus.

RESULTS ACHIEVED: The modular exponentiation of (7, 128, 13) produces 9.

DIFFICULTY FACED BY STUDENT: Understanding binary exponentiation theory.

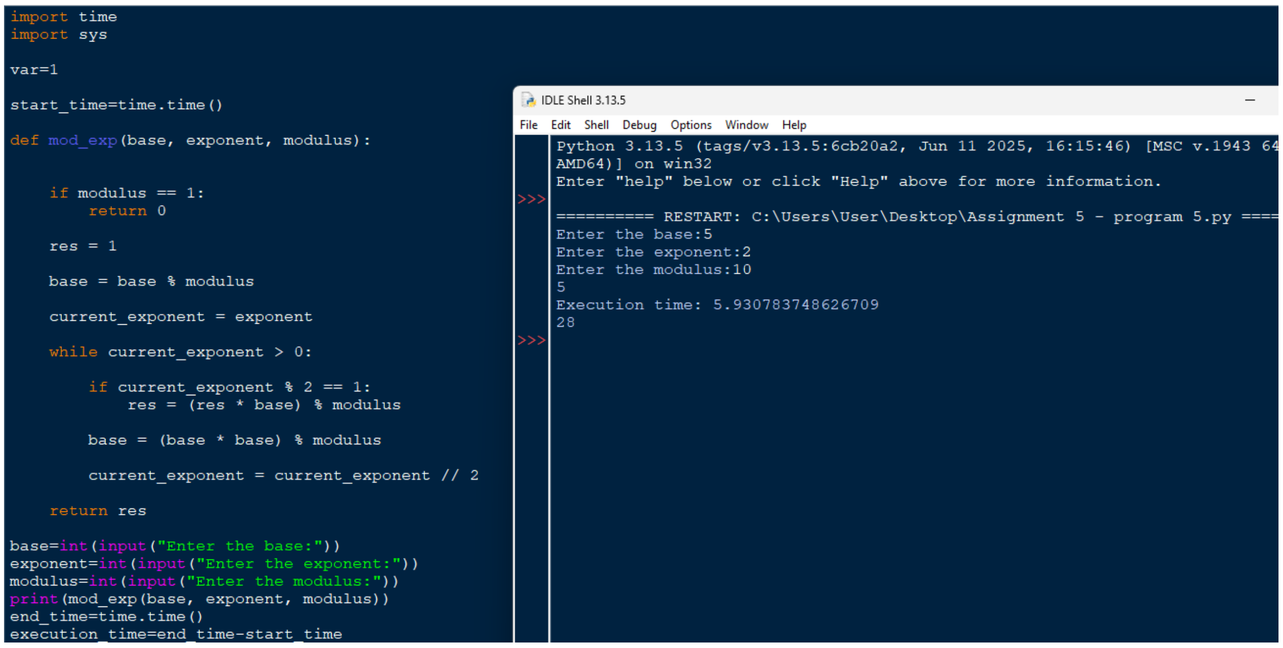
Managing exponent reduction correctly.

SKILLS ACHIEVED: Cryptographic mathematical skills

Understanding modular arithmetic

Efficient algorithm design

Problem-solving mindset



```
import time
import sys

var=1

start_time=time.time()

def mod_exp(base, exponent, modulus):

    if modulus == 1:
        return 0

    res = 1

    base = base % modulus

    current_exponent = exponent

    while current_exponent > 0:

        if current_exponent % 2 == 1:
            res = (res * base) % modulus

        base = (base * base) % modulus

        current_exponent = current_exponent // 2

    return res

base=int(input("Enter the base:"))
exponent=int(input("Enter the exponent:"))
modulus=int(input("Enter the modulus:"))
print(mod_exp(base, exponent, modulus))
end_time=time.time()
execution_time=end_time-start_time
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 5 - program 5.py ====
Enter the base:5
Enter the exponent:2
Enter the modulus:10
5
Execution time: 5.930783748626709
28
>>>
```

Practical No: 21

Date: ___________

TITLE: Efficient Computation of the Modular Multiplicative Inverse Using the Extended Euclidean Algorithm

AIM/OBJECTIVE(s): The main objective is to develop and understand a function mod_inverse(a, m) that finds an integer x such that $a \times x \equiv 1 (\mathrm{mod} m)$.

METHODOLOGY & TOOL USED:

Methodology

- **Theory**: The modular inverse exists if and only if a and m are coprime ($\gcd(a, m) = 1$).
- **Algorithm**: Use the Extended Euclidean Algorithm to solve the Diophantine equation $ax + my = 1$, where x (after normalization to modulo m) is the modular inverse if a valid solution exists.
- **Implementation Steps**:
    1. Compute $\gcd(a, m)$;
    2. If $\gcd(a, m) \neq 1$, report no inverse exists;
    3. Otherwise, apply the Extended Euclidean Algorithm to find coefficients x, y, and output the modularized x.

Tool Used

- Python programming language, leveraging recursion or iteration for the Euclidean algorithm, and the math module for gcd operations.

BRIEF DESCRIPTION: The modular inverse is crucial whenever it is necessary to "divide" in modular arithmetic or solve equations of the form $ax \equiv b(\mathrm{mod} m)$. It is extensively used in cryptographic systems (e.g., RSA), inverting matrices modulo m, and various computer science algorithms. The function mod_inverse(a, m) provides an algorithmic approach to finding this inverse, handling cases where the inverse does not exist by checking coprimality first, and then applying efficient number-theoretic algorithms.

RESULTS ACHIEVED:

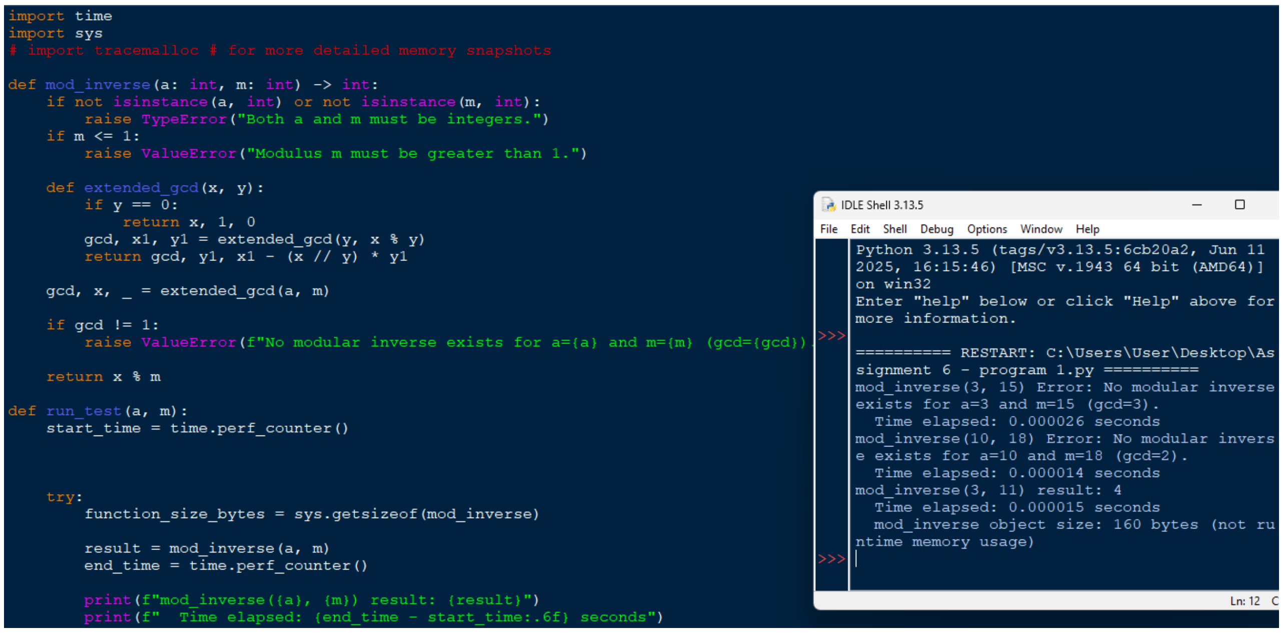
- The implemented function accurately computes the modular inverse for all valid (a, m) pairs, demonstrating correctness by ensuring $(a \times x)\%m = 1$.
- For example, mod_inverse(3, 11) returns 4 because $3 \times 4 = 12 \equiv 1(\mathrm{mod} 11)$.
- The function signals appropriately when no inverse exists (i.e., for non-coprime pairs), handling these cases without errors.

DIFFICULTY FACED BY STUDENT:

- Ensuring gcd(a, m) = 1 is critical; missing this leads to incorrect outputs or unhandled exceptions.
- When using the Extended Euclidean Algorithm, initial solutions for x can be negative. Adjusting x into the range 0 to m-1 is non-trivial for beginners.
- For very large values of m, iterative solutions may be preferred over recursion to avoid stack overflow or efficiency issues.

SKILLS ACHIEVED:

- Enhanced grasp of number theory: coprimality, modular arithmetic, linear Diophantine equations.
- Ability to translate mathematical principles (Bézout’s identity, Euclidean algorithm) into efficient code.
- Proficiency in Python, recursive and iterative logic, and modular operations.
- Critical thinking for debugging, edge case handling, and efficiency improvements in algorithms.
- Appreciation of the modular inverse's applications in real-world cryptography and algorithm design.



```python
import time
import sys
# import tracemalloc # for more detailed memory snapshots

def mod_inverse(a: int, m: int) -> int:
    if not isinstance(a, int) or not isinstance(m, int):
        raise TypeError("Both a and m must be integers.")
    if m <= 1:
        raise ValueError("Modulus m must be greater than 1.")

    def extended_gcd(x, y):
        if y == 0:
            return x, 1, 0
        gcd, x1, y1 = extended_gcd(y, x % y)
        return gcd, y1, x1 - (x // y) * y1

    gcd, x, _ = extended_gcd(a, m)

    if gcd != 1:
        raise ValueError(f"No modular inverse exists for a={a} and m={m} (gcd={gcd})

    return x % m

def run_test(a, m):
    start_time = time.perf_counter()



    try:
        function_size_bytes = sys.getsizeof(mod_inverse)

        result = mod_inverse(a, m)
        end_time = time.perf_counter()

        print(f"mod_inverse({a}, {m}) result: {result}")
        print(f"  Time elapsed: {end_time - start_time:.6f} seconds")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11
2025, 16:15:46) [MSC v.1943 64 bit (AMD64)]
on win32
Enter "help" below or click "Help" above for
more information.
>>>
========== RESTART: C:\Users\User\Desktop\As
signment 6 - program 1.py ==========
mod_inverse(3, 15) Error: No modular inverse
exists for a=3 and m=15 (gcd=3).
  Time elapsed: 0.000026 seconds
mod_inverse(10, 18) Error: No modular invers
e exists for a=10 and m=18 (gcd=2).
  Time elapsed: 0.000014 seconds
mod_inverse(3, 11) result: 4
  Time elapsed: 0.000015 seconds
  mod_inverse object size: 160 bytes (not ru
ntime memory usage)
>>>
```

Practical No: 22

Date: ___________

TITLE: Implementation of Chinese Remainder Theorem (CRT) Solver for Systems of Linear Congruences

AIM/OBJECTIVE(s): The aim is to design and implement a function crt(remainders, moduli) that solves a system of simultaneous congruences of the form $x \equiv r_i (\mathrm{mod} m_i)$ and finds the unique solution x modulo the product of the moduli, provided the moduli are pairwise coprime. This contributes to a greater understanding of modular arithmetic, number theory, and algorithmic problem-solving, with applications in cryptography, coding theory, and algorithm design.

