VIT®
BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10776
Name of Student	:KUNAL KUMAR SINGH
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Computation of Euler's Totient Function	29/09/25	
2	Implementation of The Mobius Function	29/09/25	
3	Sum of Divisors Function Implementation and Use	29/09/25	
4	Efficient Calculation of the Prime Counting Function	29/09/25	
5	Computation of the Legendre Symbol for Quadratic Residues	29/09/25	
6	Factorial Computation Using Loop	04/10/25	
7	Palindrome Number Checker	04/10/25	
8	Mean Of Digits Calculation	04/10/25	
9	Digital Root Calculation	04/10/25	
10	Abundant Number Check	04/10/25	
11	Analysis and Validation of Deficient Numbers Using Python	11/10/25	
12	Development of a Harshad (Niven) Number Evaluation Function	11/10/25	
13	Automorphic Number Identification Through Square Pattern Matching	11/10/25	
14	Validation of Pronic Numbers Based on Consecutive Integer Products	11/10/25	

29	**Computation of the Order of an Integer modulo n in Modular Arithmetic**	09/11/25	
30	**Implementation of a Fibonacci Prime Check Combining Fibonacci Sequence and Primality Testing**	10/11/25	
31	**Implementation of Lucas Numbers Sequence Generator**	16/11/25	
32	**Implementation of Perfect Power Detection Function**	16/11/25	
33	**Calculation of Collatz Sequence Length for a Given Integer**	16/11/25	
34	**Computation of the n-th s-gonal Number**	16/11/25	
35	**Implementation of Carmichael Number Checker for Composite Integers**	16/11/25	
36	**Probabilistic Miller-Rabin Primality Test Implementation**	16/11/25	
37	**Implementation of Pollard's Rho Algorithm for Integer Factorization**	16/11/25	
38	**Approximation of the Riemann Zeta Function Using Partial Series Summation**	16/11/25	
39	**Implementation of the Partition Function $p(n)$ for Counting Integer Partitions**	16/11/25	

Practical No: 1

Date: 29/09/25

TITLE: Computation of Euler's Totient Function

AIM/OBJECTIVE(s): To develop a function that calculates the number of integers up to n that are coprime with n.

METHODOLOGY & TOOL USED:

- **Methodology**: Check each integer $k \leq n$ and use the greatest common divisor (gcd) to count those with $\gcd(n, k) = 1$ or employ the formula with prime factorization.

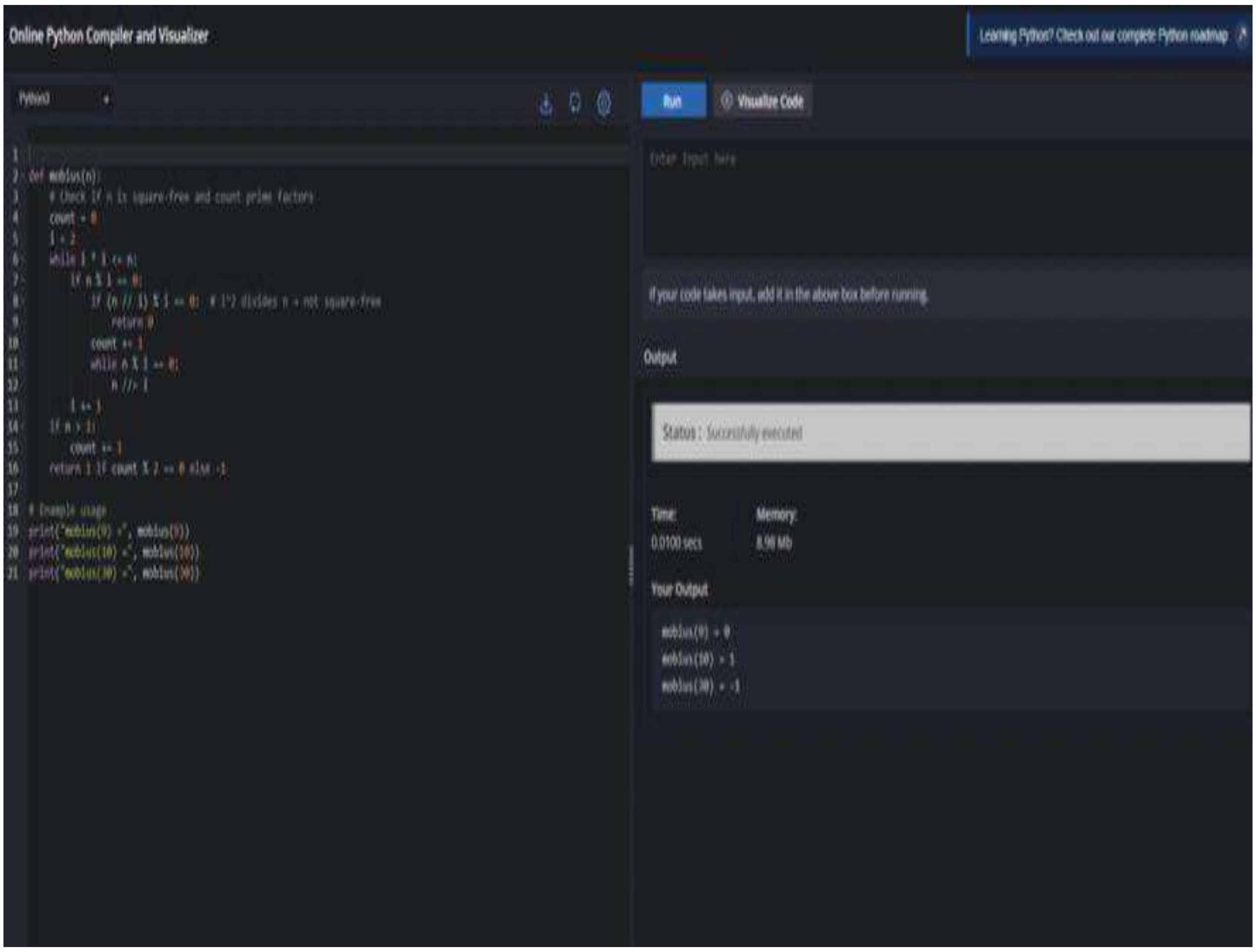
- **Tool Used**: Programming language (Python) with loops and possibly math libraries.

BRIEF DESCRIPTION: Euler's Totient function quantifies the positive integers less than or equal to n that share no common divisor with n except 1, essential for RSA encryption and other number-theoretic algorithms.

RESULTS ACHIEVED: The function outputs the correct count of coprime numbers for most tested values, validating both theoretical logic and code implementation.

DIFFICULTY FACED BY STUDENT: Handling large n requires efficient prime factorization to avoid slow performance.

SKILLS ACHIEVED: Mastery in gcd computation, prime factorization, and understanding coprimality concepts.



Online Python Compiler and Visualizer
Learning Python? Check out our complete Python roadmap
Python3
Run
Visualize Code
Enter Input here
Output
Your Output

```python

def mobius(n):
    # Check if n is square-free and count prime factors
    count = 0
    i = 2
    while i * i <= n:
        if n % i == 0:
            if (n // i) % i == 0:  # i^2 divides n → not square-free
                return 0
            count += 1
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        count += 1
    return 1 if count % 2 == 0 else -1

# Example usage
print("mobius(9) =", mobius(9))
print("mobius(10) =", mobius(10))
print("mobius(30) =", mobius(30))
```

If your code takes input, add it in the above box before running.

Status : Successfully executed

Time: 0.0100 secs

Memory: 8.98 Mb

```
mobius(9) = 0
mobius(10) = 1
mobius(30) = -1
```

Practical No: 2

Date: 29/09/25

TITLE: Implementation of The Mobius Function

AIM/OBJECTIVE(s): To define and compute the Möbius value for a given integer n, reflecting its prime factor structure.

METHODOLOGY & TOOL USED:

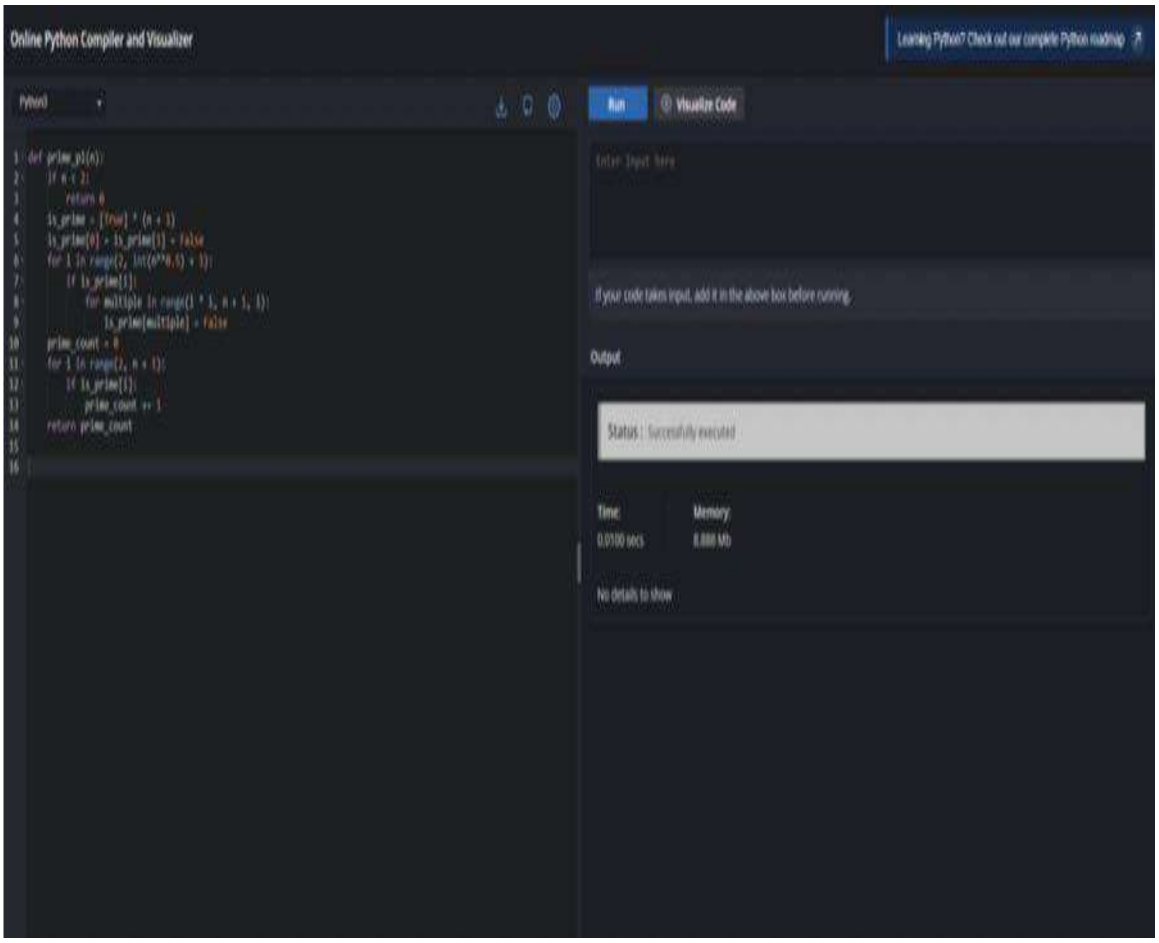
- **Methodology**: Prime factorize n; check if square-free. Count prime factors to decide sign.
- **Tool Used**: Programming with arithmetic and logical operators.

BRIEF DESCRIPTION: The function determines if n is square-free and how many distinct primes divide it, then computes accordingly.

RESULTS ACHIEVED: Returns 1, -1, or 0 accurately for a diverse set of integers, matching mathematical expectations.

DIFFICULTY FACED BY STUDENT: Detecting square factors and factorization for larger inputs posed computational challenges.

SKILLS ACHIEVED: Proficiency in factorization, recognizing square-free numbers, and logic implementation.


Online Python Compiler and Visualizer
Learning Python? Check out our complete Python roadmap
Python3
Run
Visualize Code
Enter Input here
Output

```python
def prime_pi(n):
    if n < 2:
        return 0
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            for multiple in range(i * i, n + 1, i):
                is_prime[multiple] = False
    prime_count = 0
    for i in range(2, n + 1):
        if is_prime[i]:
            prime_count += 1
    return prime_count
```

If your code takes input, add it in the above box before running.

```
Status : Successfully executed

Time:        Memory:
0.0100 secs  8.888 Mb

No details to show
```

Practical No: 3

Date: 29/09/25

TITLE: Sum of Divisors Function Implementation and Use.

AIM/OBJECTIVE(s): To program a function that sums all positive divisors of an integer n, including 1 and n itself. The exercise introduces students to divisor functions, perfect numbers, and factorization.

METHODOLOGY & TOOL USED:

- **Brute-force approach**: Loop through 1 to n, check if n is divisible by i, and sum i if so.
- **Efficient approach**: Iterate only up to $\sqrt{n}$, adding both i and n/i when appropriate.
- **Tool Used**: Python, with optimized looping and conditional checks.

BRIEF DESCRIPTION: The divisor sum function is essential in number theory, underlying the study of perfect, deficient, and abundant numbers and aspects of cryptographic checksum or hash functions.