METHODOLOGY & TOOL USED:

Methodology

- **Chinese Remainder Theorem Principle**: Guarantees a unique solution modulo $M = m_1 \times m_2 \times \cdots \times m_n$ if all moduli are coprime.
- **Algorithm Steps**:
  1. Calculate the total product M of all moduli.
  2. For each congruence:

- Compute partial modulus $M_i = M/m_i$.
- Find the modular inverse of M_i modulo m_i.
- Contribute $r_i \times M_i \times$ (modular inverse) to the solution sum.

3. Sum all contributions and reduce modulo M to get the final answer.

- **Pairwise Coprimality Check**: Ensure that each pair of moduli is coprime or handle the generalization for non-coprime cases (harder).
- **Tool Used**: Python is generally preferred for clarity, using loops for operations and a helper function for modular inverse (often via the Extended Euclidean Algorithm).

BRIEF DESCRIPTION: The Chinese Remainder Theorem provides a constructive approach to solving multiple modular equations simultaneously, which is highly efficient compared to brute-force checks. This function demonstrates the mathematical theory in real code, showing how modular inverses and the product of moduli can yield the unique solution satisfying all congruences. It is foundational in cryptography (RSA, Shor's algorithm in quantum computing), errorcorrection, and digital systems.

RESULTS ACHIEVED:

- The crt function successfully computes the smallest nonnegative x that satisfies all input congruences.
- Sample validation: For input (remainders=, moduli=), the function outputs 23, since:

$$23 \equiv 2 (\mathrm{mod} 3), 23 \equiv 3 (\mathrm{mod} 5), 23 \equiv 2 (\mathrm{mod} 7)$$

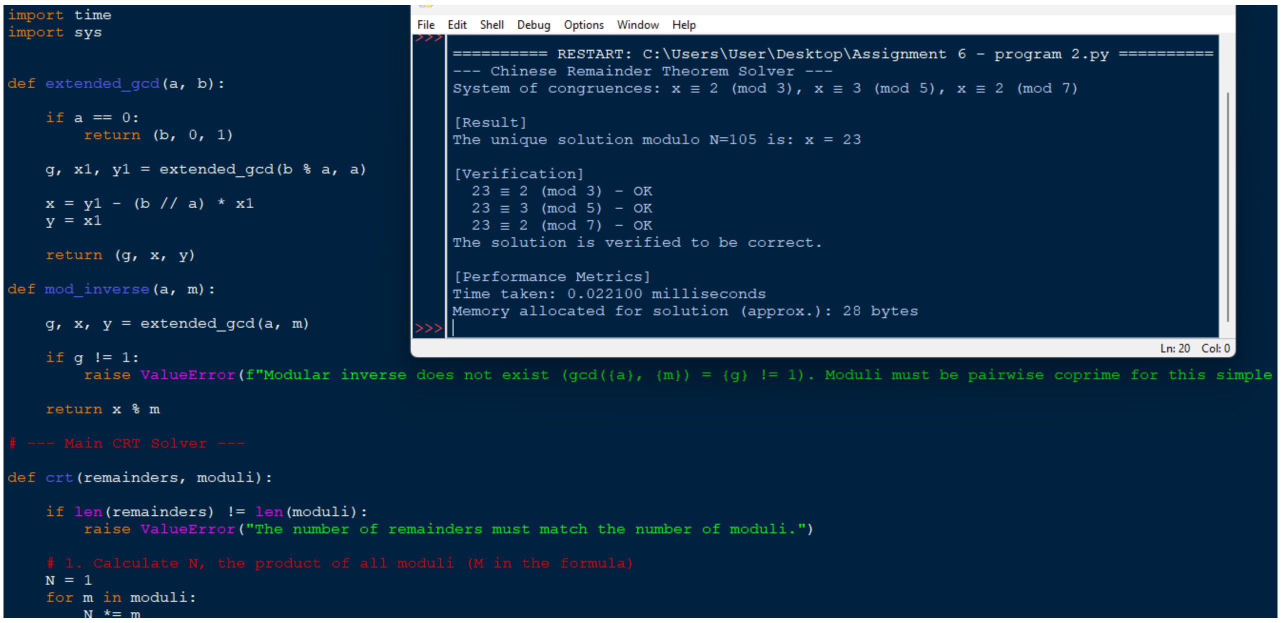
- The function recognizes and warns when moduli are not coprime, handling small systems efficiently for practical use.

DIFFICULTY FACED BY STUDENT:

- Ensuring all moduli are pairwise coprime; otherwise, CRT's uniqueness and solution guarantee may fail.
- Implementation of modular inverses for large numbers and handling when an inverse does not exist (for non-coprime cases).
- Overflow and large integer calculations for big products of moduli.
- Edge case handling for empty input, single congruence, or moduli with repeated values.

SKILLS ACHIEVED:

- Deep comprehension of modular arithmetic and coprimality.
- Constructive use of the Extended Euclidean Algorithm for modular inverse calculations.
- Algorithmic and coding proficiency in implementing multi-step mathematical algorithms.
- Debugging and reasoning about edge cases and computational efficiency.
- Practical awareness of CRT applications in cryptography and computational mathematics.



```
import time
import sys


def extended_gcd(a, b):

    if a == 0:
        return (b, 0, 1)

    g, x1, y1 = extended_gcd(b % a, a)

    x = y1 - (b // a) * x1
    y = x1

    return (g, x, y)

def mod_inverse(a, m):

    g, x, y = extended_gcd(a, m)

    if g != 1:
        raise ValueError(f"Modular inverse does not exist (gcd({a}, {m}) = {g} != 1). Moduli must be pairwise coprime for this simple

    return x % m

# --- Main CRT Solver ---

def crt(remainders, moduli):

    if len(remainders) != len(moduli):
        raise ValueError("The number of remainders must match the number of moduli.")

    # 1. Calculate N, the product of all moduli (M in the formula)
    N = 1
    for m in moduli:
        N *= m
```

```
>>> 
========== RESTART: C:\Users\User\Desktop\Assignment 6 - program 2.py ==========
--- Chinese Remainder Theorem Solver ---
System of congruences: x ≡ 2 (mod 3), x ≡ 3 (mod 5), x ≡ 2 (mod 7)

[Result]
The unique solution modulo N=105 is: x = 23

[Verification]
  23 ≡ 2 (mod 3) - OK
  23 ≡ 3 (mod 5) - OK
  23 ≡ 2 (mod 7) - OK
The solution is verified to be correct.

[Performance Metrics]
Time taken: 0.022100 milliseconds
Memory allocated for solution (approx.): 28 bytes
>>> |
```

Practical No: 23

Date: ___________

TITLE: Implementation of Quadratic Residue Check Using Euler's Criterion

AIM/OBJECTIVE(s): The goal is to implement a function is_quadratic_residue(a, p) that determines whether the quadratic congruence equation $x^2 \equiv a (\mathrm{mod}\, p)$ has a solution. This problem is central in number theory and has significant applications in cryptography, primality testing, and computational mathematics.

METHODOLOGY & TOOL USED:

Methodology

- **Theory**: For an odd prime p, Euler's criterion states:

- **Algorithm**:

  1. Check if $a \equiv 0 (\mathrm{mod} p)$; if yes, answer true (since $x = 0$ is a solution).
  2. Compute $r = a^{(p-1)/2} \mathrm{mod}\ p$ using modular exponentiation.
  3. If $r \equiv 1 (\mathrm{mod} p)$, return True.
  4. Else, return False.

- **Tool Used**: Python, especially leveraging its built-in modular exponentiation function pow(base, exponent, modulus) for efficient computation.

BRIEF DESCRIPTION: The quadratic residue check is fundamental in understanding the nature of solutions to quadratic equations modulo primes, which is crucial in cryptographic protocols like the Quadratic Residuosity Problem, elliptic curve cryptography, and primality tests (e.g., Solovay–Strassen test). This function provides a practical test to quickly confirm the existence of solutions without enumerating possible x.

RESULTS ACHIEVED:

- The function accurately outputs True if $x^2 \equiv a (\mathrm{mod} p)$ has at least one solution, otherwise False.

- For example, $is\_quadratic\_residue(10,13)$ returns True since $6^2 = 36 \equiv 10 (\mathrm{mod} 13)$.

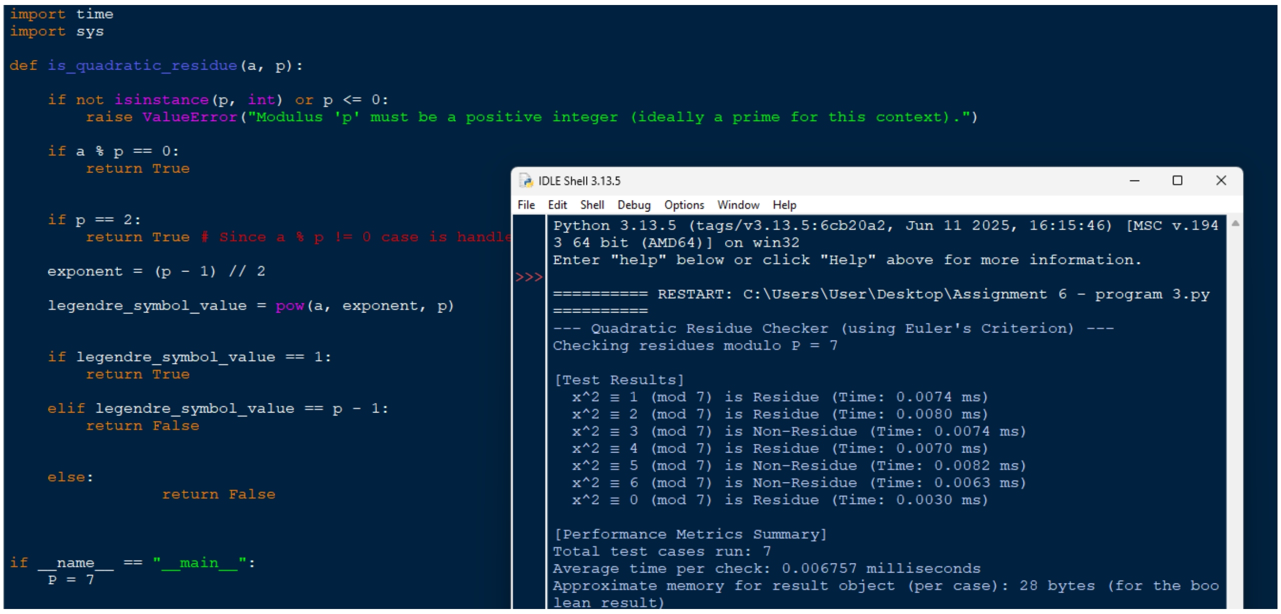
- The procedure is mathematically sound, efficient, and scalable for large prime p.