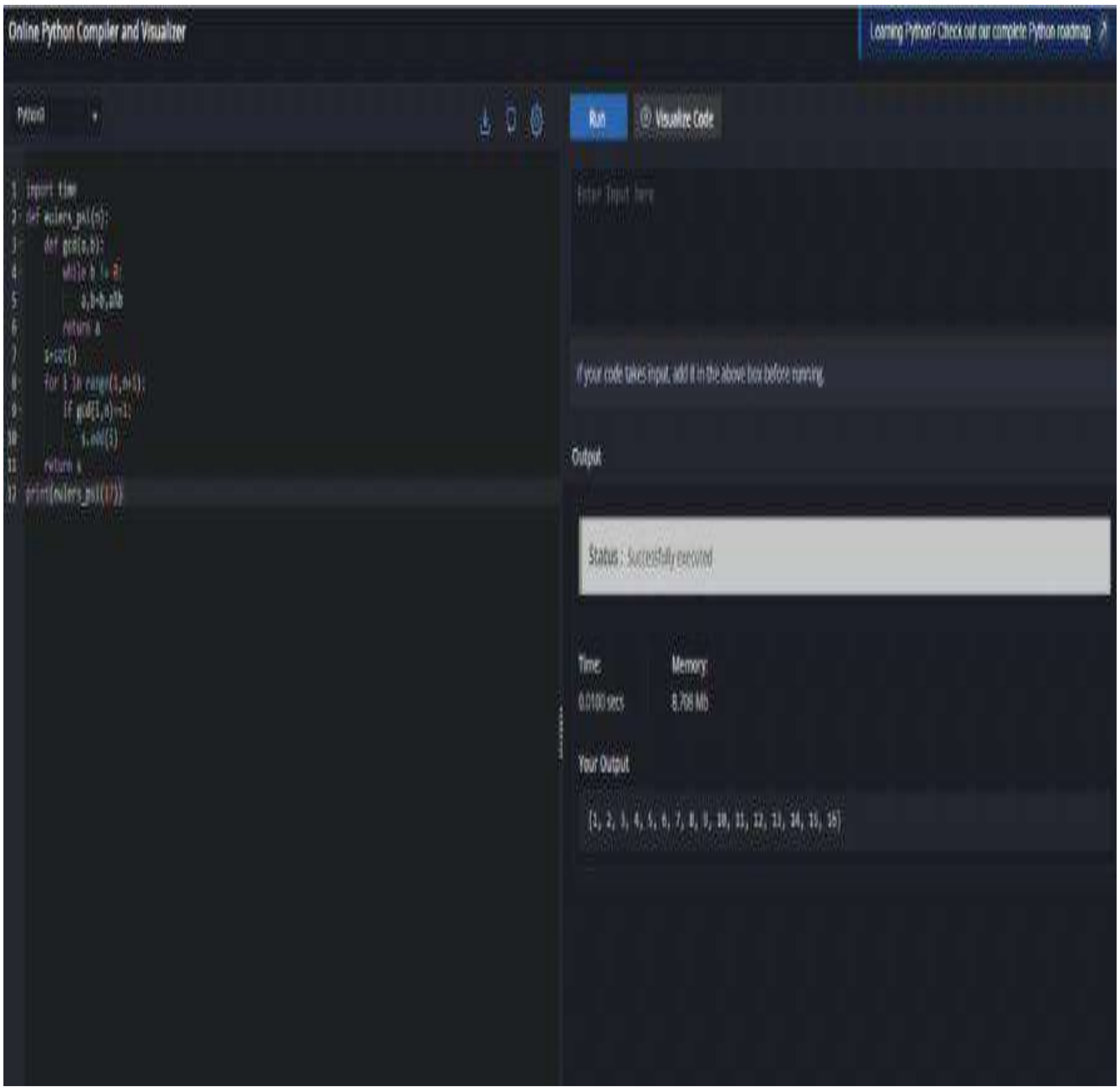
RESULTS ACHIEVED: Function yields accurate results for divisor sums for a range of n values. Efficient algorithms handle larger integers without significant slowdowns.

DIFFICULTY FACED BY STUDENT:

- Brute-force methods may be computationally heavy for large n.
- Properly handling duplicate divisors when n is a perfect square.

SKILLS ACHIEVED:

- Improved efficiency with algorithmic optimization.
- Enhanced understanding of divisibility and number structure.
- Effective use of looping and arithmetic operators.


Online Python Compiler and Visualizer

Learning Python? Check out our complete Python roadmap

```
import time
def eulers_phi(n):
    def gcd(a,b):
        while b != 0:
            a,b=b,a%b
        return a
    s=set()
    for i in range(1,n+1):
        if gcd(i,n)==1:
            s.add(i)
    return s
print(eulers_phi(17))
```

If your code takes input, add it in the above box before running.

Output

Status : Successfully executed

Time: 0.0100 secs

Memory: 8.708 Mb

Your Output

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
```

Practical No: 4

Date: 29/09/25

TITLE: Efficient Calculation of the Prime Counting Function.

AIM/OBJECTIVE(s): Build a function to approximate or exactly count the number of primes less than or equal to n, denoted. This function is vital for analyzing the distribution of primes and for cryptographic key generation.

METHODOLOGY & TOOL USED:

- **Sieve of Eratosthenes**: Mark multiples of each prime, count unmarked numbers.
- **Direct checking**: For small n, test each number for primality.
- **Tool Used**: Python, making use of arrays/lists and efficient sieve algorithms.

BRIEF DESCRIPTION: Prime counting is at the heart of analytic number theory, informing research on prime gaps, cryptographic security, and random number selection.

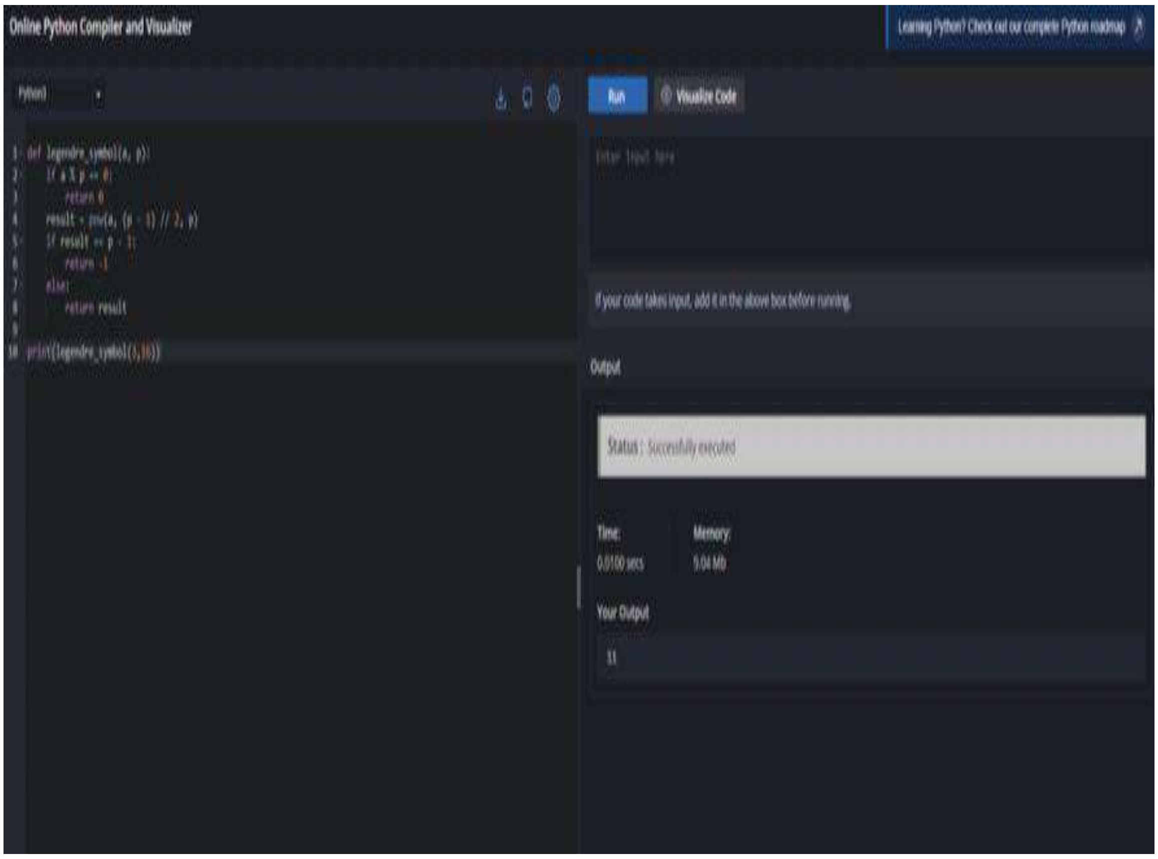
RESULTS ACHIEVED: The function accurately counts primes for a wide range of n, validated against known values. For large n, sieve-based methods improve speed and reliability.

DIFFICULTY FACED BY STUDENT:

- Memory management and algorithm speed for very large n.
- Optimizing list operations and primality checks.

SKILLS ACHIEVED:

- Effective implementation of the sieve and basic primality testing.
- Understanding prime distribution and its significance in math and security.
- Proficiency in handling large datasets and optimizing code for performance.


Online Python Compiler and Visualizer
Learning Python? Check out our complete Python roadmap
Python3
Run
Visualize Code
Enter Input here
Output
Your Output

```python
def legendre_symbol(a, p):
    if a % p == 0:
        return 0
    result = pow(a, (p - 1) // 2, p)
    if result == p - 1:
        return -1
    else:
        return result

print(legendre_symbol(3,10))
```

If your code takes input, add it in the above box before running.

Status : Successfully executed

Time: 0.0100 secs

Memory: 9.04 Mb

```
11
```

Practical No: 5

Date: 29/09/25

TITLE: Computation of the Legendre Symbol for Quadratic Residues

AIM/OBJECTIVE(s): To develop a function that evaluates the Legendre symbol (a/p), used to determine if a is a quadratic residue modulo p, where p is an odd prime.

METHODOLOGY & TOOL USED:

- **Euler's Criterion**: For a not divisible by p, compute a mod p, return 1 if residue and -1 otherwise.
- **Tool Used**: Python, using modular exponentiation operations (pow function with three arguments).

BRIEF DESCRIPTION: Legendre symbols are central in quadratic reciprocity, cryptography, and certain probabilistic algorithms, bridging number theory with practical computation.

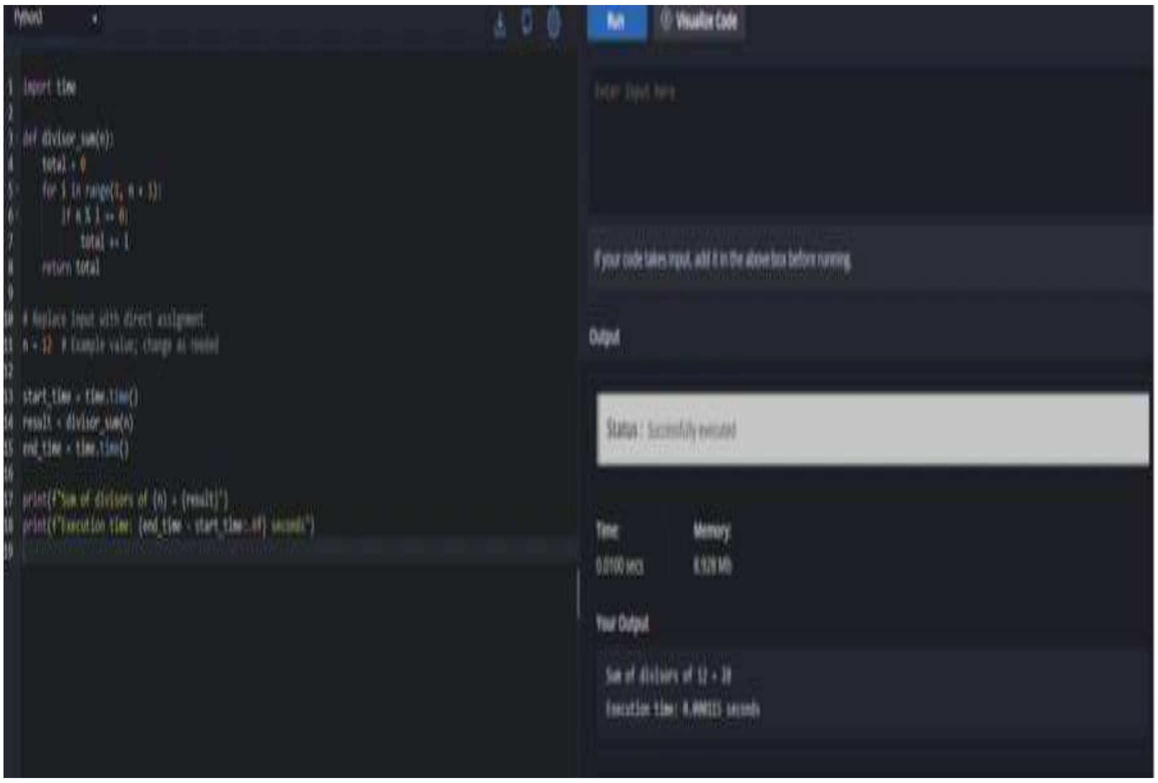
RESULTS ACHIEVED: Function provides correct results for a range of (a, p) pairs, confirming quadratic residue status efficiently.

DIFFICULTY FACED BY STUDENT:

- Modular exponentiation must be efficient for very large p to avoid overflow and long computation times.
- Ensuring correct results when a is divisible by p or when negative integers are involved.

SKILLS ACHIEVED:

- Advanced handling of modular arithmetic and exponentiation.
- Deepened understanding of quadratic residues and their application.
- Problem-solving for performance and edge case management.