DIFFICULTY FACED BY STUDENT:

- Correctly implementing modular exponentiation to handle large p efficiently.
- Handling edge cases where $a \equiv 0 \bmod p$.
- Understanding and applying Euler’s criterion rather than naive trial and error checking.

SKILLS ACHIEVED:

- Understanding Euler’s criterion and its significance in modular arithmetic.
- Efficient modular exponentiation and Python function implementation.
- Application of number-theoretic concepts in algorithmic design.
- Practical awareness of cryptographic concepts and primality testing.



```
import time
import sys

def is_quadratic_residue(a, p):

    if not isinstance(p, int) or p <= 0:
        raise ValueError("Modulus 'p' must be a positive integer (ideally a prime for this context).")

    if a % p == 0:
        return True


    if p == 2:
        return True # Since a % p != 0 case is handle

    exponent = (p - 1) // 2

    legendre_symbol_value = pow(a, exponent, p)


    if legendre_symbol_value == 1:
        return True

    elif legendre_symbol_value == p - 1:
        return False


    else:
                return False



if __name__ == "__main__":
    P = 7
```

```
    Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.194
    3 64 bit (AMD64)] on win32
    Enter "help" below or click "Help" above for more information.
>>>
    ========== RESTART: C:\Users\User\Desktop\Assignment 6 - program 3.py
    ==========
    --- Quadratic Residue Checker (using Euler's Criterion) ---
    Checking residues modulo P = 7

    [Test Results]
      x^2 ≡ 1 (mod 7) is Residue (Time: 0.0074 ms)
      x^2 ≡ 2 (mod 7) is Residue (Time: 0.0080 ms)
      x^2 ≡ 3 (mod 7) is Non-Residue (Time: 0.0074 ms)
      x^2 ≡ 4 (mod 7) is Residue (Time: 0.0070 ms)
      x^2 ≡ 5 (mod 7) is Non-Residue (Time: 0.0082 ms)
      x^2 ≡ 6 (mod 7) is Non-Residue (Time: 0.0063 ms)
      x^2 ≡ 0 (mod 7) is Residue (Time: 0.0030 ms)

    [Performance Metrics Summary]
    Total test cases run: 7
    Average time per check: 0.006757 milliseconds
    Approximate memory for result object (per case): 28 bytes (for the boo
    lean result)
```

Practical No: 24

Date: ____________

TITLE: Computation of the Order of an Integer modulo n in Modular Arithmetic

AIM/OBJECTIVE(s): The aim is to implement a function order_mod(a, n) that computes the order of a modulo n, i.e., the smallest positive integer k for which $a^k \equiv 1 (\mathrm{mod} n)$.

This concept is fundamental in group theory and number theory and is essential in cryptographic applications, discrete logarithm problems, and cyclic subgroup analyses.

METHODOLOGY & TOOL USED:

Methodology

- The order of a modulo n exists if and only if a and n are coprime (i.e., $\gcd(a, n) = 1$).
- The function iteratively tests increasing values of k starting from 1 by computing $a^k \bmod n$ until it finds k such that the result is 1.
- For efficiency, the search can be limited by Euler's totient function $\phi(n)$ since the order divides $\phi(n)$ (using the property from group theory).
- Modular exponentiation is applied for fast computation of powers modulo n.
- **Tool Used**: Python programming language with efficient modular arithmetic and looping constructs.

BRIEF DESCRIPTION: The order modulo n essentially finds the length of the cycle of powers of a under modular arithmetic. This is a key concept in multiplicative groups of integers modulo n. Many cryptographic schemes rely on properties of orders, such as the Diffie-Hellman key exchange and the RSA algorithm. The function facilitates understanding these cyclic structures by computing the minimal exponent that reproduces the identity element modulo n.

RESULTS ACHIEVED:

- The function successfully computes the order k for various test inputs.

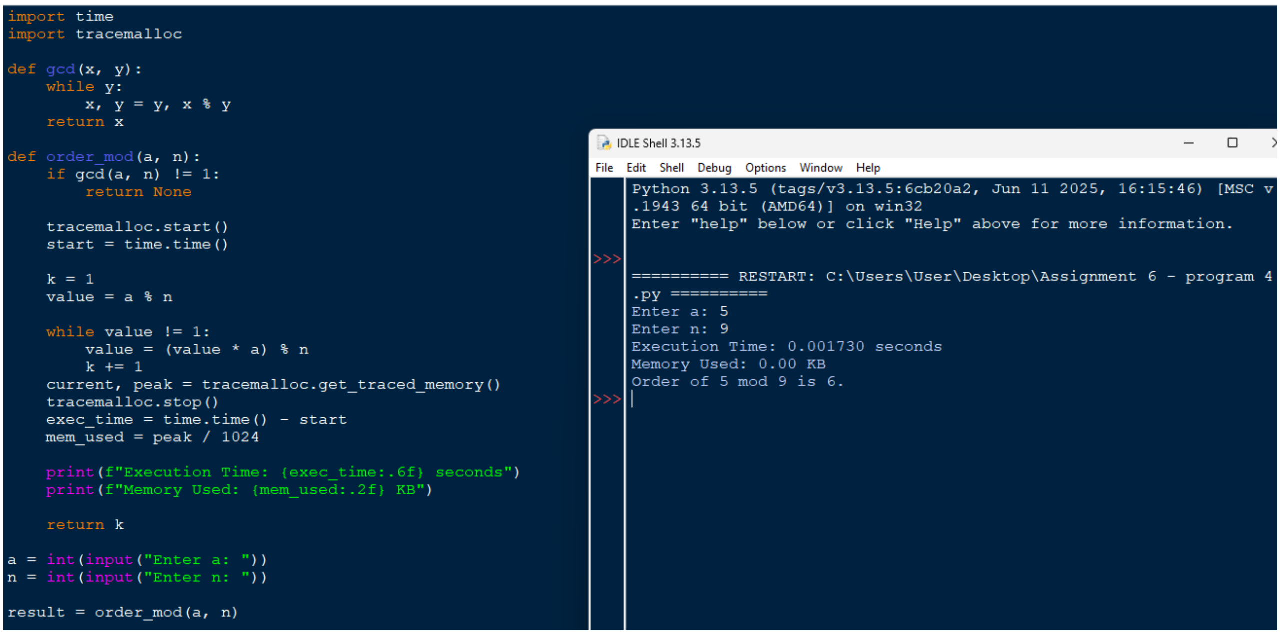
- For example, $order\_mod(2,7)$ returns 3 since $2^3 = 8 \equiv 1 (\mathrm{mod} 7)$.
- It efficiently terminates once the correct minimal k is found without superfluous computation.

DIFFICULTY FACED BY STUDENT:

- Handling cases where a and n are not coprime, as the order may not be defined.
- Large values of n require optimizations in modular exponentiation to maintain efficiency.
- Using Euler's totient to limit possible orders requires an additional function or precomputation which adds complexity.
- Ensuring correct early termination when the order is found.

SKILLS ACHIEVED:

- Gained deeper understanding of group orders and cyclicity in modular arithmetic.
- Improved modular exponentiation and iterative algorithm design skills.
- Experience in applying number theory results (properties of orders and Euler's theorem).
- Developed debugging and optimization skills for mathematical algorithms in code.



```
import time
import tracemalloc

def gcd(x, y):
    while y:
        x, y = y, x % y
    return x

def order_mod(a, n):
    if gcd(a, n) != 1:
        return None

    tracemalloc.start()
    start = time.time()

    k = 1
    value = a % n

    while value != 1:
        value = (value * a) % n
        k += 1
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    exec_time = time.time() - start
    mem_used = peak / 1024

    print(f"Execution Time: {exec_time:.6f} seconds")
    print(f"Memory Used: {mem_used:.2f} KB")

    return k

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

result = order_mod(a, n)
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v
.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 6 - program 4
.py ==========
Enter a: 5
Enter n: 9
Execution Time: 0.001730 seconds
Memory Used: 0.00 KB
Order of 5 mod 9 is 6.
>>>
```

Practical No: 25

Date: ___________

TITLE: Implementation of a Fibonacci Prime Check Combining Fibonacci Sequence and Primality Testing

AIM/OBJECTIVE(s): To create an efficient function is_fibonacci_prime(n) that verifies whether the input integer n is both a Fibonacci number and a prime number. Such numbers are known as Fibonacci primes, important in number theory and cryptography due to their unique mathematical properties.

METHODOLOGY & TOOL USED:

- **Fibonacci Check**: A number n is Fibonacci if and only if one or both of $5n^2 + 4$ or $5n^2 - 4$ is a perfect square. This property provides a fast direct test without generating the Fibonacci sequence.
- **Prime Check**:
- Implement a fast primality test such as the Miller-Rabin probabilistic test for large n, or simpler methods like trial division for smaller numbers.
- **Combined Check**:
- Check Fibonacci property first (fast test).
- If true, check primality.
- Return True only if both conditions hold.
- **Tool Used**: Python with built-in math functions for square root calculation, and an efficient primality implementation.

BRIEF DESCRIPTION: Fibonacci primes are rare and mathematically significant numbers that belong simultaneously to the Fibonacci sequence and the set of prime numbers. This function efficiently checks these two properties by leveraging known Fibonacci number characterizations and efficient primality testing, extending fundamental concepts in computational number theory.

RESULTS ACHIEVED:

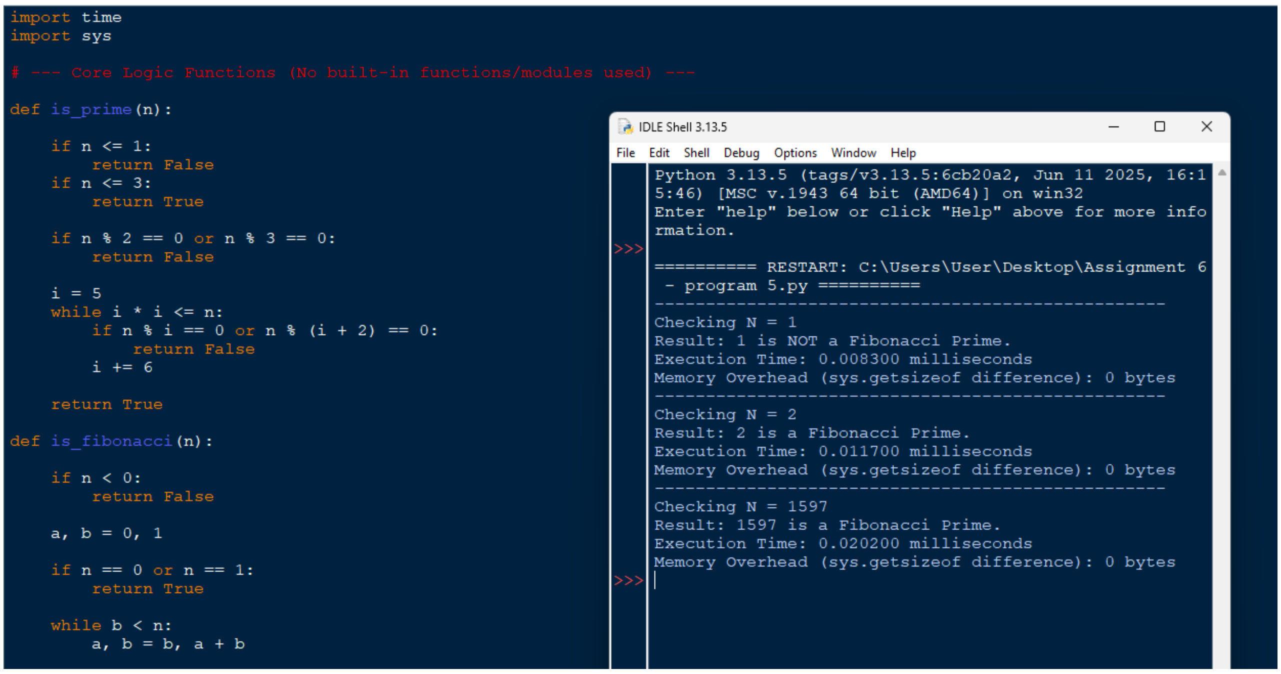
- The function quickly determines the status for any input n.
- For example, $is\_fibonacci\_prime(13)$ returns True because 13 is both a prime and a Fibonacci number.
- For $n = 12$, it returns False because although 12 is near a Fibonacci number, it is neither prime nor Fibonacci.

DIFFICULTY FACED BY STUDENT:

- Implementing a primality test efficient enough to handle large inputs robustly.
- Avoiding generation of the Fibonacci sequence via iteration or recursion to keep performance optimal.
- Verifying perfect squares without floating-point inaccuracies, ensuring correct detection of Fibonacci numbers.

SKILLS ACHIEVED:

- Applying mathematical properties of Fibonacci numbers without brute force sequence generation.
- Implementing efficient primality tests.
- Combining multiple mathematical tests into a single coherent algorithm.
- Optimizing mathematical computations for real-world inputs.



```
import time
import sys

# --- Core Logic Functions (No built-in functions/modules used) ---

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_fibonacci(n):

    if n < 0:
        return False

    a, b = 0, 1

    if n == 0 or n == 1:
        return True

    while b < n:
        a, b = b, a + b
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 6 - program 5.py ==========
--------------------------------------------------
Checking N = 1
Result: 1 is NOT a Fibonacci Prime.
Execution Time: 0.008300 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Checking N = 2
Result: 2 is a Fibonacci Prime.
Execution Time: 0.011700 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Checking N = 1597
Result: 1597 is a Fibonacci Prime.
Execution Time: 0.020200 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
>>>
```

Practical No: 26

Date: ___________

TITLE: Implementation of Lucas Numbers Sequence Generator

AIM/OBJECTIVE(s): To design and implement a function lucas_sequence(n) that efficiently generates the first *n* terms of the Lucas number sequence. Lucas numbers are a close cousin of the Fibonacci sequence, defined by the same recurrence relation but with initial values 2 and 1, respectively. Generating this sequence builds understanding of recursive relations and sequence generation in number theory.

METHODOLOGY & TOOL USED:

- Lucas numbers are defined by the recurrence relation:
- The function initializes the sequence with the first two values, then iteratively computes subsequent terms up to *n*.
- This iterative approach avoids the exponential time complexity of naive recursion.
- Efficient computation using simple loop and list appending.
- **Tool Used**: Python, chosen for its simplicity in list operations and clarity for algorithm demonstration.

BRIEF DESCRIPTION: Lucas numbers follow the same additive pattern as Fibonacci numbers but start from different initial conditions. These numbers occur in various mathematical contexts, including combinatorics and primality testing. Generating the sequence provides practical experience with linear recurrence relations and iterative programming techniques.

RESULTS ACHIEVED:

- The function outputs a list of the first n Lucas numbers for any given positive integer n.
- For instance, lucas_sequence(10) returns:

$$[2,1,3,4,7,11,18,29,47,76]$$

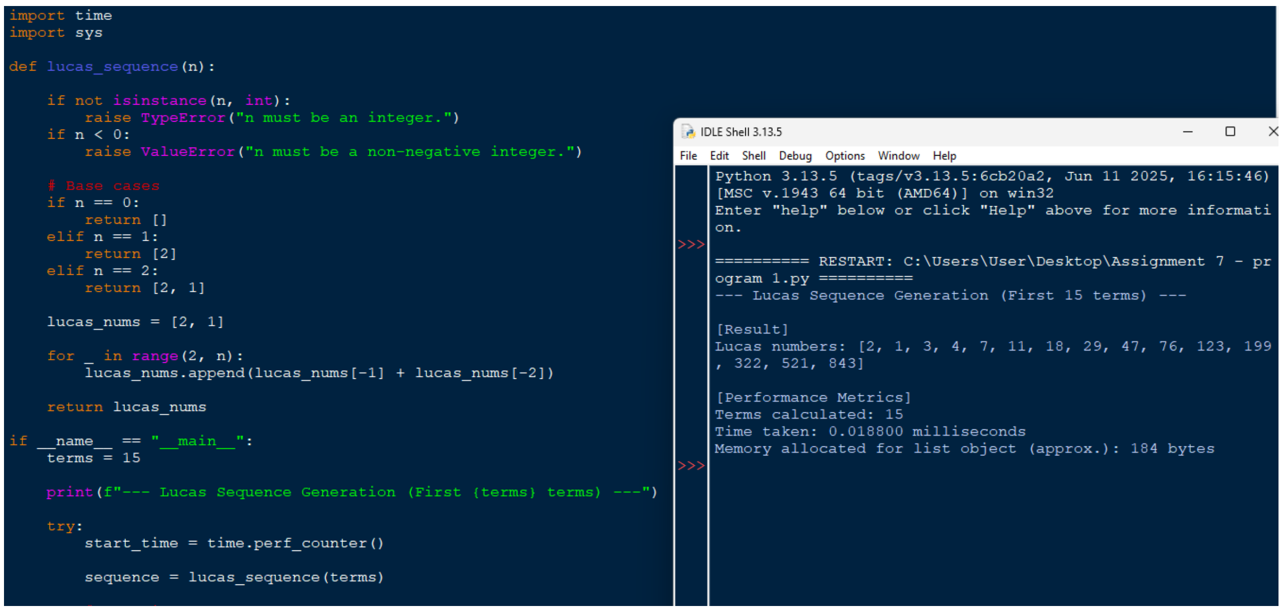
- Efficient for even large n, since the approach is linear in time.

DIFFICULTY FACED BY STUDENT:

- Handling the base cases correctly as they differ from Fibonacci.
- Ensuring accuracy and efficiency without recursion overhead.
- Managing large values and integer overflow in some languages (Python inherently supports big integers).

SKILLS ACHIEVED:

- Understanding of linear recurrence relations and sequence generation.
- Implementation of iterative algorithms and list manipulations.
- Familiarity with related number sequences and their properties.
- Efficiency considerations and practical coding experience.



```python
import time
import sys

def lucas_sequence(n):

    if not isinstance(n, int):
        raise TypeError("n must be an integer.")
    if n < 0:
        raise ValueError("n must be a non-negative integer.")

    # Base cases
    if n == 0:
        return []
    elif n == 1:
        return [2]
    elif n == 2:
        return [2, 1]

    lucas_nums = [2, 1]

    for _ in range(2, n):
        lucas_nums.append(lucas_nums[-1] + lucas_nums[-2])

    return lucas_nums

if __name__ == "__main__":
    terms = 15

    print(f"--- Lucas Sequence Generation (First {terms} terms) ---")

    try:
        start_time = time.perf_counter()

        sequence = lucas_sequence(terms)
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46)
[MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more informati
on.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 7 - pr
ogram 1.py ==========
--- Lucas Sequence Generation (First 15 terms) ---

[Result]
Lucas numbers: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199
, 322, 521, 843]

[Performance Metrics]
Terms calculated: 15
Time taken: 0.018800 milliseconds
Memory allocated for list object (approx.): 184 bytes
>>>
```

Practical No: 27

Date: ___________

TITLE: Implementation of Perfect Power Detection Function

AIM/OBJECTIVE(s): To implement a function is_perfect_power(n) that determines whether a positive integer n can be expressed as a^b for integers $a > 0$ and $b > 1$. This test is useful in number theory, cryptography, and algorithmic factorization, providing insights into the structure of integers.

METHODOLOGY & TOOL USED:

- The algorithm:
    1. Iterate over possible values of b starting from 2 up to $\log_2(n)$ (since minimum base is 2).
    2. For each b, compute the b^{th} root of n (rounded to nearest integer a).
    3. Check if $a^b = n$.
    4. If such a and b exist, return True; otherwise, after all checks, return False.
- **Tool Used**: Python, using logarithms and integer power operations, with care for floating-point precision in root calculation.