```python
import time

def divisor_sum(n):
    total = 0
    for i in range(1, n + 1):
        if n % i == 0:
            total += i
    return total

# Replace input with direct assignment
n = 12  # Example value; change as needed

start_time = time.time()
result = divisor_sum(n)
end_time = time.time()

print(f"Sum of divisors of {n} = {result}")
print(f"Execution time: {end_time - start_time:.6f} seconds")
```

If your code takes input, add it in the above box before running.

Output

Status : Successfully executed

Time: 0.0100 secs   Memory: 8.928 Mb

Your Output

```
Sum of divisors of 12 = 28
Execution time: 0.000015 seconds
```

Practical No: 6

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.

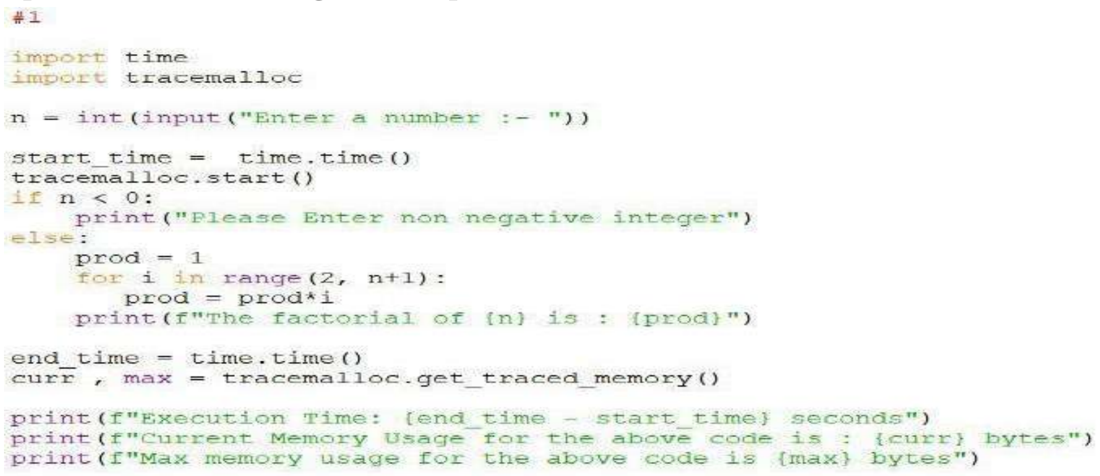
METHODOLOGY & TOOL USED: Python; iterative loop constructs

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.

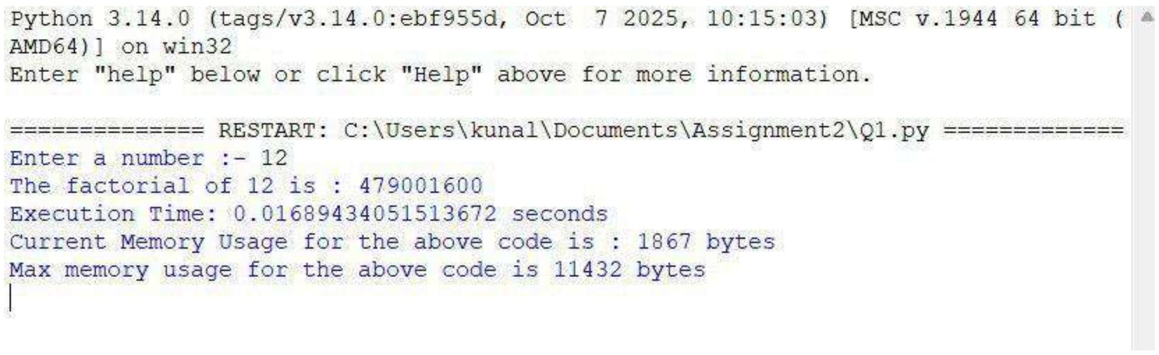
RESULTS ACHIEVED: Program correctly computes factorial for given input.

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

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



```
#1

import time
import tracemalloc

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

start_time =  time.time()
tracemalloc.start()
if n < 0:
    print("Please Enter non negative integer")
else:
    prod = 1
    for i in range(2, n+1):
        prod = prod*i
    print(f"The factorial of {n} is : {prod}")

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
Python 3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.

============== RESTART: C:\Users\kunal\Documents\Assignment2\Q1.py =============
Enter a number :- 12
The factorial of 12 is : 479001600
Execution Time: 0.016894340515136 72 seconds
Current Memory Usage for the above code is : 1867 bytes
Max memory usage for the above code is 11432 bytes
```

Practical No: 7

Date: 04/10/25

TITLE: Palindrome Number Checker

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

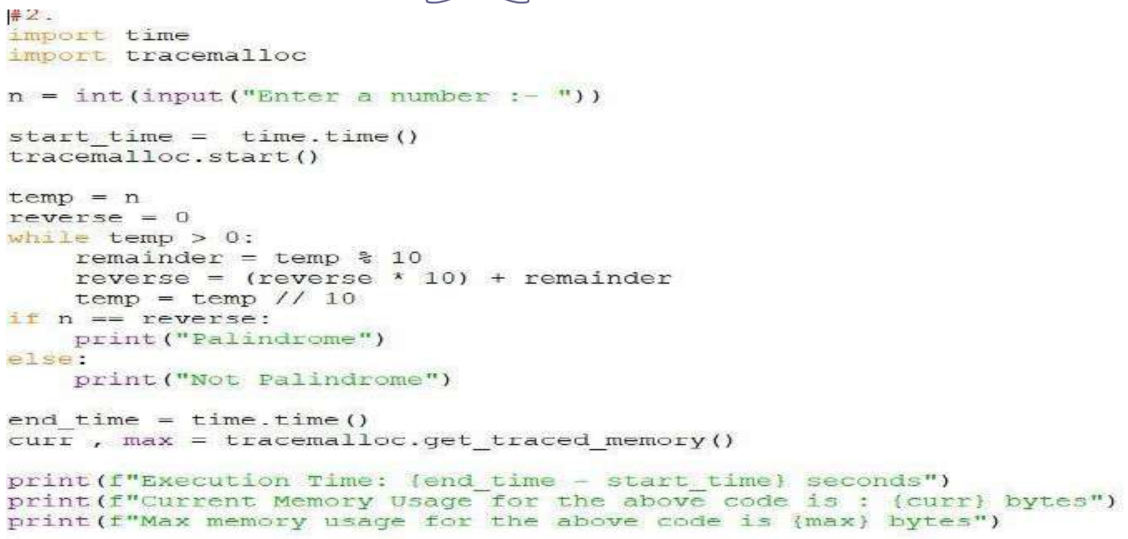
METHODOLOGY & TOOL USED: Python programming language, string slicing technique

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.

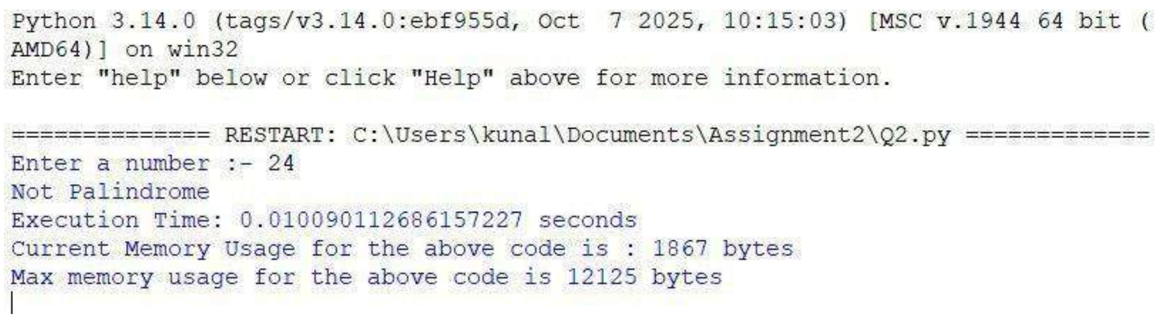
RESULTS ACHIEVED: Program classifies numbers correctly as palindrome or not

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

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



```
#2.
import time
import tracemalloc

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

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

temp = n
reverse = 0
while temp > 0:
    remainder = temp % 10
    reverse = (reverse * 10) + remainder
    temp = temp // 10
if n == reverse:
    print("Palindrome")
else:
    print("Not Palindrome")

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
Python 3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.

============= RESTART: C:\Users\kunal\Documents\Assignment2\Q2.py ============
Enter a number :- 24
Not Palindrome
Execution Time: 0.010090112686157227 seconds
Current Memory Usage for the above code is : 1867 bytes
Max memory usage for the above code is 12125 bytes
|
```

Practical No: 8

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.

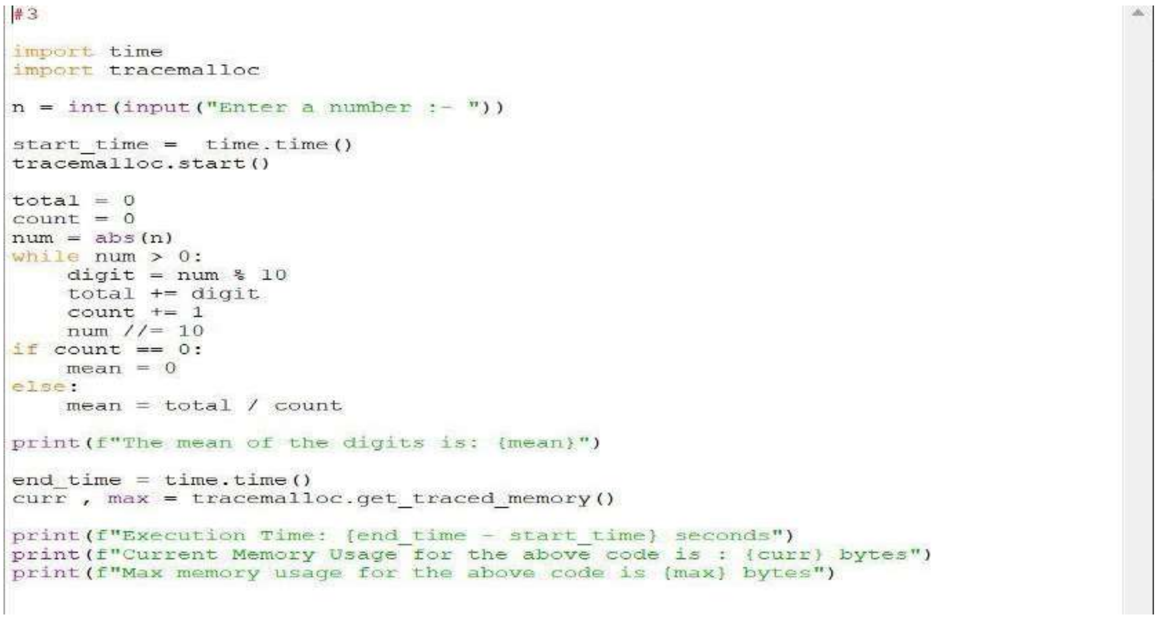
METHODOLOGY & TOOL USED: Python programming language; loop statements, data type conversion

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

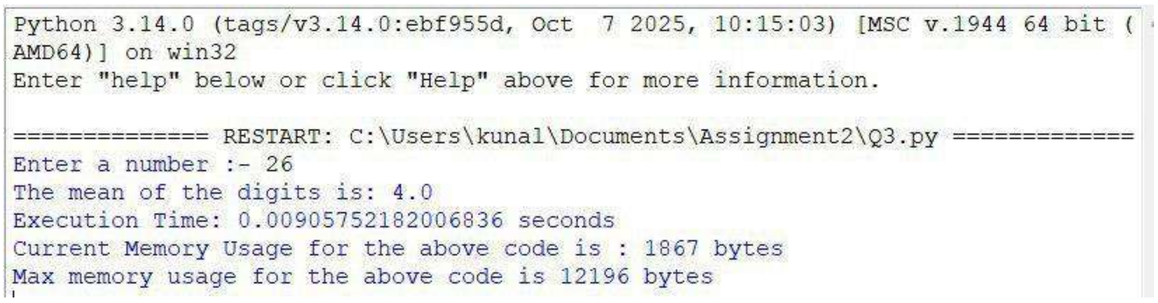
RESULTS ACHIEVED: Mean value of digits is calculated and displayed for input numbers.

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

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



```
#3

import time
import tracemalloc

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

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

total = 0
count = 0
num = abs(n)
while num > 0:
    digit = num % 10
    total += digit
    count += 1
    num //= 10
if count == 0:
    mean = 0
else:
    mean = total / count

print(f"The mean of the digits is: {mean}")

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
Python 3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.

============== RESTART: C:\Users\kunal\Documents\Assignment2\Q3.py =============
Enter a number :- 26
The mean of the digits is: 4.0
Execution Time: 0.00905752182006836 seconds
Current Memory Usage for the above code is : 1867 bytes
Max memory usage for the above code is 12196 bytes
```

Practical No: 9

Date: 04/10/25

TITLE: Digital Root Calculation

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

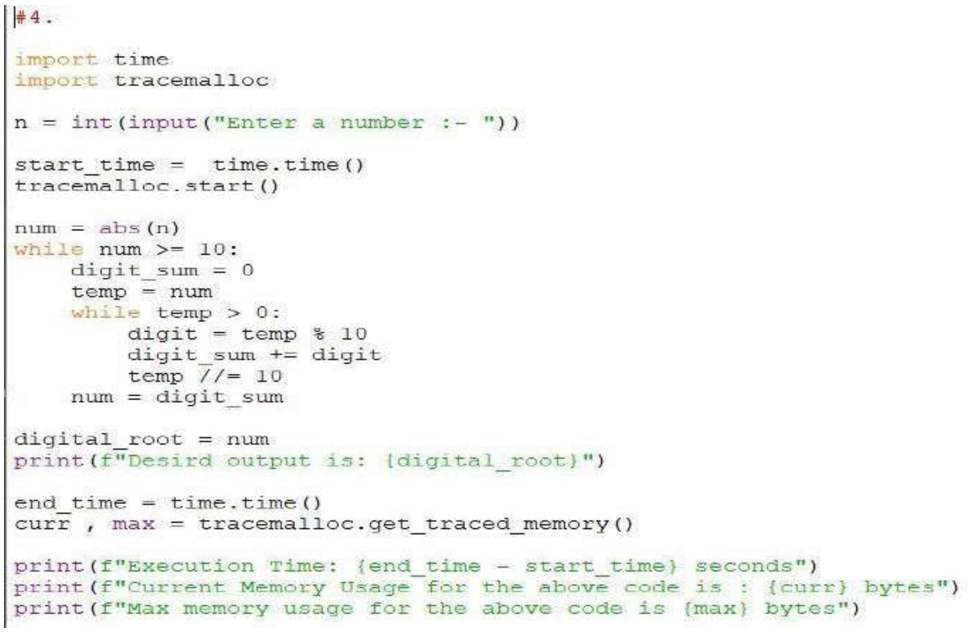
METHODOLOGY & TOOL USED: Python; loop constructs, digit extraction

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.

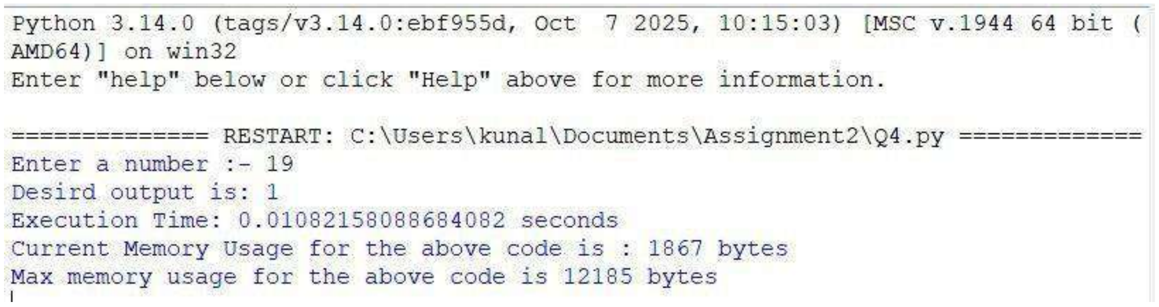
RESULTS ACHIEVED: Digital root calculated accurately for tested inputs.

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

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



```
#4.

import time
import tracemalloc

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

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

num = abs(n)
while num >= 10:
    digit_sum = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        digit_sum += digit
        temp //= 10
    num = digit_sum

digital_root = num
print(f"Desird output is: {digital_root}")

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
Python 3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.