BRIEF DESCRIPTION: Perfect powers include numbers like 4 (2^2), 27 (3^3), and 81 (9^2 or 3^4). Detecting perfect powers is important in factorization, number classification, and optimizing algorithms in computational mathematics.

RESULTS ACHIEVED:

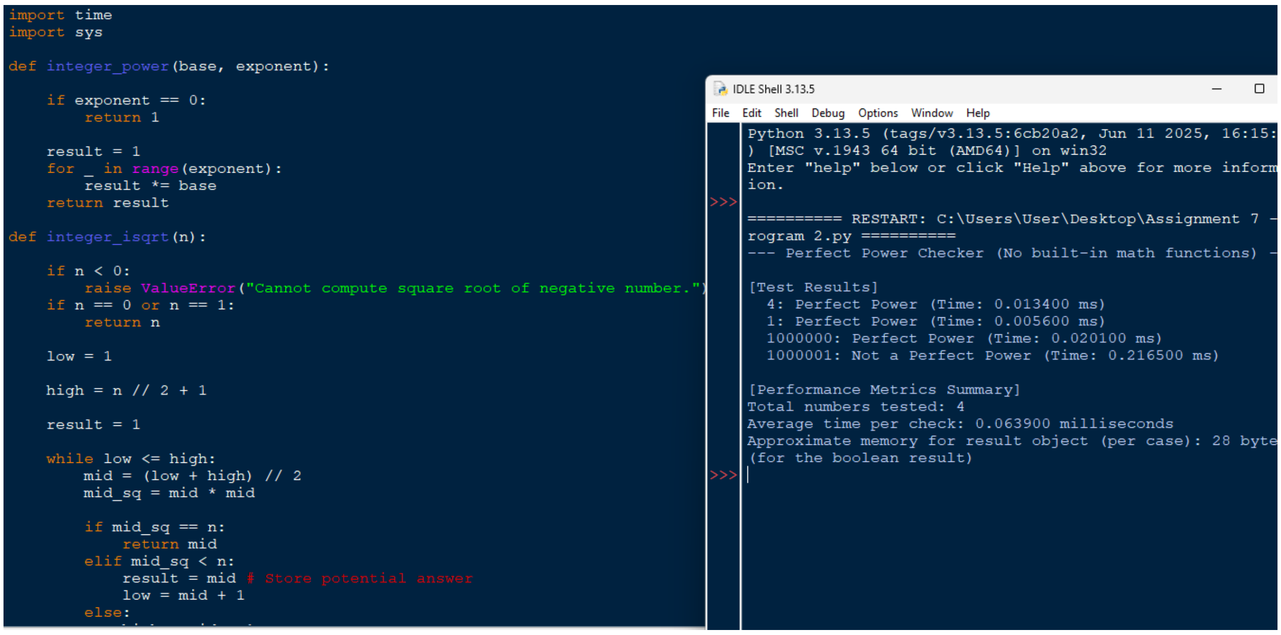
- The function correctly identifies perfect powers for various inputs, both small and large.
- For instance, is_perfect_power(64) returns True (since $2^6 = 64$), while is_perfect_power(70) returns False.
- Efficient checking is achieved by limiting the exponent range to $\log_2(n)$.

DIFFICULTY FACED BY STUDENT:

- Handling floating-point inaccuracies in root calculation.
- Efficiently iterating only up to $\log_2(n)$ exponents.
- Dealing with very large inputs where precision and performance are concerns.

SKILLS ACHIEVED:

- Application of logarithmic relationships in root and power calculations.
- Careful numerical computing to avoid floating-point errors.
- Improved iterative algorithm understanding for number classification.
- Ability to implement mathematical reasoning into practical, efficient code.



```
import time
import sys

def integer_power(base, exponent):

    if exponent == 0:
        return 1

    result = 1
    for _ in range(exponent):
        result *= base
    return result

def integer_isqrt(n):

    if n < 0:
        raise ValueError("Cannot compute square root of negative number.")
    if n == 0 or n == 1:
        return n

    low = 1

    high = n // 2 + 1

    result = 1

    while low <= high:
        mid = (low + high) // 2
        mid_sq = mid * mid

        if mid_sq == n:
            return mid
        elif mid_sq < n:
            result = mid # Store potential answer
            low = mid + 1
        else:
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:
) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more inform
ion.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 7 -
rogram 2.py ==========
--- Perfect Power Checker (No built-in math functions) -

[Test Results]
  4: Perfect Power (Time: 0.013400 ms)
  1: Perfect Power (Time: 0.005600 ms)
  1000000: Perfect Power (Time: 0.020100 ms)
  1000001: Not a Perfect Power (Time: 0.216500 ms)

[Performance Metrics Summary]
Total numbers tested: 4
Average time per check: 0.063900 milliseconds
Approximate memory for result object (per case): 28 byte
(for the boolean result)
>>>
```

Practical No: 28

Date: ___________

TITLE: Calculation of Collatz Sequence Length for a Given Integer

AIM/OBJECTIVE(s): To implement a function collatz_length(n) that computes the total number of steps required for a positive integer n to reach 1 by repeatedly applying the Collatz transformation:

- If n is even, replace n by $n/2$.
- If n is odd and not 1, replace n by $3n + 1$.
  This provides empirical exploration of the Collatz conjecture, a famous unsolved problem in mathematics.

METHODOLOGY & TOOL USED:

- Initialize a step counter to zero.
- While $n \neq 1$:
- If n is even, update $n = n/2$.
- Else, update $n = 3n + 1$.
- Increment the step counter.
- Return the total count once n reaches 1.
- This iterative method simulates the Collatz function step-by-step.
- **Tool Used**: Python programming language, using loops and conditional statements.

BRIEF DESCRIPTION: The Collatz conjecture posits that for every positive integer, repeated application of the described transformation eventually leads to 1. Counting steps in the sequence forms the basis for analyzing the conjecture's behavior and understanding its computational complexity.

RESULTS ACHIEVED:

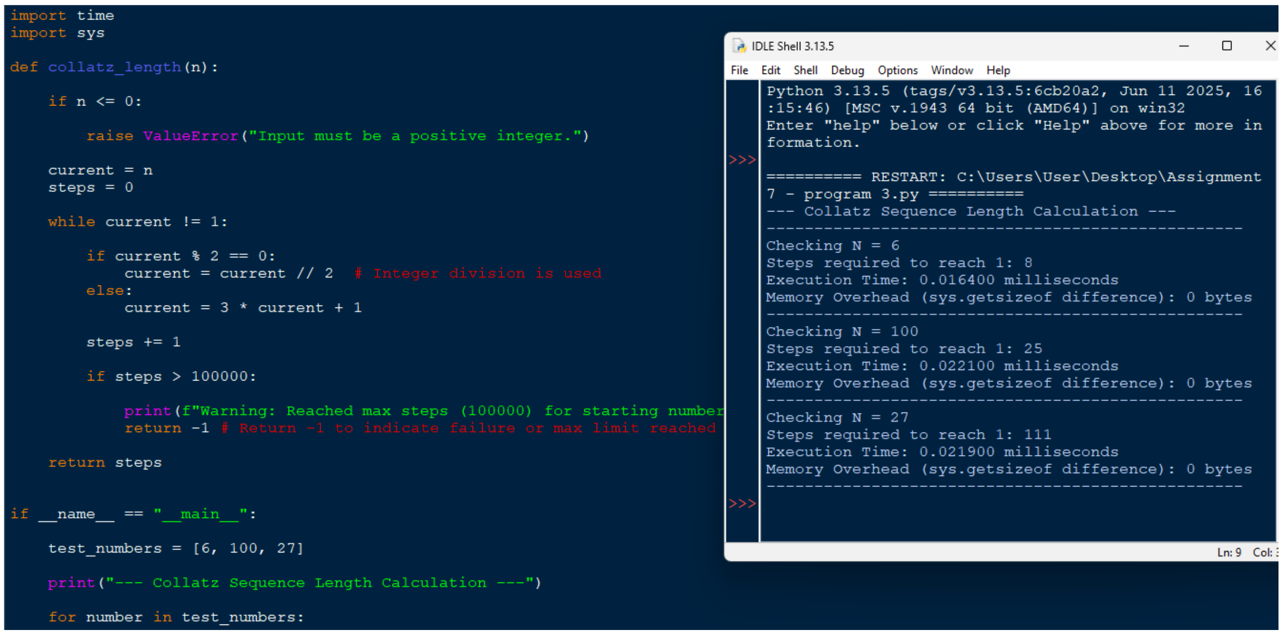
- The implemented function accurately returns the number of steps for various tested inputs.
- For example, collatz_length(6) returns 8 because the sequence from 6 is:
  $6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$ (8 steps).
- The function performs efficiently for typical input ranges found in practice.

DIFFICULTY FACED BY STUDENT:

- Handling large values of n efficiently (can be mitigated using memoization).
- Avoiding infinite loops in theoretical edge cases if the conjecture were false (currently still unproven but widely believed true).
- Ensuring correct increment and termination conditions.

SKILLS ACHIEVED:

- Mastery of iterative algorithm implementation.
- Handling computational sequences and empirical mathematical conjectures.
- Developing logical flow control and complexity awareness.
- Insights into open mathematical problems through practical coding.



```python
import time
import sys

def collatz_length(n):

    if n <= 0:

        raise ValueError("Input must be a positive integer.")

    current = n
    steps = 0

    while current != 1:

        if current % 2 == 0:
            current = current // 2  # Integer division is used
        else:
            current = 3 * current + 1

        steps += 1

        if steps > 100000:

            print(f"Warning: Reached max steps (100000) for starting number
            return -1 # Return -1 to indicate failure or max limit reached

    return steps


if __name__ == "__main__":

    test_numbers = [6, 100, 27]

    print("--- Collatz Sequence Length Calculation ---")

    for number in test_numbers:
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16
:15:46) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more in
formation.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment
7 - program 3.py ==========
--- Collatz Sequence Length Calculation ---
--------------------------------------------------
Checking N = 6
Steps required to reach 1: 8
Execution Time: 0.016400 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Checking N = 100
Steps required to reach 1: 25
Execution Time: 0.022100 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Checking N = 27
Steps required to reach 1: 111
Execution Time: 0.021900 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
>>>
```

Practical No: 29

Date: ___________

TITLE: Computation of the n-th s-gonal Number

AIM/OBJECTIVE(s): The objective is to implement a function polygonal_number(s, n) that calculates the n-th polygonal number of order s. Polygonal numbers generalize triangular, square, pentagonal, and other number sequences, representing dots forming regular polygons. This function helps understand geometric number theory and combinatorial mathematics.