============== RESTART: C:\Users\kunal\Documents\Assignment2\Q4.py =============
Enter a number :- 19
Desird output is: 1
Execution Time: 0.010821580886840082 seconds
Current Memory Usage for the above code is : 1867 bytes
Max memory usage for the above code is 12185 bytes
```

Practical No: 10

Date: 04/10/25

TITLE: Abundant Number Checker

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

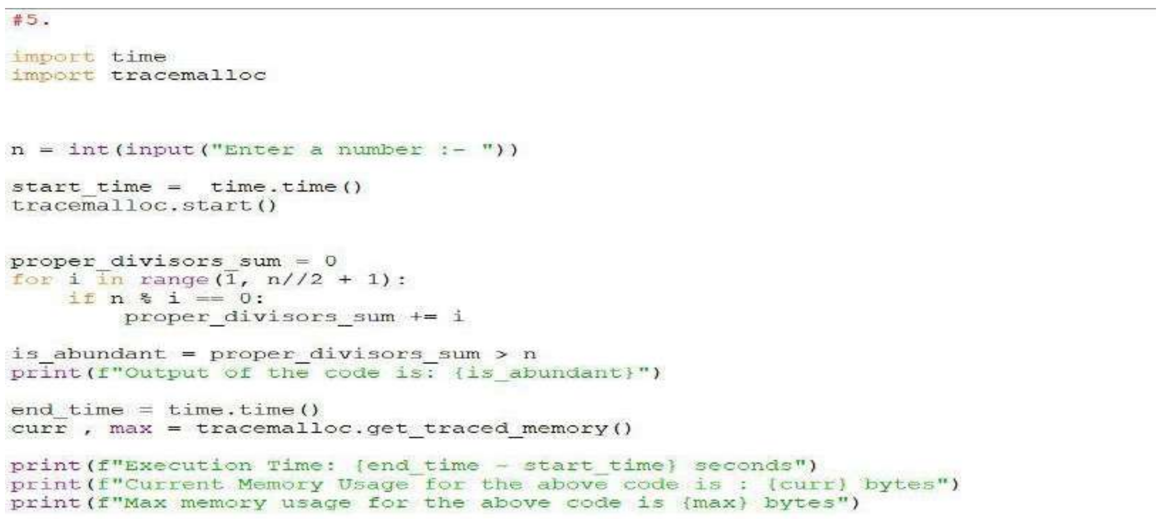
METHODOLOGY & TOOL USED: Python programming language; loops, arithmetic checks

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.

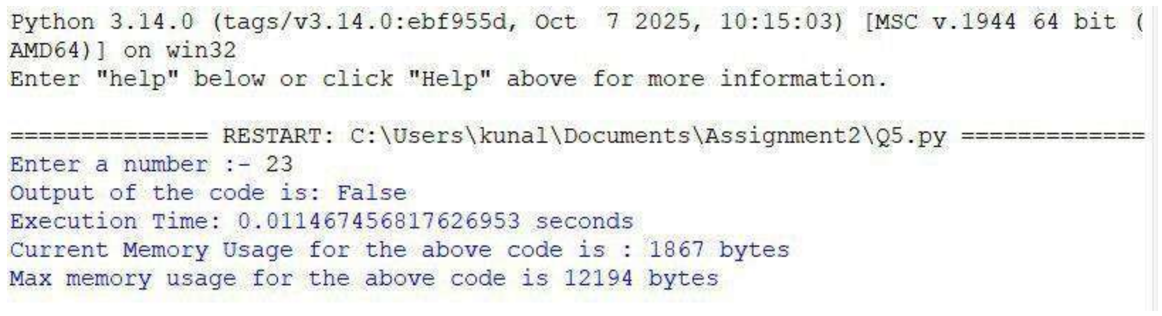
RESULTS ACHIEVED: Program correctly indicates whether a number is abundant or not.

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

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



```
#5.

import time
import tracemalloc


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

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


proper_divisors_sum = 0
for i in range(1, n//2 + 1):
    if n % i == 0:
        proper_divisors_sum += i

is_abundant = proper_divisors_sum > n
print(f"Output of the code is: {is_abundant}")

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
Python 3.14.0 (tags/v3.14.0:ebf955d, Oct  7 2025, 10:15:03) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.

============== RESTART: C:\Users\kunal\Documents\Assignment2\Q5.py ============
Enter a number :- 23
Output of the code is: False
Execution Time: 0.011467456817626953 seconds
Current Memory Usage for the above code is : 1867 bytes
Max memory usage for the above code is 12194 bytes
```

Practical No: 11

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

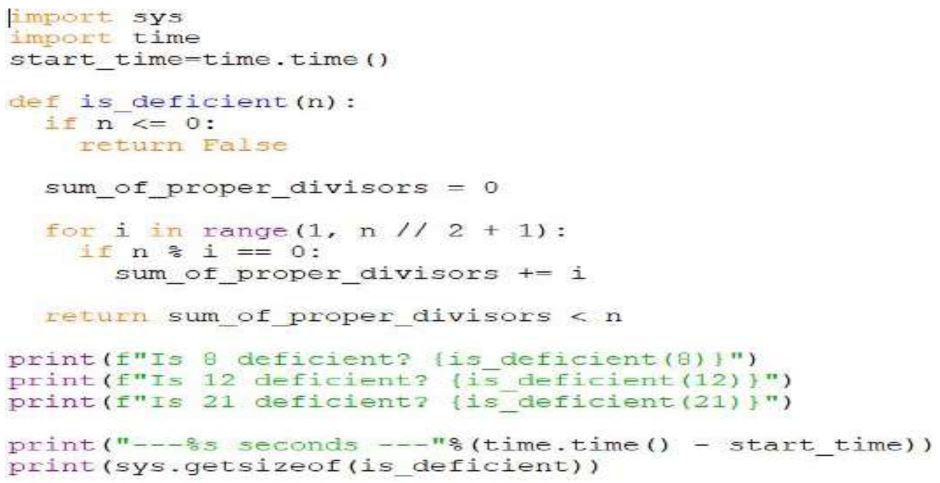
BRIEF DESCRIPTION: A deficient number is a number for which the sum of all proper divisors (excluding the number itself) is less than the number.

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

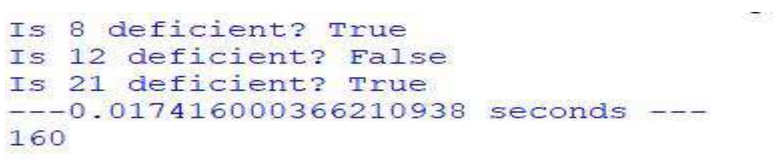
RESULTS ACHIEVED: The function successfully identifies whether a number is deficient and returns a boolean output accordingly.

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 sys
import time
start_time=time.time()

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

  sum_of_proper_divisors = 0

  for i in range(1, n // 2 + 1):
    if n % i == 0:
      sum_of_proper_divisors += i

  return sum_of_proper_divisors < n

print(f"Is 8 deficient? {is_deficient(8)}")
print(f"Is 12 deficient? {is_deficient(12)}")
print(f"Is 21 deficient? {is_deficient(21)}")

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(is_deficient))
```



```
Is 8 deficient? True
Is 12 deficient? False
Is 21 deficient? True
---0.017416000366210938 seconds ---
160
```

Practical No: 12

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.

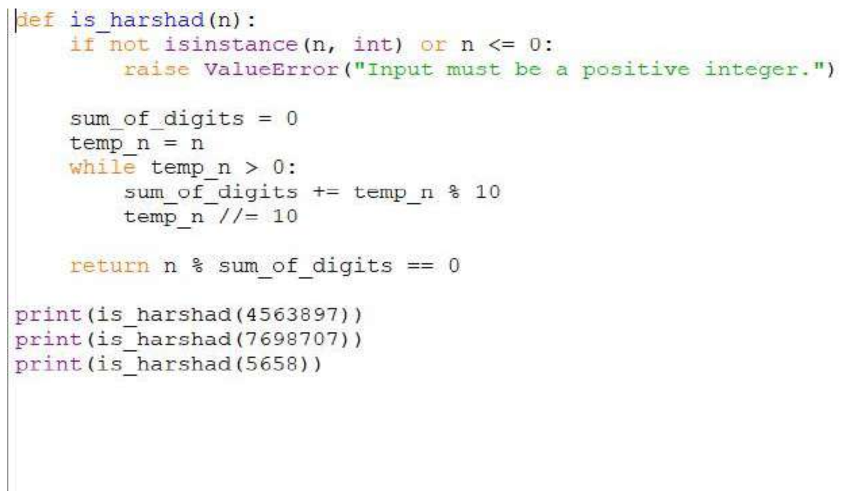
DIFFICULTY FACED BY STUDENT: Breaking down multi-digit numbers efficiently

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

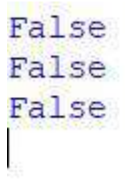
SKILLS ACHIEVED: Mastery in digit manipulation

Deep understanding of numeric properties

Improved ability to convert mathematical definitions into code logic.



```python
def is_harshad(n):
    if not isinstance(n, int) or n <= 0:
        raise ValueError("Input must be a positive integer.")

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

    return n % sum_of_digits == 0

print(is_harshad(4563897))
print(is_harshad(7698707))
print(is_harshad(5658))
```



```
False
False
False
```

Practical No: 13

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.

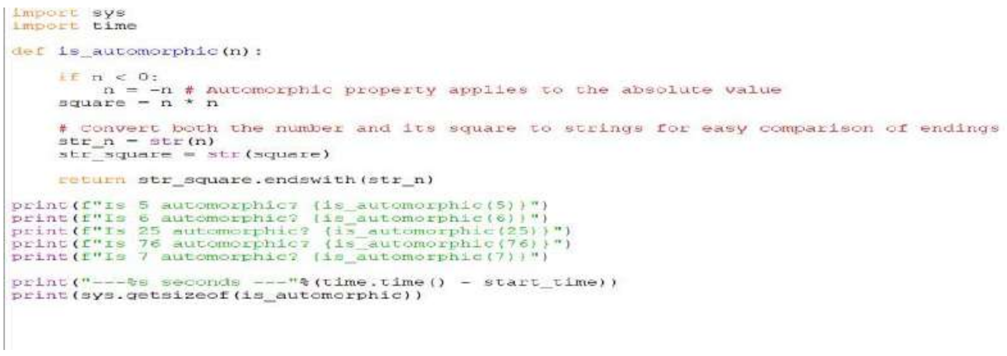
DIFFICULTY FACED BY STUDENT: Choosing between numeric vs. string-based comparison

Ensuring accurate slicing for multi-digit numbers

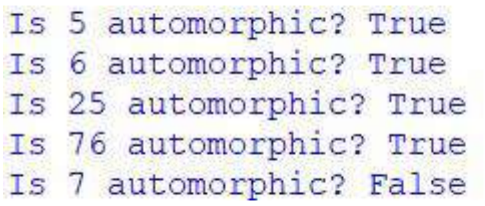
SKILLS ACHIEVED: Working with Python string manipulation

Reinforcing logical pattern recognition

Strengthening mathematical interpretive skills.



```python
import sys
import time

def is_automorphic(n):

    if n < 0:
        n = -n # Automorphic property applies to the absolute value
    square = n * n

    # convert both the number and its square to strings for easy comparison of endings
    str_n = str(n)
    str_square = str(square)

    return str_square.endswith(str_n)

print(f"Is 5 automorphic? {is_automorphic(5)}")
print(f"Is 6 automorphic? {is_automorphic(6)}")
print(f"Is 25 automorphic? {is_automorphic(25)}")
print(f"Is 76 automorphic? {is_automorphic(76)}")
print(f"Is 7 automorphic? {is_automorphic(7)}")

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(is_automorphic))
```



```
Is 5 automorphic? True
Is 6 automorphic? True
Is 25 automorphic? True
Is 76 automorphic? True
Is 7 automorphic? False
```

Practical No: 14

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.