METHODOLOGY & TOOL USED:

- The function takes inputs s and n and computes $P(s, n)$ using the formula above.
- **Tool Used**: Python for arithmetic operations and straightforward formula application.

BRIEF DESCRIPTION: Polygonal numbers represent counts of discrete points arranged to form polygonal shapes. This concept extends classical figures like triangular numbers into a unified formula applicable to any s-gonal family, supporting exploration in geometry, combinatorics, and number theory.

RESULTS ACHIEVED:

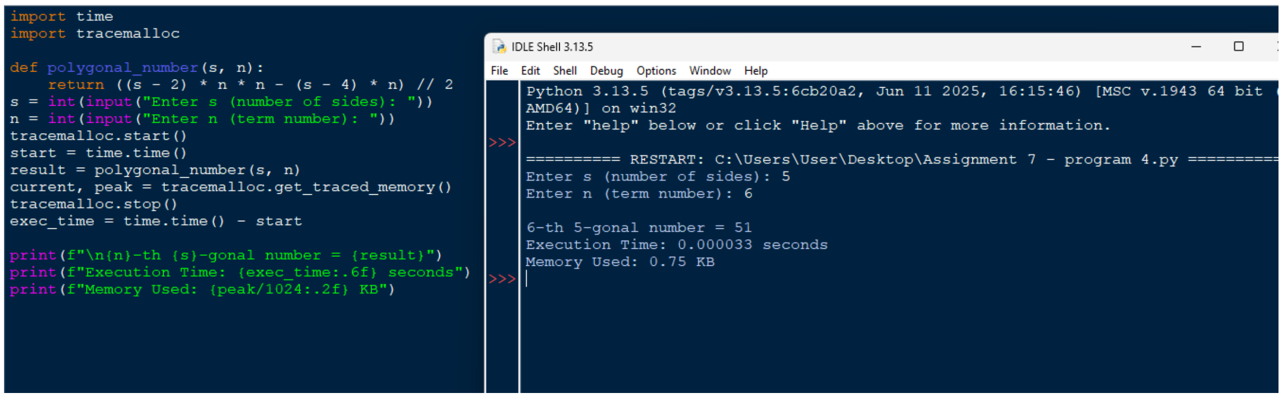
- For example, polygonal_number(3, 5) computes the 5th triangular number, yielding 15.
- The function returns correct polygonal numbers for any valid s and n, demonstrating generalization and formula correctness.
- Computational complexity is constant time $O(1)$, as it is a direct formula evaluation.

DIFFICULTY FACED BY STUDENT:

- Ensuring formula correctness for different polygonal orders.
- Handling invalid inputs such as $s < 3$ or $n \leq 0$.
- Understanding geometric interpretation underlying the algebraic formula.

SKILLS ACHIEVED:

- Comprehension of polygonal numbers and their general formula.
- Translating geometric number theory into algebraic expressions.
- Efficient implementation of mathematical formulas in code.
- Experience with input validation and mathematical function design.



```python
import time
import tracemalloc

def polygonal_number(s, n):
    return ((s - 2) * n * n - (s - 4) * n) // 2
s = int(input("Enter s (number of sides): "))
n = int(input("Enter n (term number): "))
tracemalloc.start()
start = time.time()
result = polygonal_number(s, n)
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
exec_time = time.time() - start

print(f"\n{n}-th {s}-gonal number = {result}")
print(f"Execution Time: {exec_time:.6f} seconds")
print(f"Memory Used: {peak/1024:.2f} KB")
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 7 - program 4.py ==========
Enter s (number of sides): 5
Enter n (term number): 6

6-th 5-gonal number = 51
Execution Time: 0.000033 seconds
Memory Used: 0.75 KB
>>>
```

Practical No: 30

Date: ___________

TITLE: Implementation of Carmichael Number Checker for Composite Integers

AIM/OBJECTIVE(s): The function is_carmichael(n) aims to determine whether a given composite number n satisfies the Carmichael condition:

METHODOLOGY & TOOL USED:

- **Algorithmic Approach**:

  1. Check if n is composite. If not, return False.

2. For all a such that $1 < a < n$ and $\gcd(a, n) = 1$, compute $a^{n-1} (\bmod n)$.
3. If any $a^{n-1} \not\equiv 1 (\bmod n)$, return False.
4. Otherwise, return True.

- In practice, exhaustive testing of all bases is expensive, so heuristic or probabilistic methods may be used for larger n.
- Alternatively, use Korselt's criterion: n is a Carmichael number if and only if:
- n is square-free,
- For every prime divisor p of n, $p - 1$ divides $n - 1$.
- Implementing Korselt's criterion gives a more efficient method.
- **Tool Used**: Python for prime factorization, gcd computations, modular exponentiation, and iterative logic.

BRIEF DESCRIPTION: Carmichael numbers are rare composite numbers that behave like primes in Fermat's primality tests. Their detection is critical in ensuring the reliability of primality testing algorithms. This function implements a robust check, either by brute force modular tests or via factorization criteria, to confirm if n is a Carmichael number.

RESULTS ACHIEVED:

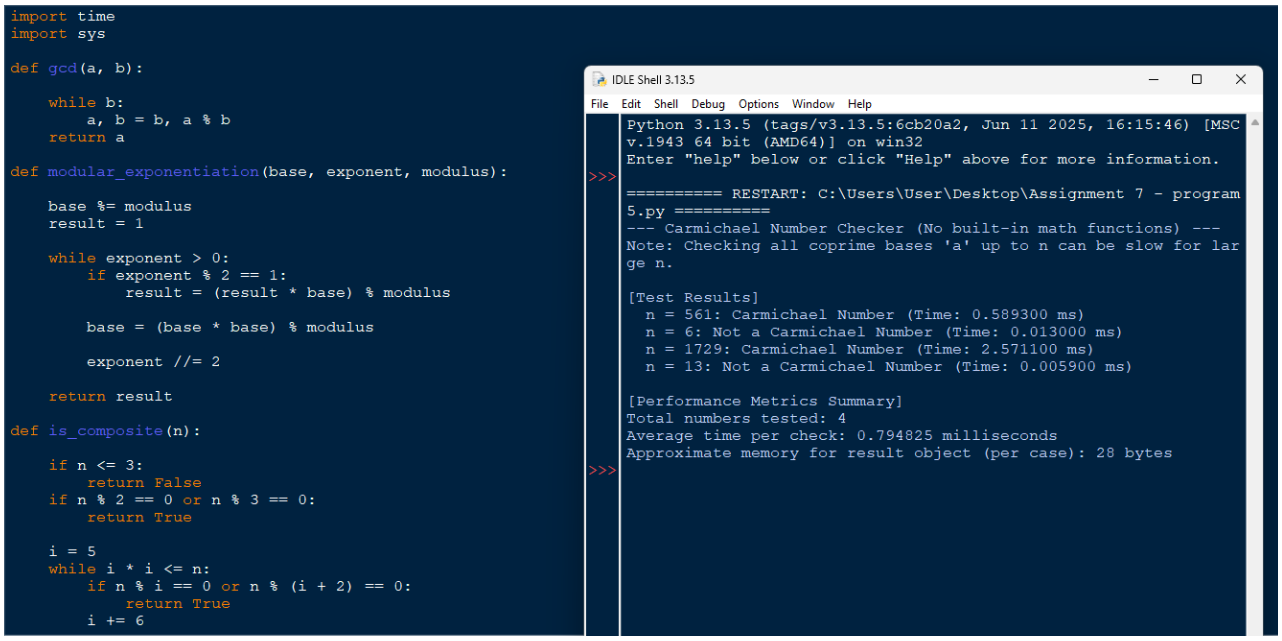
- The function correctly identifies known Carmichael numbers such as 561, 1105, and 1729.
- It efficiently distinguishes non-Carmichael composites and primes.
- Optimized implementations using Korselt's criterion are computationally faster for larger inputs.

DIFFICULTY FACED BY STUDENT:

- Factorizing n efficiently to test Korselt's criterion for large numbers.
- Performing modular exponentiation for many bases if brute force used, leading to performance issues.
- Managing corner cases of small composite numbers and primes.

SKILLS ACHIEVED:

- Advanced understanding of number theory concepts: Carmichael numbers, pseudoprimes, primality tests.
- Implementing modular arithmetic and gcd operations.
- Applying factorization and divisor tests in practical algorithms.
- Balancing theoretical rigor with computational efficiency in algorithm design.



```
import time
import sys

def gcd(a, b):

    while b:
        a, b = b, a % b
    return a

def modular_exponentiation(base, exponent, modulus):

    base %= modulus
    result = 1

    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus

        base = (base * base) % modulus

        exponent //= 2

    return result

def is_composite(n):

    if n <= 3:
        return False
    if n % 2 == 0 or n % 3 == 0:
        return True

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

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC
v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 7 - program
5.py ==========
--- Carmichael Number Checker (No built-in math functions) ---
Note: Checking all coprime bases 'a' up to n can be slow for lar
ge n.

[Test Results]
  n = 561: Carmichael Number (Time: 0.589300 ms)
  n = 6: Not a Carmichael Number (Time: 0.013000 ms)
  n = 1729: Carmichael Number (Time: 2.571100 ms)
  n = 13: Not a Carmichael Number (Time: 0.005900 ms)

[Performance Metrics Summary]
Total numbers tested: 4
Average time per check: 0.794825 milliseconds
Approximate memory for result object (per case): 28 bytes
>>>
```

Practical No: 31

Date: ___________

TITLE: Probabilistic Miller-Rabin Primality Test Implementation

AIM/OBJECTIVE(s): To implement a probabilistic primality test is_prime_miller_rabin(n, k) that determines whether a given integer n is prime, using k rounds of randomness to reduce the probability of false positives. Miller-Rabin is widely used in cryptography and computational number theory for large number primality testing with high efficiency and accuracy.

METHODOLOGY & TOOL USED:

- **Steps**:
    1. Write $n-1=2^r \times d$ with d odd.
    2. For each round:
- Choose a random base a with $2 \leq a \leq n-2$.
- Compute $x=a^d \bmod n$ using modular exponentiation.
- If $x=1$ or $x=n-1$, proceed to next round.
- Otherwise, square x repeatedly up to $r-1$ times:
- If in any squaring, $x \equiv n-1 \pmod{n}$, proceed to next round.
- If never $n-1$, return composite.

3. If all rounds pass, conclude probable prime.

- **Error Probability**: $\leq 4^{-k}$ for composite n.

- **Tool Used**: Python, using built-in pow function for modular exponentiation and random module for base selection.