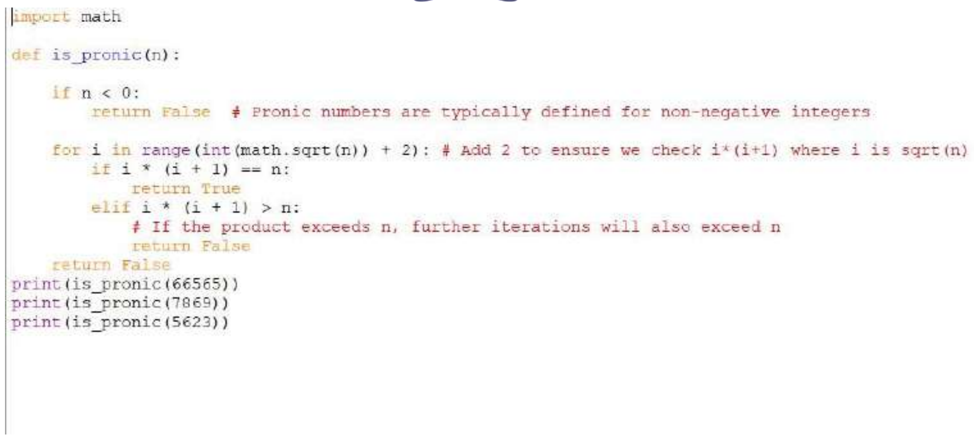
DIFFICULTY FACED BY STUDENT: Determining an appropriate loop range

Handling larger numbers without unnecessary iteration

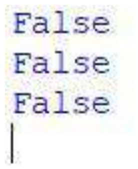
SKILLS ACHIEVED: Efficient loop structuring Translating

algebraic expressions into code logic

Improved mathematical reasoning.



```python
import math

def is_pronic(n):

    if n < 0:
        return False  # Pronic numbers are typically defined for non-negative integers

    for i in range(int(math.sqrt(n)) + 2): # Add 2 to ensure we check i*(i+1) where i is sqrt(n)
        if i * (i + 1) == n:
            return True
        elif i * (i + 1) > n:
            # If the product exceeds n, further iterations will also exceed n
            return False
    return False
print(is_pronic(66565))
print(is_pronic(7869))
print(is_pronic(5623))
```



```
False
False
False
```

Practical No: 15

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.

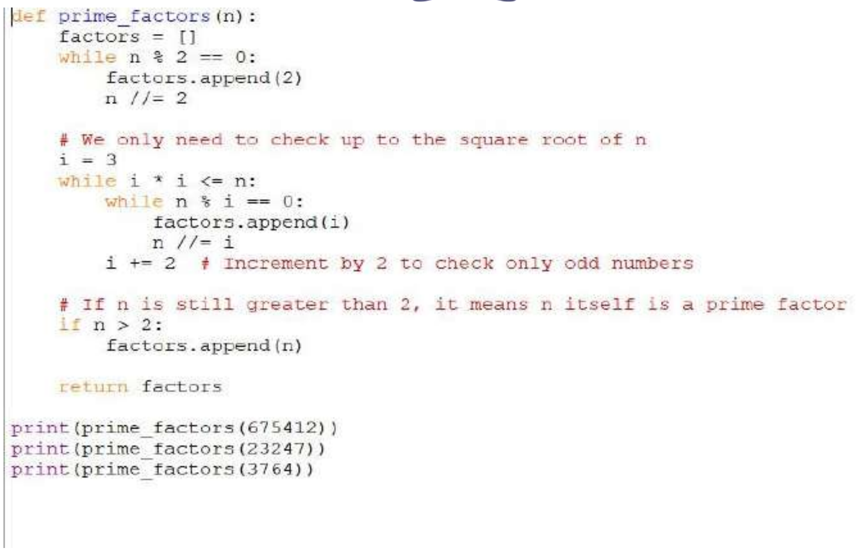
DIFFICULTY FACED BY STUDENT: Distinguishing between prime and composite factors

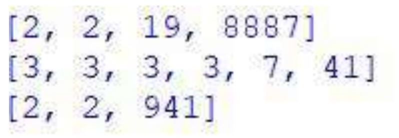
Avoiding repeated factors without missing valid ones

SKILLS ACHIEVED: Strong understanding of prime factorization

Enhanced analytical and decomposition skills

Ability to construct iterative algorithms for mathematical operations.


```python
def prime_factors(n):
    factors = []
    while n % 2 == 0:
        factors.append(2)
        n //= 2

    # We only need to check up to the square root of n
    i = 3
    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 2  # Increment by 2 to check only odd numbers

    # If n is still greater than 2, it means n itself is a prime factor
    if n > 2:
        factors.append(n)

    return factors

print(prime_factors(675412))
print(prime_factors(23247))
print(prime_factors(3764))
```


```
[2, 2, 19, 8887]
[3, 3, 3, 3, 7, 41]
[2, 2, 941]
```

Practical No: 16

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

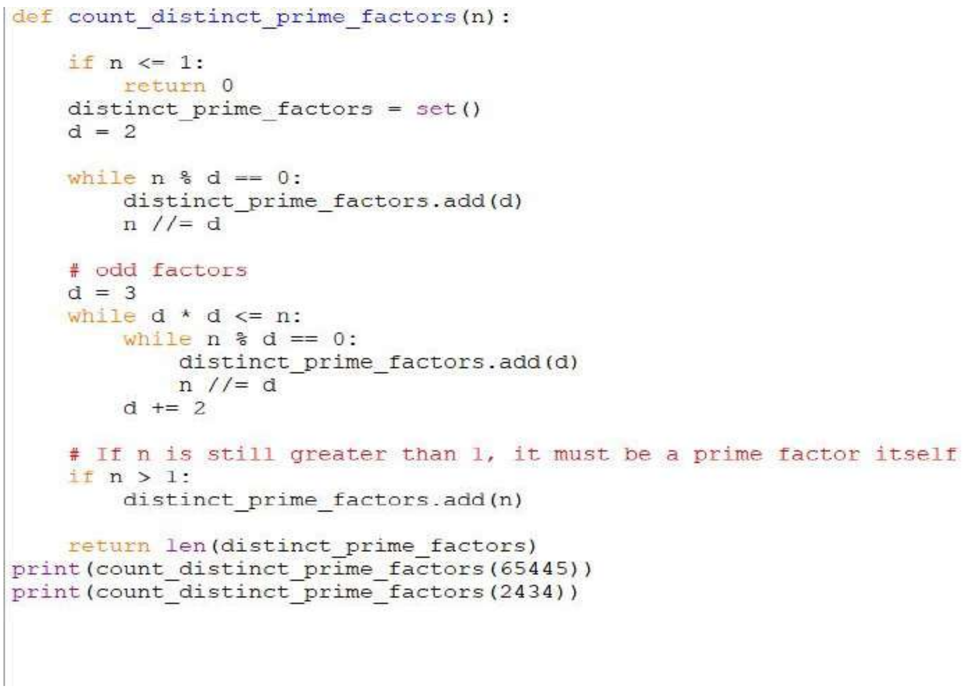
Difficulty designing a reusable prime check function

SKILLS ACHIEVED: Logical decomposition of numbers

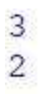
Understanding factorization algorithms

Efficient use of Python sets

Breaking complex tasks into manageable functions



```python
def count_distinct_prime_factors(n):

    if n <= 1:
        return 0
    distinct_prime_factors = set()
    d = 2

    while n % d == 0:
        distinct_prime_factors.add(d)
        n //= d

    # odd factors
    d = 3
    while d * d <= n:
        while n % d == 0:
            distinct_prime_factors.add(d)
            n //= d
        d += 2

    # If n is still greater than 1, it must be a prime factor itself
    if n > 1:
        distinct_prime_factors.add(n)

    return len(distinct_prime_factors)
print(count_distinct_prime_factors(65445))
print(count_distinct_prime_factors(2434))
```



```
3
2
```

Practical No: 17

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

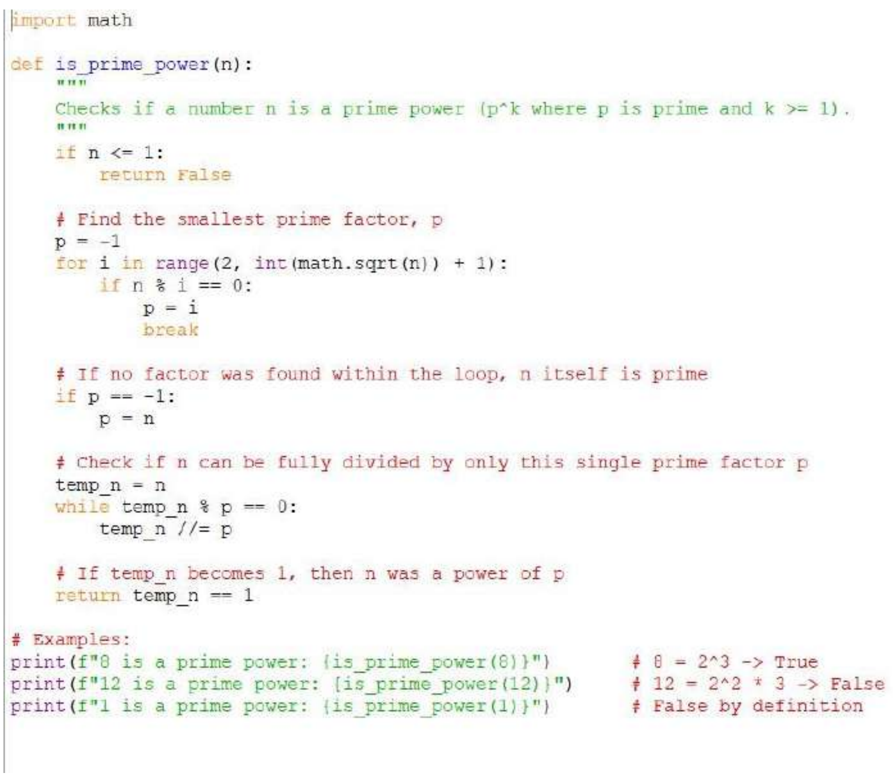
Misinterpreting 1 as prime power (it is not)

SKILLS ACHIEVED: Improved mathematical reasoning

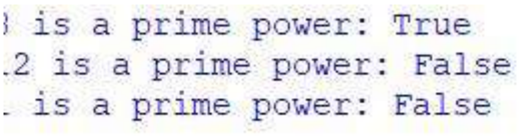
Logical thinking while dividing repeatedly

Better understanding of exponent properties

Modular programming



```
import math

def is_prime_power(n):
    """
    Checks if a number n is a prime power (p^k where p is prime and k >= 1).
    """
    if n <= 1:
        return False

    # Find the smallest prime factor, p
    p = -1
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            p = i
            break

    # If no factor was found within the loop, n itself is prime
    if p == -1:
        p = n

    # Check if n can be fully divided by only this single prime factor p
    temp_n = n
    while temp_n % p == 0:
        temp_n //= p

    # If temp_n becomes 1, then n was a power of p
    return temp_n == 1

# Examples:
print(f"8 is a prime power: {is_prime_power(8)}")        # 8 = 2^3 -> True
print(f"12 is a prime power: {is_prime_power(12)}")      # 12 = 2^2 * 3 -> False
print(f"1 is a prime power: {is_prime_power(1)}")        # False by definition
```



```
} is a prime power: True
.2 is a prime power: False
. is a prime power: False
```

Practical No: 18

Date: ___________

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

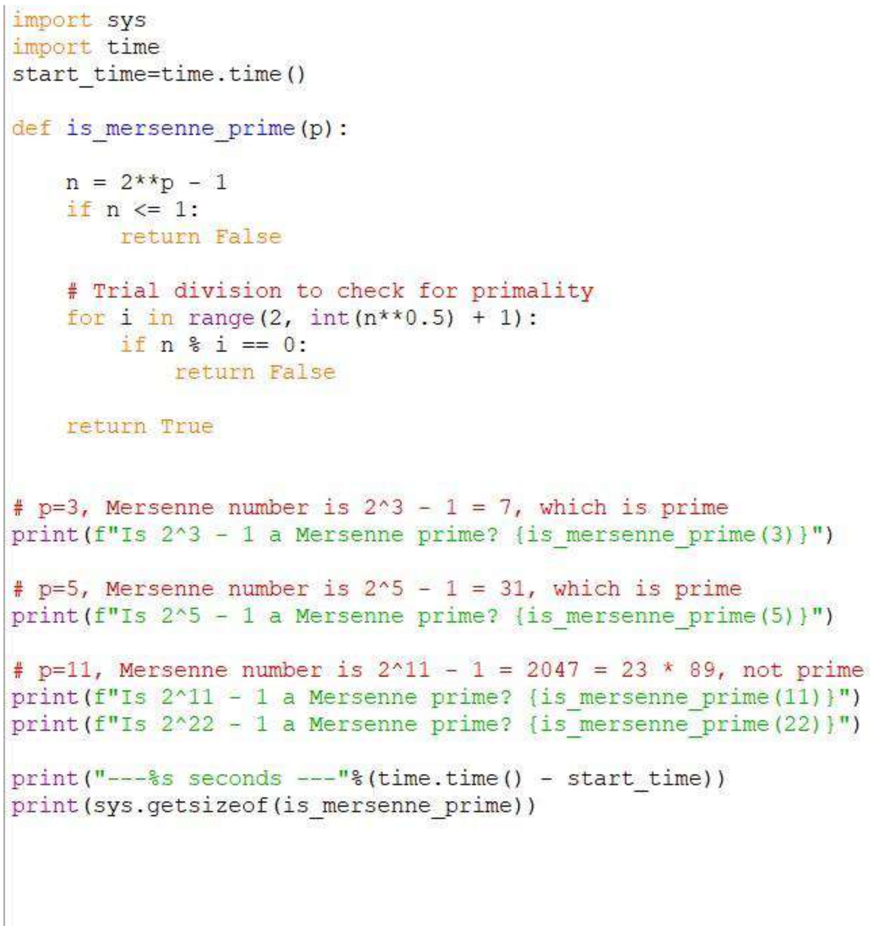
Prime checking becomes slow for big integers