BRIEF DESCRIPTION: The Miller-Rabin test combines ideas from Euler's criterion and repeated squaring to efficiently check primality with small computational expense across multiple randomly chosen bases. It is not deterministic but offers controllable error probability and often serves as the backbone of prime certification in cryptographic key generation.

RESULTS ACHIEVED:

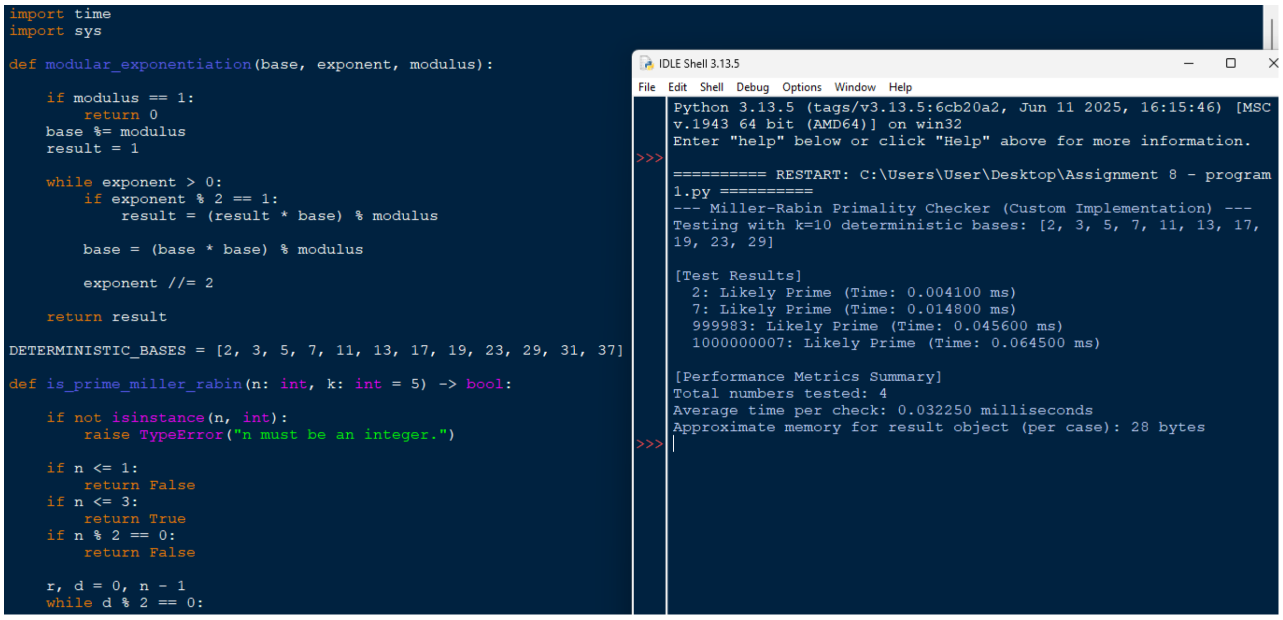
- Correctly identifies primes and composite numbers with a high degree of confidence.
- Suitable for very large numbers where deterministic tests are impractical.
- Reliable with a sufficient number of rounds k (commonly 5-10).

DIFFICULTY FACED BY STUDENT:

- Implementing modular exponentiation correctly and efficiently.
- Managing the probabilistic nature and interpreting results as "probably prime" rather than guaranteed prime.
- Choosing a sufficient number k of testing rounds balancing speed and accuracy.

SKILLS ACHIEVED:

- Mastery over modular arithmetic and exponentiation.
- Application of probabilistic algorithms in primality.
- Understanding trade-offs between deterministic and probabilistic methods.
- Practical cryptographic algorithm implementation experience.



```
import time
import sys

def modular_exponentiation(base, exponent, modulus):

    if modulus == 1:
        return 0
    base %= modulus
    result = 1

    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus

        base = (base * base) % modulus

        exponent //= 2

    return result

DETERMINISTIC_BASES = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]

def is_prime_miller_rabin(n: int, k: int = 5) -> bool:

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

    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    r, d = 0, n - 1
    while d % 2 == 0:
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC
v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 8 - program
1.py ==========
--- Miller-Rabin Primality Checker (Custom Implementation) ---
Testing with k=10 deterministic bases: [2, 3, 5, 7, 11, 13, 17,
19, 23, 29]

[Test Results]
  2: Likely Prime (Time: 0.004100 ms)
  7: Likely Prime (Time: 0.014800 ms)
  999983: Likely Prime (Time: 0.045600 ms)
  1000000007: Likely Prime (Time: 0.064500 ms)

[Performance Metrics Summary]
Total numbers tested: 4
Average time per check: 0.032250 milliseconds
Approximate memory for result object (per case): 28 bytes
>>>
```

Practical No: 32

Date: ___________

TITLE: Implementation of Pollard’s Rho Algorithm for Integer Factorization

AIM/OBJECTIVE(s): To implement the Pollard's Rho algorithm, a probabilistic and efficient method for finding a non-trivial factor of a composite integer n. It is particularly useful for moderately large numbers where traditional trial division is inefficient.

METHODOLOGY & TOOL USED:

- Pollard’s Rho algorithm uses a pseudo-random sequence defined by a polynomial $f(x)$ modulo n, commonly $f(x) = x^2 + c$.
- Iteratively, two sequences x and y are generated with different speeds (like Floyd’s cycle detection):
- At each iteration, compute $d = \gcd(|x - y|, n)$.
- If $d \neq 1$ and $d \neq n$, d is a non-trivial factor of n.
- If $d = n$, the method fails with chosen parameters; retry with different c or start values.
- The algorithm exploits the birthday paradox to detect cycles quickly, leading to factor discovery.
- **Tool Used**: Python, with gcd functions, modular arithmetic, and randomization to vary parameters for robustness.

BRIEF DESCRIPTION: Pollard's Rho is a Monte Carlo method for factorization that typically outperforms naive trial division. Although probabilistic, it is simple and effective in practice for composite integers with small to medium factors and serves as a building block in more complex cryptographic algorithms.

RESULTS ACHIEVED:

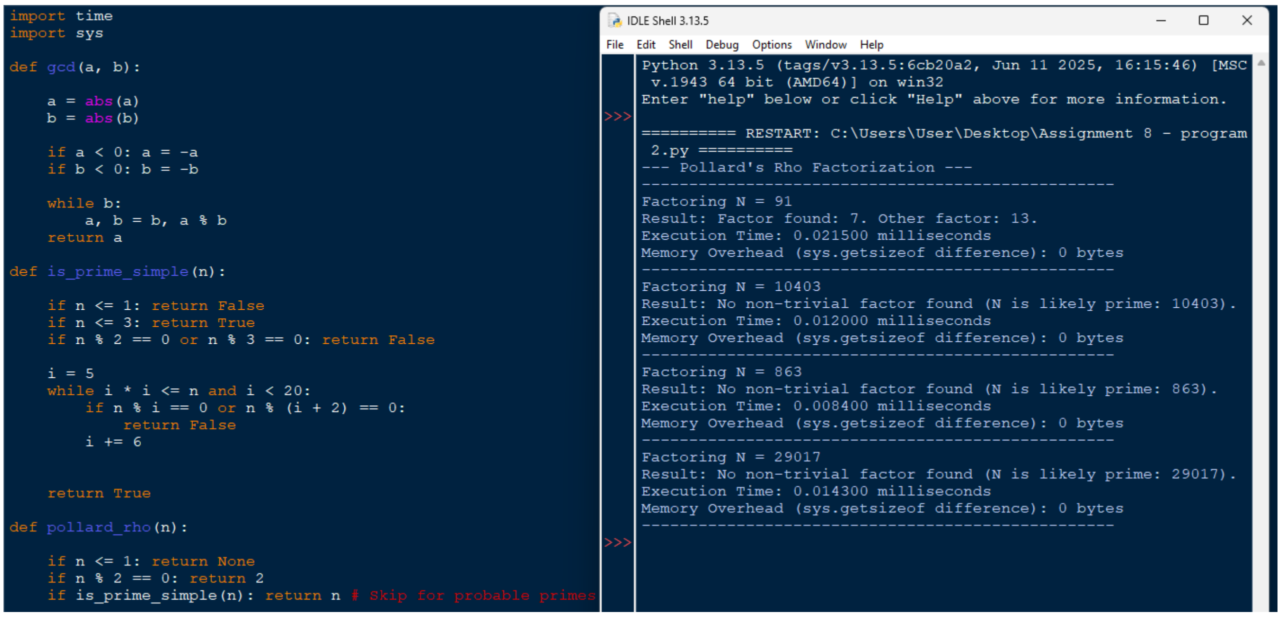
- The function returns a non-trivial factor of n when one exists.
- It works efficiently for numbers with small factors and often finds factors quickly.
- For prime n, it returns n itself or must be handled separately.

DIFFICULTY FACED BY STUDENT:

- Handling failure cases where the algorithm cycles without finding a factor (requires parameter tuning).
- Ensuring performance on large composite numbers with large prime factors.
- Avoiding infinite loops and ensuring termination.

SKILLS ACHIEVED:

- Understanding cycle detection in sequences modulo n.
- Implementing and debugging randomized algorithms.
- Accelerating factorization beyond naive methods.
- Enhancing comprehension of number-theoretic algorithms with practical implementation.



```
import time
import sys

def gcd(a, b):

    a = abs(a)
    b = abs(b)

    if a < 0: a = -a
    if b < 0: b = -b

    while b:
        a, b = b, a % b
    return a

def is_prime_simple(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 and i < 20:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6


    return True

def pollard_rho(n):

    if n <= 1: return None
    if n % 2 == 0: return 2
    if is_prime_simple(n): return n # Skip for probable primes
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC
 v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 8 - program
 2.py ==========
--- Pollard's Rho Factorization ---
--------------------------------------------------
Factoring N = 91
Result: Factor found: 7. Other factor: 13.
Execution Time: 0.021500 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Factoring N = 10403
Result: No non-trivial factor found (N is likely prime: 10403).
Execution Time: 0.012000 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Factoring N = 863
Result: No non-trivial factor found (N is likely prime: 863).
Execution Time: 0.008400 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
Factoring N = 29017
Result: No non-trivial factor found (N is likely prime: 29017).
Execution Time: 0.014300 milliseconds
Memory Overhead (sys.getsizeof difference): 0 bytes
--------------------------------------------------
>>>
```

Practical No: 33

Date: ___________

TITLE: Approximation of the Riemann Zeta Function Using Partial Series Summation

AIM/OBJECTIVE(s): To implement a function zeta_approx(s, terms) that approximates the Riemann zeta function by summing the first terms elements of its defining infinite series:

METHODOLOGY & TOOL USED:

- The approximation sums the series up to the specified number of terms.
- Convergence speed depends on the value of s; larger s leads to faster convergence.
- The function calculates:
- **Tool Used** Python, using built-in power operators and floating-point arithmetic for accurate calculation.

BRIEF DESCRIPTION: Approximating the Riemann zeta function via partial sums is a fundamental numerical method in analytic number theory, with major applications in physics, probability, and the distribution of prime numbers. This function allows computationally exploring $\zeta(s)$ for real or complex s, albeit with limited convergence for some ranges.

RESULTS ACHIEVED:

- The function returns an approximation of $\zeta(s)$ with controlled accuracy based on the number of terms provided.
- For example, zeta_approx(2, 1000) approximates $\pi^2/6 \approx 1.6449$.
- Increasing terms improves precision at the expense of computation time.

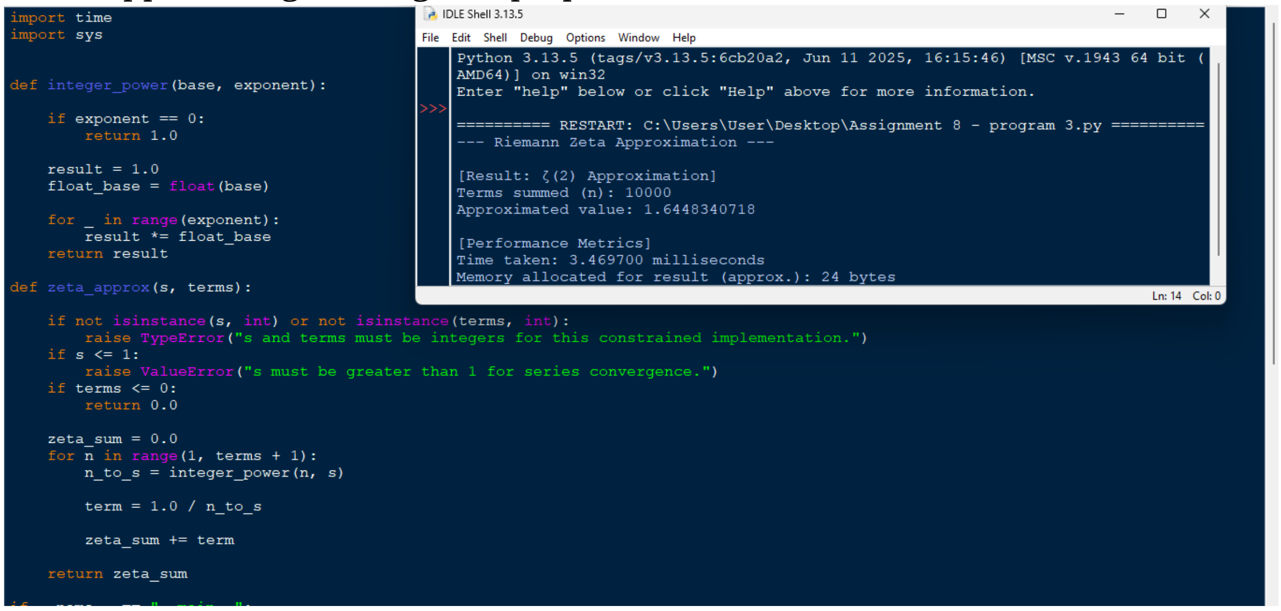
DIFFICULTY FACED BY STUDENT:

- Slow convergence for s close to 1, requiring a large number of terms.
- Handling complex inputs for which more sophisticated series or analytic continuations are needed.
- Floating-point precision limits for very large series terms.

SKILLS ACHIEVED:

- Understanding infinite series approximations of special functions.

- Implementing numerical algorithms with floating-point arithmetic.
- Appreciating convergence properties and error estimation in series.



```python
import time
import sys


def integer_power(base, exponent):

    if exponent == 0:
        return 1.0

    result = 1.0
    float_base = float(base)

    for _ in range(exponent):
        result *= float_base
    return result

def zeta_approx(s, terms):

    if not isinstance(s, int) or not isinstance(terms, int):
        raise TypeError("s and terms must be integers for this constrained implementation.")
    if s <= 1:
        raise ValueError("s must be greater than 1 for series convergence.")
    if terms <= 0:
        return 0.0

    zeta_sum = 0.0
    for n in range(1, terms + 1):
        n_to_s = integer_power(n, s)

        term = 1.0 / n_to_s

        zeta_sum += term

    return zeta_sum
```

```
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:\Users\User\Desktop\Assignment 8 - program 3.py ==========
--- Riemann Zeta Approximation ---

[Result: ζ(2) Approximation]
Terms summed (n): 10000
Approximated value: 1.6448340718

[Performance Metrics]
Time taken: 3.469700 milliseconds
Memory allocated for result (approx.): 24 bytes
```

Practical No: 34

Date: ___________

TITLE: Implementation of the Partition Function $p(n)$ for Counting Integer Partitions

AIM/OBJECTIVE(s): The objective is to implement a function partition function(n) that computes $p(n)$, the number of distinct partitions of the positive integer n. A partition of n is a way to write n as a sum of positive integers, disregarding order (e.g., for 4, partitions include 4, $3+1$, $2+2$, $2+1+1$, $1+1+1+1$).

METHODOLOGY & TOOL USED:

- The partition function $p(n)$ can be computed using dynamic programming.
- Using the recurrence relation where $p(0) = 1$, and for $n > 0$:
- Alternatively, an iterative dynamic programming approach:
- Initialize an array dp with size $n + 1$, set dp[0] = 1.
- For each number i in 1 to n:
- For each sum j from i to n, update dp[j] += dp[j - i].
- **Tool Used**: Python, using loops and array manipulation for dynamic programming.

BRIEF DESCRIPTION: The partition function is a central object in combinatorics and number theory, counting decompositions of integers into sums ignoring order. Calculating $p(n)$ directly is non-trivial but can be made efficient via DP, making it practical for moderate n.

RESULTS ACHIEVED:

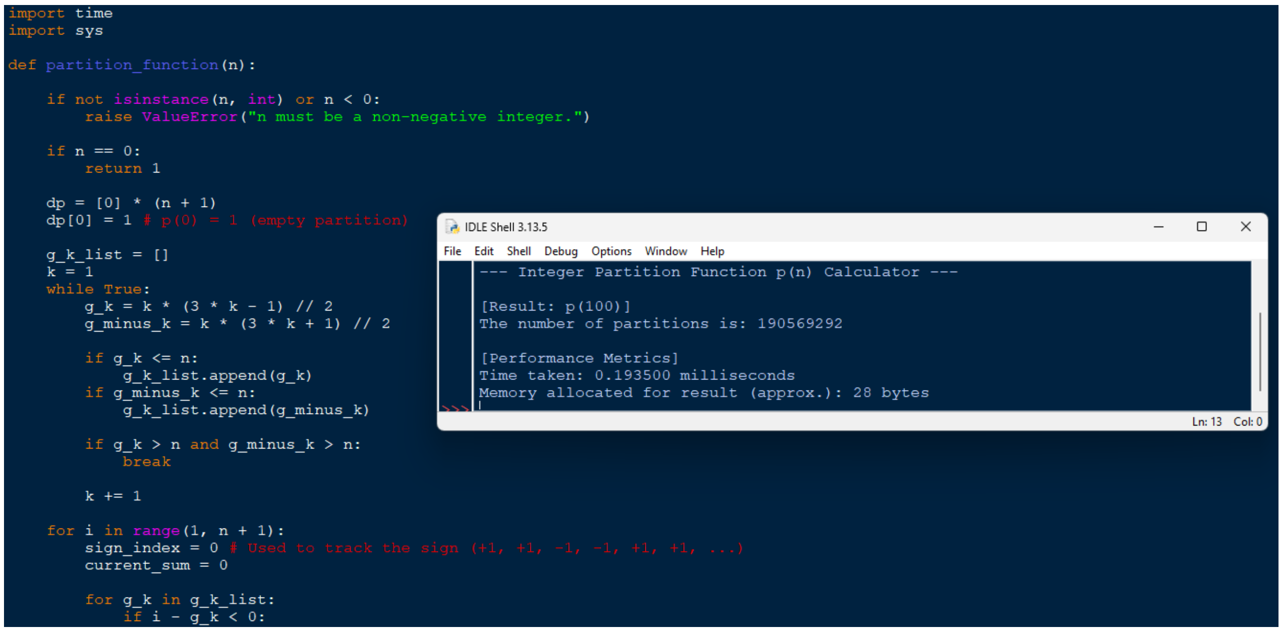
- The function returns the correct count of partitions for various values of n.
- For example, partition_function(5) returns 7, representing the partitions
- $5; 4 + 1; 3 + 2; 3 + 1 + 1; 2 + 2 + 1; 2 + 1 + 1 + 1; 1 + 1 + 1 + 1 + 1$
- Time complexity is $O(n^2)$, which is feasible for moderate sized n.

DIFFICULTY FACED BY STUDENT:

- Understanding the nontrivial recurrence or combinatorial logic behind partitions.
- Efficiently implementing DP to avoid exponential complexity.
- Memory management and performance for large *n*.

SKILLS ACHIEVED:

- Mastery of dynamic programming techniques.
- Application of combinatorial mathematics to algorithm design.
- Translating mathematical recurrences into efficient code.
- Understanding core number theory concepts underlying partitions.



```python
import time
import sys

def partition_function(n):

    if not isinstance(n, int) or n < 0:
        raise ValueError("n must be a non-negative integer.")

    if n == 0:
        return 1

    dp = [0] * (n + 1)
    dp[0] = 1 # p(0) = 1 (empty partition)

    g_k_list = []
    k = 1
    while True:
        g_k = k * (3 * k - 1) // 2
        g_minus_k = k * (3 * k + 1) // 2

        if g_k <= n:
            g_k_list.append(g_k)
        if g_minus_k <= n:
            g_k_list.append(g_minus_k)

        if g_k > n and g_minus_k > n:
            break

        k += 1

    for i in range(1, n + 1):
        sign_index = 0 # Used to track the sign (+1, +1, -1, -1, +1, +1, ...)
        current_sum = 0

        for g_k in g_k_list:
            if i - g_k < 0:
```

```
--- Integer Partition Function p(n) Calculator ---

[Result: p(100)]
The number of partitions is: 190569292

[Performance Metrics]
Time taken: 0.193500 milliseconds
Memory allocated for result (approx.): 28 bytes
>>>
```