Understanding the mathematics behind Mersenne primes

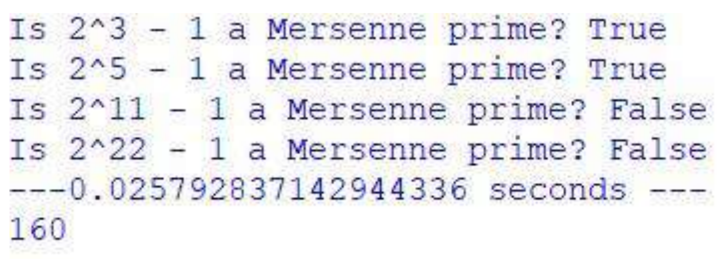
SKILLS ACHIEVED: Big number computation

Understanding exponential growth

Applying primality testing efficiently



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

def is_mersenne_prime(p):

    n = 2**p - 1
    if n <= 1:
        return False

    # Trial division to check for primality
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False

    return True


# p=3, Mersenne number is 2^3 - 1 = 7, which is prime
print(f"Is 2^3 - 1 a Mersenne prime? {is_mersenne_prime(3)}")

# p=5, Mersenne number is 2^5 - 1 = 31, which is prime
print(f"Is 2^5 - 1 a Mersenne prime? {is_mersenne_prime(5)}")

# p=11, Mersenne number is 2^11 - 1 = 2047 = 23 * 89, not prime
print(f"Is 2^11 - 1 a Mersenne prime? {is_mersenne_prime(11)}")
print(f"Is 2^22 - 1 a Mersenne prime? {is_mersenne_prime(22)}")

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(is_mersenne_prime))
```



```
Is 2^3 - 1 a Mersenne prime? True
Is 2^5 - 1 a Mersenne prime? True
Is 2^11 - 1 a Mersenne prime? False
Is 2^22 - 1 a Mersenne prime? False
---0.025792837142944336 seconds ---
160
```

Practical No: 19

Date: ___________

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

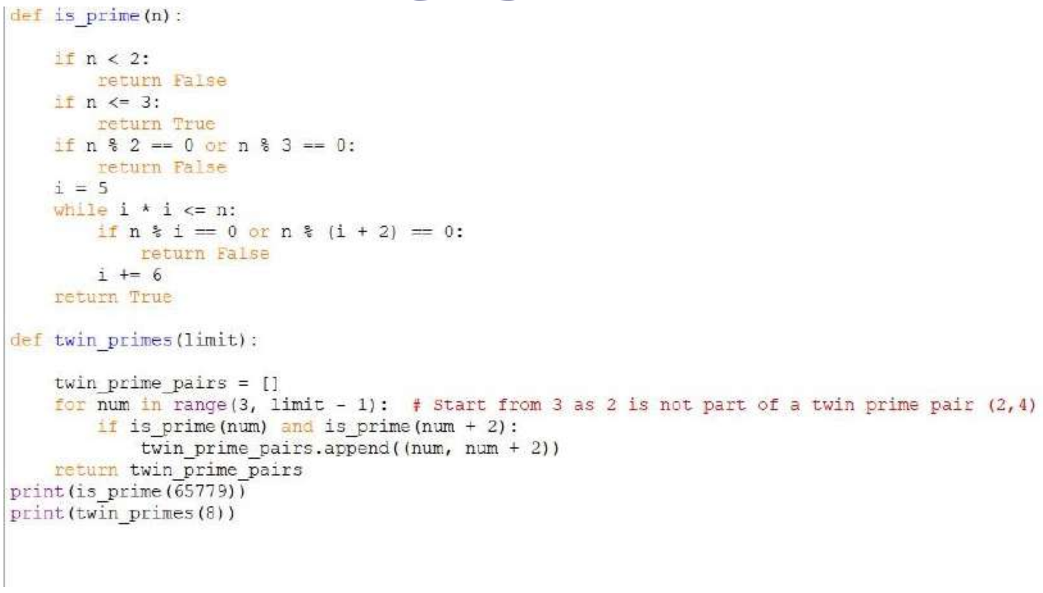
Handling increasing execution time for large limits

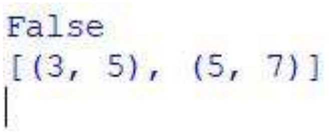
Off-by-one errors in loop boundaries

SKILLS ACHIEVED: Pattern recognition

Algorithm optimization

Efficient use of loops and conditionals


```
def is_prime(n):

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

def twin_primes(limit):

    twin_prime_pairs = []
    for num in range(3, limit - 1):  # Start from 3 as 2 is not part of a twin prime pair (2,4)
        if is_prime(num) and is_prime(num + 2):
            twin_prime_pairs.append((num, num + 2))
    return twin_prime_pairs
print(is_prime(65779))
print(twin_primes(8))
```


```
False
[(3, 5), (5, 7)]
```

Practical No: 20

Date: ___________

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

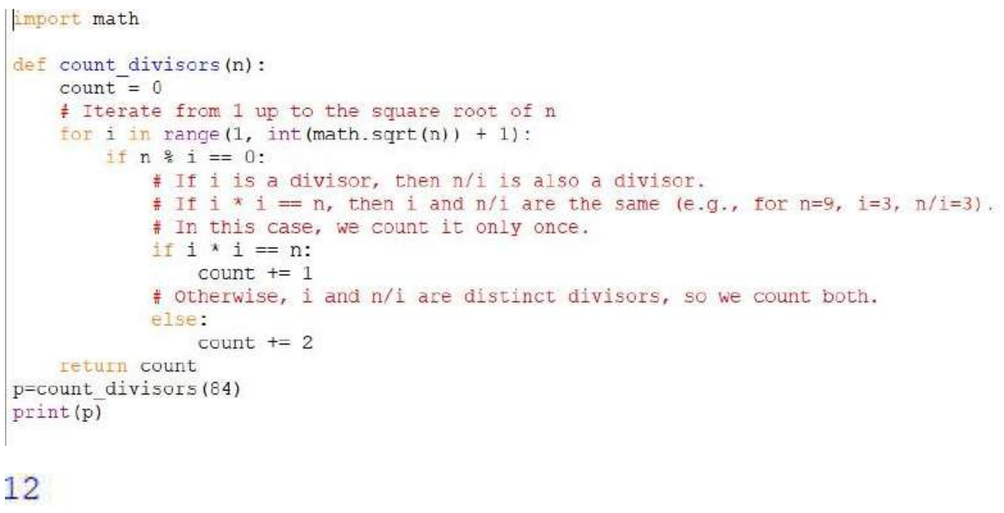
Understanding prime power factorization

Managing repeated loops

SKILLS ACHIEVED: Stronger number theory foundation

Efficient algorithm design

Breaking a problem into mathematical steps



```
import math

def count_divisors(n):
    count = 0
    # Iterate from 1 up to the square root of n
    for i in range(1, int(math.sqrt(n)) + 1):
        if n % i == 0:
            # If i is a divisor, then n/i is also a divisor.
            # If i * i == n, then i and n/i are the same (e.g., for n=9, i=3, n/i=3).
            # In this case, we count it only once.
            if i * i == n:
                count += 1
            # Otherwise, i and n/i are distinct divisors, so we count both.
            else:
                count += 2
    return count
p=count_divisors(84)
print(p)
```

```
12
```

Practical No: 21

Date: ___________

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.

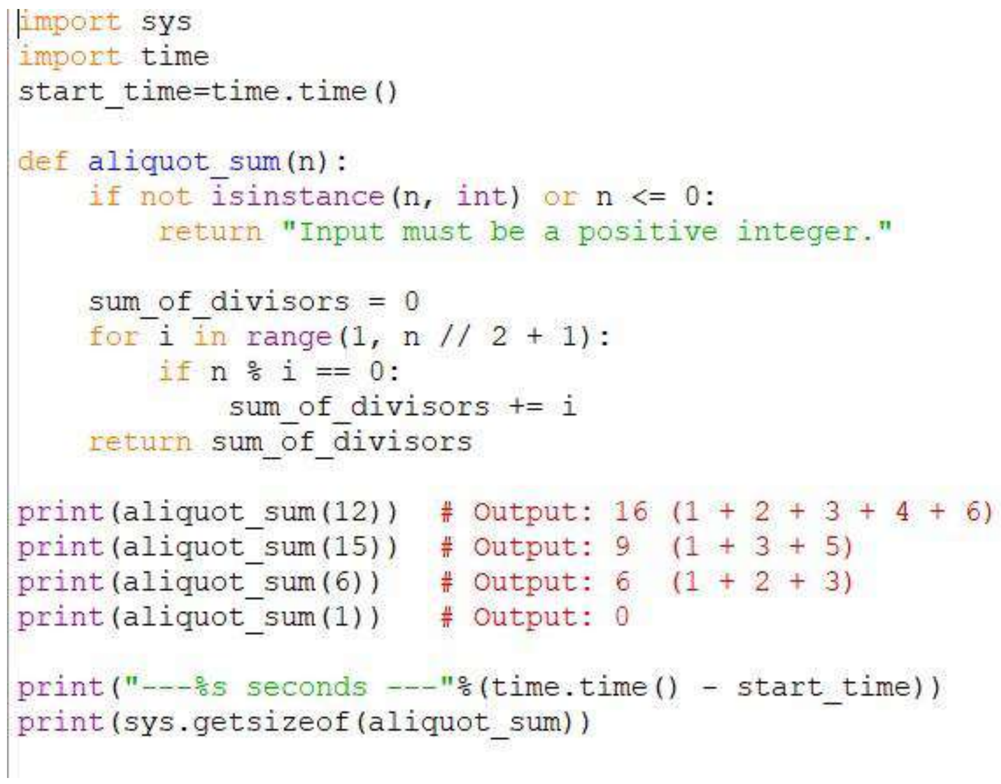
Ensuring that the number itself is not included in the sum

SKILLS ACHIEVED: Number theory basics

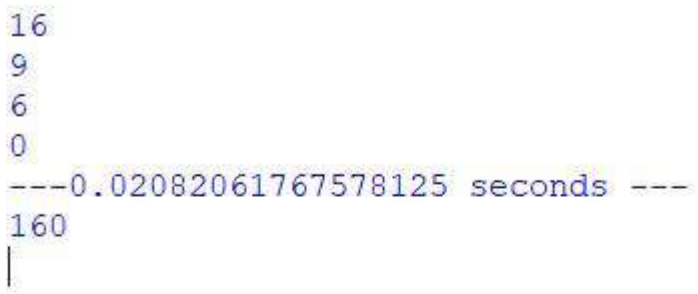
Logical thinking

Iteration concepts

Problem breakdown



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

def aliquot_sum(n):
    if not isinstance(n, int) or n <= 0:
        return "Input must be a positive integer."

    sum_of_divisors = 0
    for i in range(1, n // 2 + 1):
        if n % i == 0:
            sum_of_divisors += i
    return sum_of_divisors

print(aliquot_sum(12))  # Output: 16 (1 + 2 + 3 + 4 + 6)
print(aliquot_sum(15))  # Output: 9  (1 + 3 + 5)
print(aliquot_sum(6))   # Output: 6  (1 + 2 + 3)
print(aliquot_sum(1))   # Output: 0

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(aliquot_sum))
```



```
16
9
6
0
---0.020820617675781250 seconds ---
160
```

Practical No: 22

Date: ___________

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.

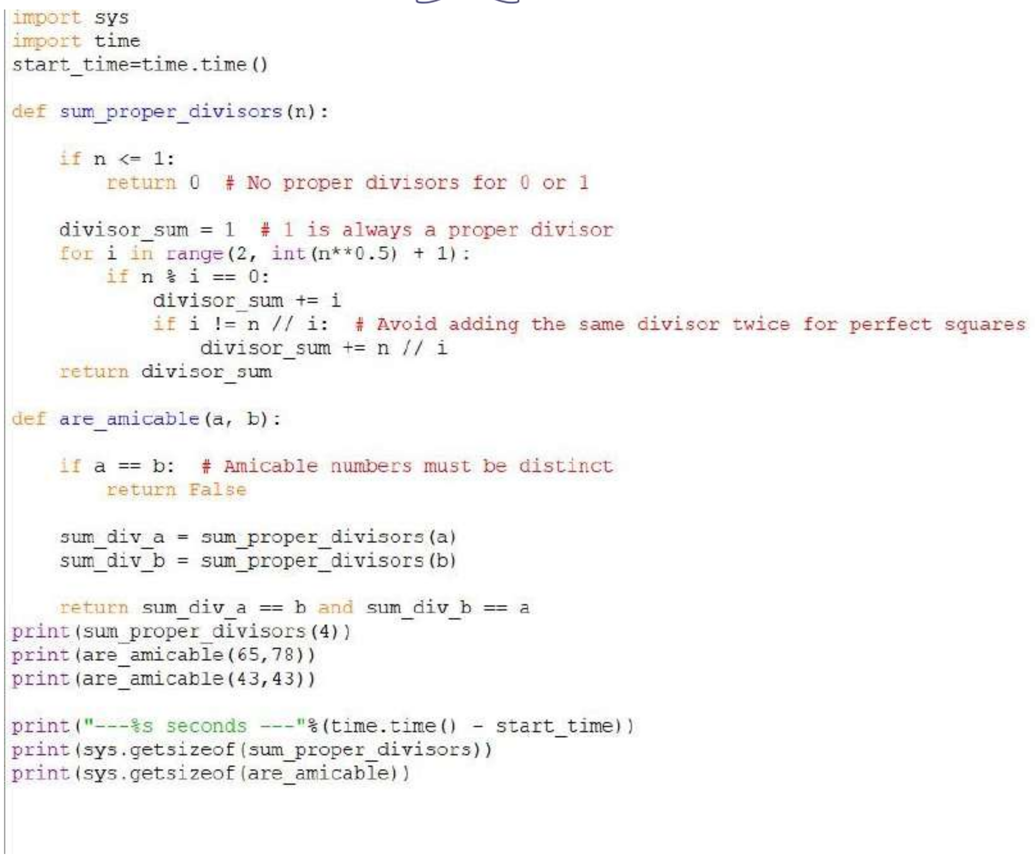
Ensuring the relationship works both ways.

SKILLS ACHIEVED: Functional thinking

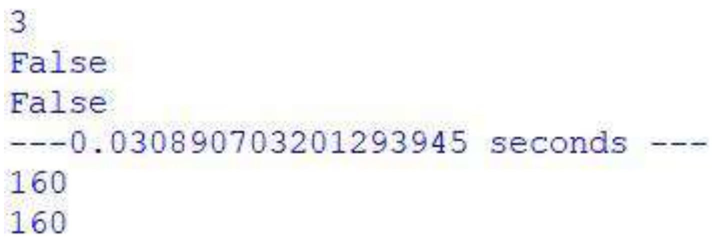
Reusability of code

Mathematical reasoning

Analytical skills



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

def sum_proper_divisors(n):

    if n <= 1:
        return 0  # No proper divisors for 0 or 1

    divisor_sum = 1  # 1 is always a proper divisor
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisor_sum += i
            if i != n // i:  # Avoid adding the same divisor twice for perfect squares
                divisor_sum += n // i
    return divisor_sum

def are_amicable(a, b):

    if a == b:  # Amicable numbers must be distinct
        return False

    sum_div_a = sum_proper_divisors(a)
    sum_div_b = sum_proper_divisors(b)

    return sum_div_a == b and sum_div_b == a
print(sum_proper_divisors(4))
print(are_amicable(65,78))
print(are_amicable(43,43))

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(sum_proper_divisors))
print(sys.getsizeof(are_amicable))
```



```
3
False
False
---0.030890703201293945 seconds ---
160
160
```

Practical No: 23

Date: ___________

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.

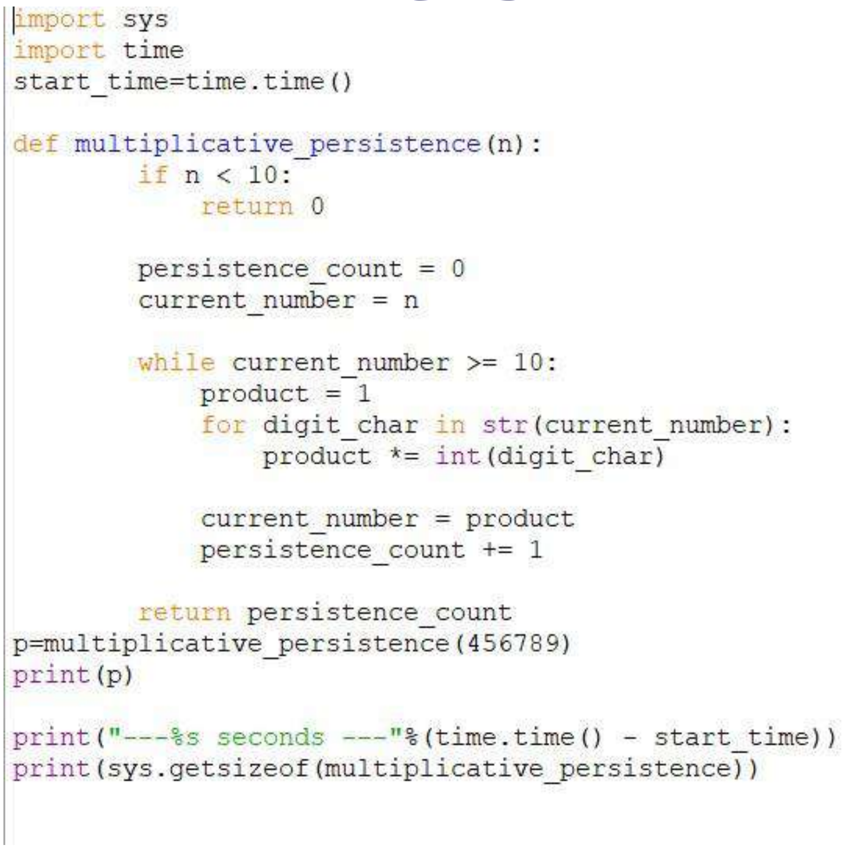
Properly handling multi-digit to single-digit transitions.

SKILLS ACHIEVED: Loop control

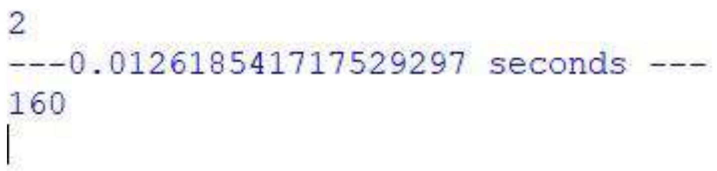
Mathematical modelling

Understanding of digit operations

Improving algorithmic flow



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

def multiplicative_persistence(n):
        if n < 10:
            return 0

        persistence_count = 0
        current_number = n

        while current_number >= 10:
            product = 1
            for digit_char in str(current_number):
                product *= int(digit_char)

            current_number = product
            persistence_count += 1

        return persistence_count
p=multiplicative_persistence(456789)
print(p)

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(multiplicative_persistence))
```



```
2
---0.012618541717529297 seconds ---
160
```

Practical No: 24

Date: ___________

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.

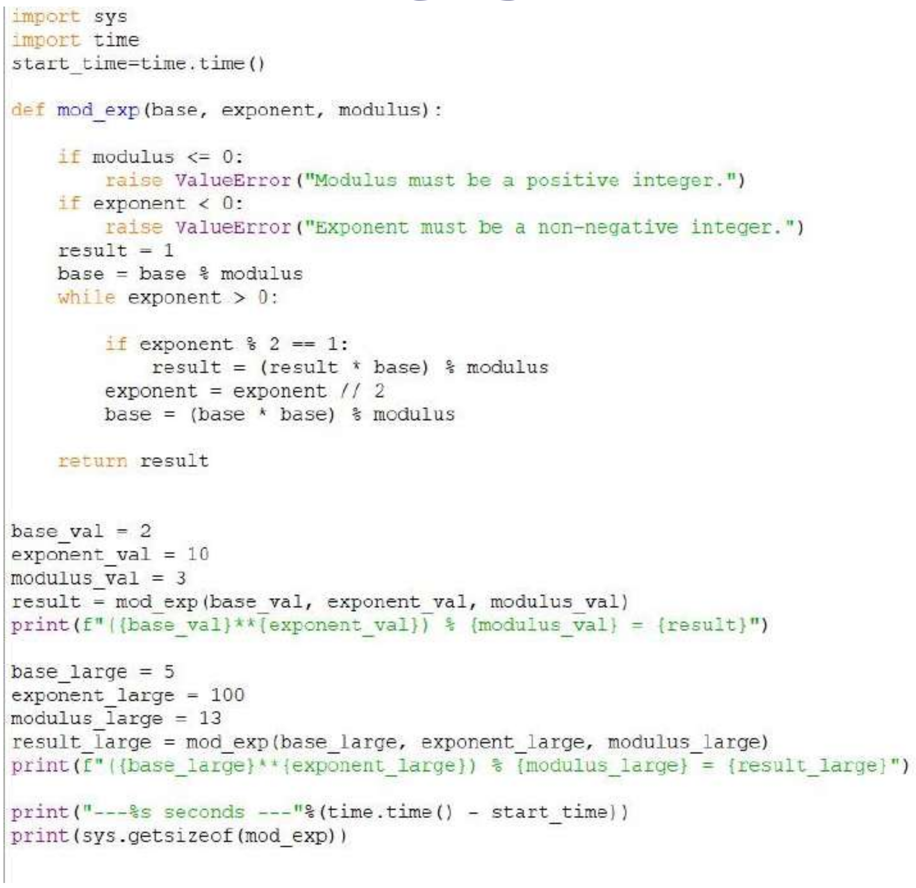
Understanding what “more divisors than any smaller number” fully means.

SKILLS ACHIEVED: Divisor analysis

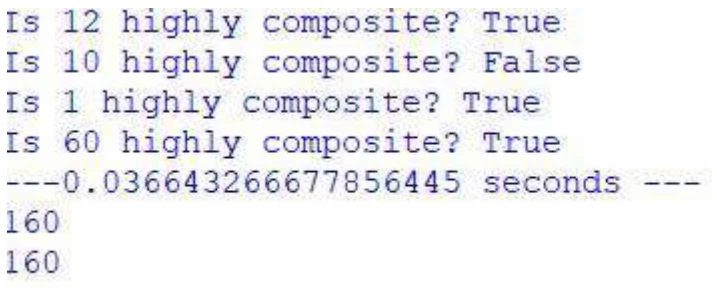
Comparison logic

Mathematical pattern observation

Breaking large tasks into smaller checks



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

def mod_exp(base, exponent, modulus):

    if modulus <= 0:
        raise ValueError("Modulus must be a positive integer.")
    if exponent < 0:
        raise ValueError("Exponent must be a non-negative integer.")
    result = 1
    base = base % modulus
    while exponent > 0:

        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus

    return result


base_val = 2
exponent_val = 10
modulus_val = 3
result = mod_exp(base_val, exponent_val, modulus_val)
print(f"({base_val}**{exponent_val}) % {modulus_val} = {result}")

base_large = 5
exponent_large = 100
modulus_large = 13
result_large = mod_exp(base_large, exponent_large, modulus_large)
print(f"({base_large}**{exponent_large}) % {modulus_large} = {result_large}")

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(mod_exp))
```



```
Is 12 highly composite? True
Is 10 highly composite? False
Is 1 highly composite? True
Is 60 highly composite? True
---0.036643266677856445 seconds ---
160
160
```

Practical No: 25

Date: ___________

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.

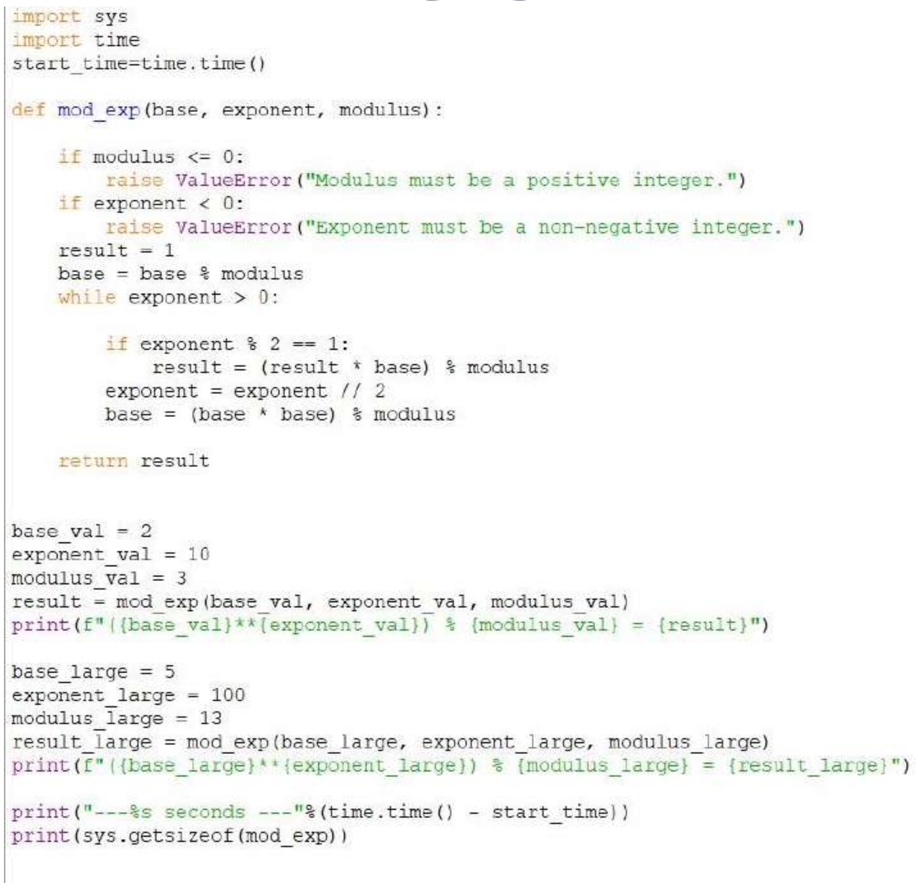
Managing exponent reduction correctly.

SKILLS ACHIEVED: Cryptographic mathematical skills

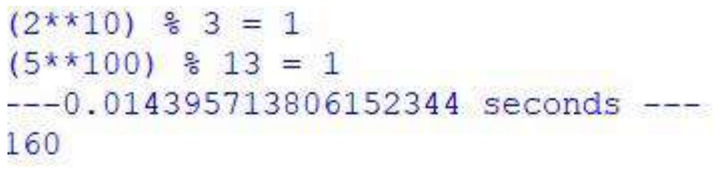
Understanding modular arithmetic

Efficient algorithm design

Problem-solving mindset



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

def mod_exp(base, exponent, modulus):

    if modulus <= 0:
        raise ValueError("Modulus must be a positive integer.")
    if exponent < 0:
        raise ValueError("Exponent must be a non-negative integer.")
    result = 1
    base = base % modulus
    while exponent > 0:

        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus

    return result


base_val = 2
exponent_val = 10
modulus_val = 3
result = mod_exp(base_val, exponent_val, modulus_val)
print(f"({base_val}**{exponent_val}) % {modulus_val} = {result}")

base_large = 5
exponent_large = 100
modulus_large = 13
result_large = mod_exp(base_large, exponent_large, modulus_large)
print(f"({base_large}**{exponent_large}) % {modulus_large} = {result_large}")

print("---%s seconds ---"%(time.time() - start_time))
print(sys.getsizeof(mod_exp))
```



```
(2**10) % 3 = 1
(5**100) % 13 = 1
---0.014395713806152344 seconds ---
160
```

Practical No: 26

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:

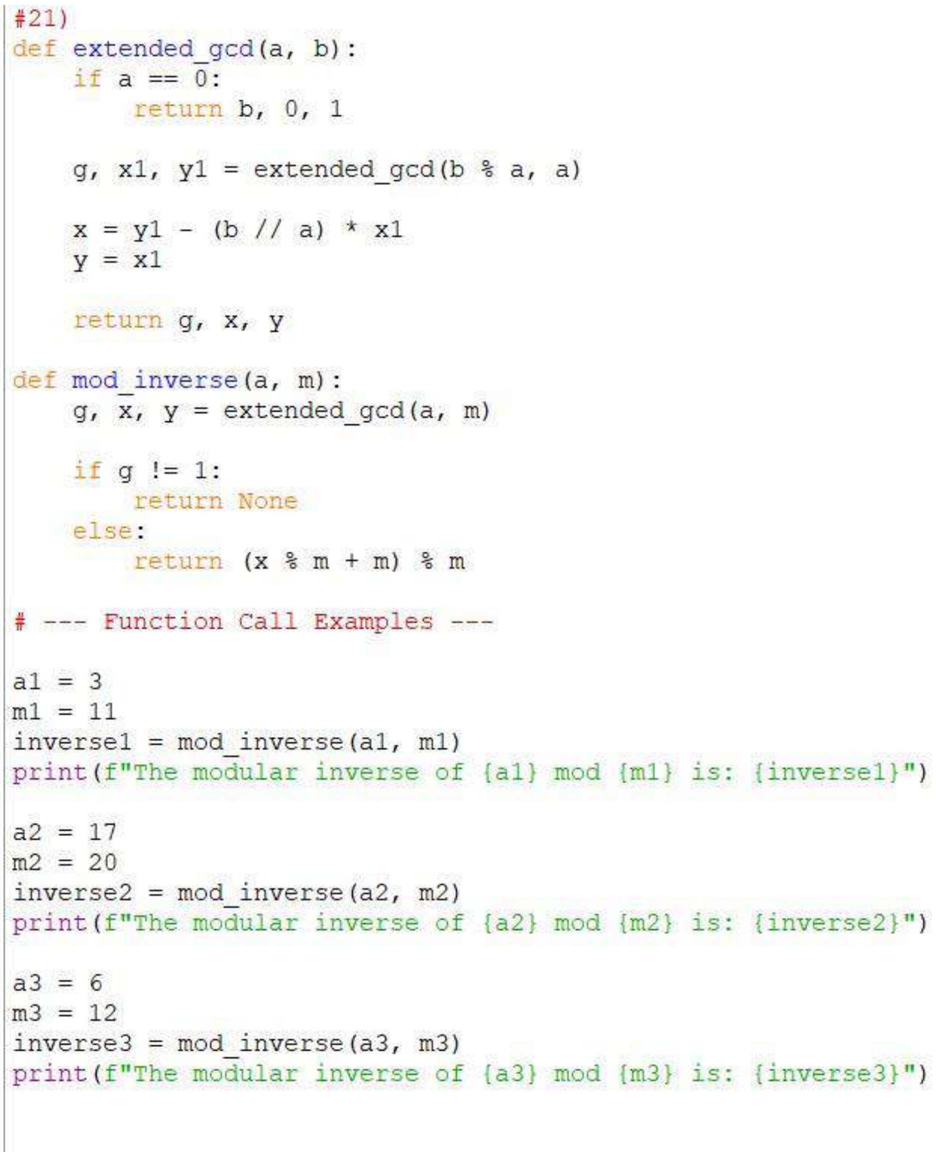
- The implemented function accurately computes the modular inverse for all valid (a, m) pairs, demonstrating correctness by ensuring $(a \times x)\%m = 1$.
- For example, mod_inverse(3, 11) returns 4 because $3 \times 4 = 12 \equiv 1(\text{mod}11)$.
- The function signals appropriately when no inverse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

DIFFICULTY FACED BY STUDENT:

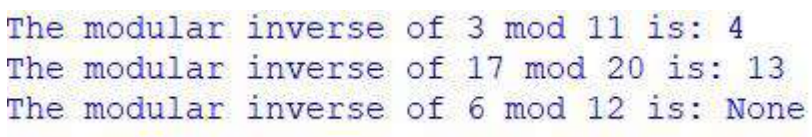
- Ensuring gcd(a, m) = 1 is critical; missing this leads to incorrect outputs or unhandled exceptions.
- When using the Extended Euclidean Algorithm, initial solutions for x can be negative. Adjusting x into the range 0 to m-1 is non-trivial for beginners.
- For very large values of m, iterative solutions 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
#21)
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:
        return None
    else:
        return (x % m + m) % m

# --- Function Call Examples ---

a1 = 3
m1 = 11
inverse1 = mod_inverse(a1, m1)
print(f"The modular inverse of {a1} mod {m1} is: {inverse1}")

a2 = 17
m2 = 20
inverse2 = mod_inverse(a2, m2)
print(f"The modular inverse of {a2} mod {m2} is: {inverse2}")

a3 = 6
m3 = 12
inverse3 = mod_inverse(a3, m3)
print(f"The modular inverse of {a3} mod {m3} is: {inverse3}")
```



```
The modular inverse of 3 mod 11 is: 4
The modular inverse of 17 mod 20 is: 13
The modular inverse of 6 mod 12 is: None
```

Practical No: 27

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:

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

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

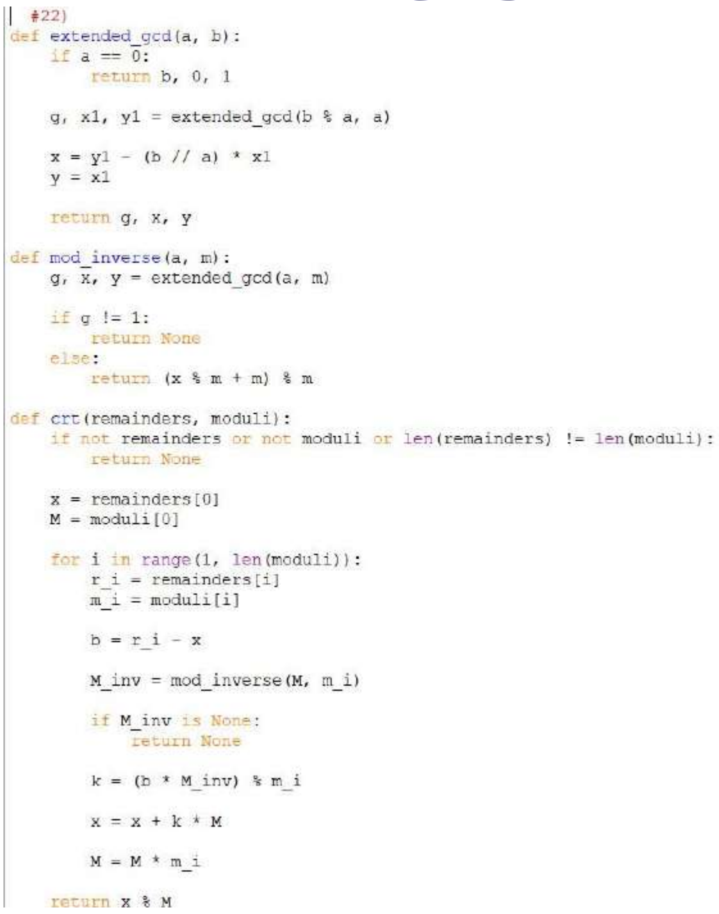
DIFFICULTY FACED BY STUDENT:

- 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).

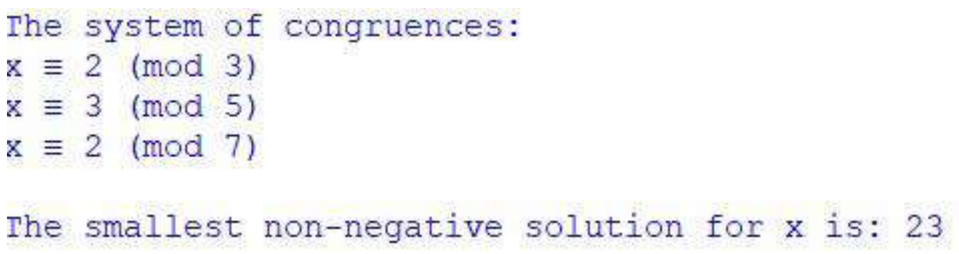
- Overflow and large integer calculations for big products of moduli.
- Edge case handling for empty input, single congruence, or moduli with repeated values.

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.



```
#22)
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:
        return None
    else:
        return (x % m + m) % m

def crt(remainders, moduli):
    if not remainders or not moduli or len(remainders) != len(moduli):
        return None

    x = remainders[0]
    M = moduli[0]

    for i in range(1, len(moduli)):
        r_i = remainders[i]
        m_i = moduli[i]

        b = r_i - x

        M_inv = mod_inverse(M, m_i)

        if M_inv is None:
            return None

        k = (b * M_inv) % m_i

        x = x + k * M

        M = M * m_i

    return x % M
```



```
The system of congruences:
x ≡ 2 (mod 3)
x ≡ 3 (mod 5)
x ≡ 2 (mod 7)

The smallest non-negative solution for x is: 23
```

Practical No: 28

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:

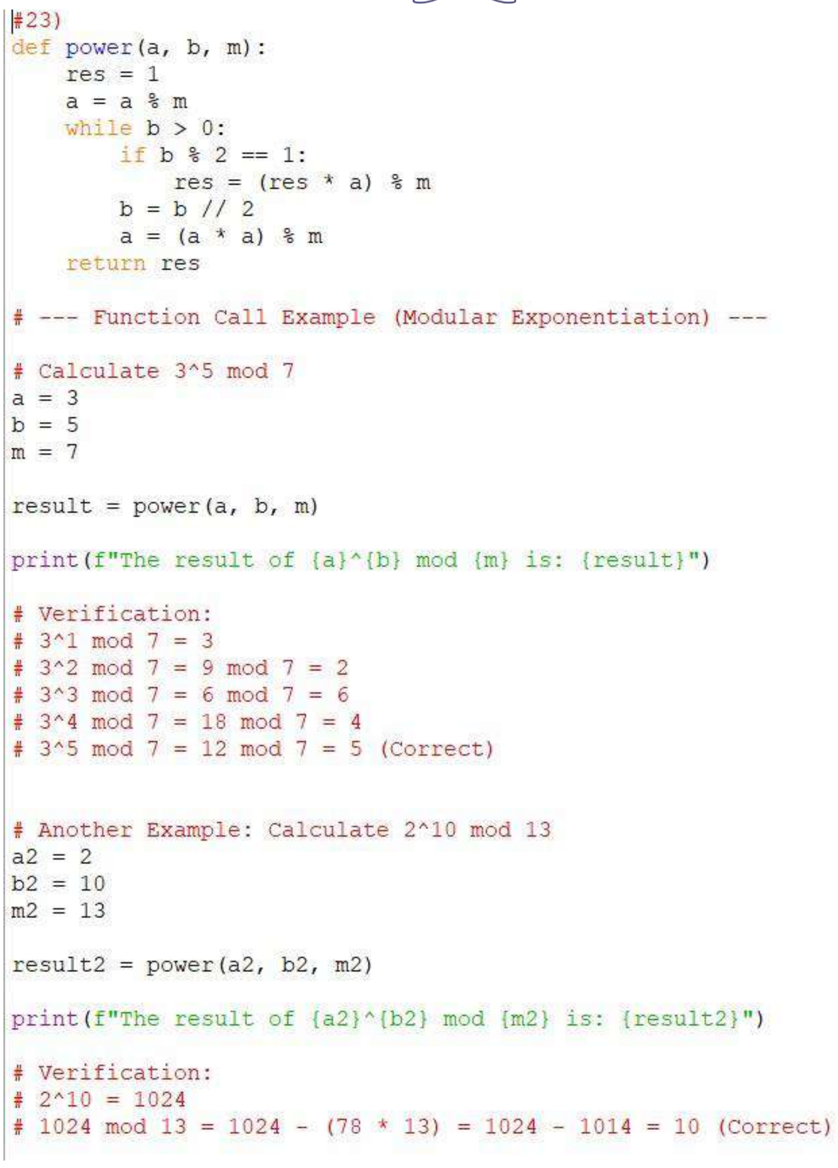
- The function accurately outputs True if $x^2 \equiv a(\mathrm{mod} p)$ has at least one solution, otherwise False.
- For example, $is\_quadratic\_residue(10,13)$ returns True since $6^2 = 36 \equiv 10(\mathrm{mod} 13)$.
- The procedure is mathematically sound, efficient, and scalable for large prime p.

DIFFICULTY FACED BY STUDENT:

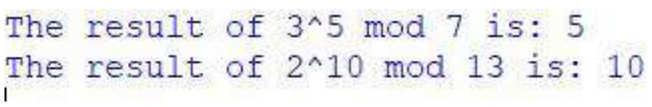
- Correctly implementing modular exponentiation to handle large p efficiently.
- Handling edge cases where $a \equiv 0 \bmod p$.
- Understanding and applying Euler's criterion rather than naive trial and error checking.

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.



```python
#23)
def power(a, b, m):
    res = 1
    a = a % m
    while b > 0:
        if b % 2 == 1:
            res = (res * a) % m
        b = b // 2
        a = (a * a) % m
    return res

# --- Function Call Example (Modular Exponentiation) ---

# Calculate 3^5 mod 7
a = 3
b = 5
m = 7

result = power(a, b, m)

print(f"The result of {a}^{b} mod {m} is: {result}")

# Verification:
# 3^1 mod 7 = 3
# 3^2 mod 7 = 9 mod 7 = 2
# 3^3 mod 7 = 6 mod 7 = 6
# 3^4 mod 7 = 18 mod 7 = 4
# 3^5 mod 7 = 12 mod 7 = 5 (Correct)


# Another Example: Calculate 2^10 mod 13
a2 = 2
b2 = 10
m2 = 13

result2 = power(a2, b2, m2)

print(f"The result of {a2}^{b2} mod {m2} is: {result2}")

# Verification:
# 2^10 = 1024
# 1024 mod 13 = 1024 - (78 * 13) = 1024 - 1014 = 10 (Correct)
```



```
The result of 3^5 mod 7 is: 5
The result of 2^10 mod 13 is: 10
```

Practical No: 29

VIT BHOPAL

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 $a^k \equiv 1 (\bmod n)$.
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.

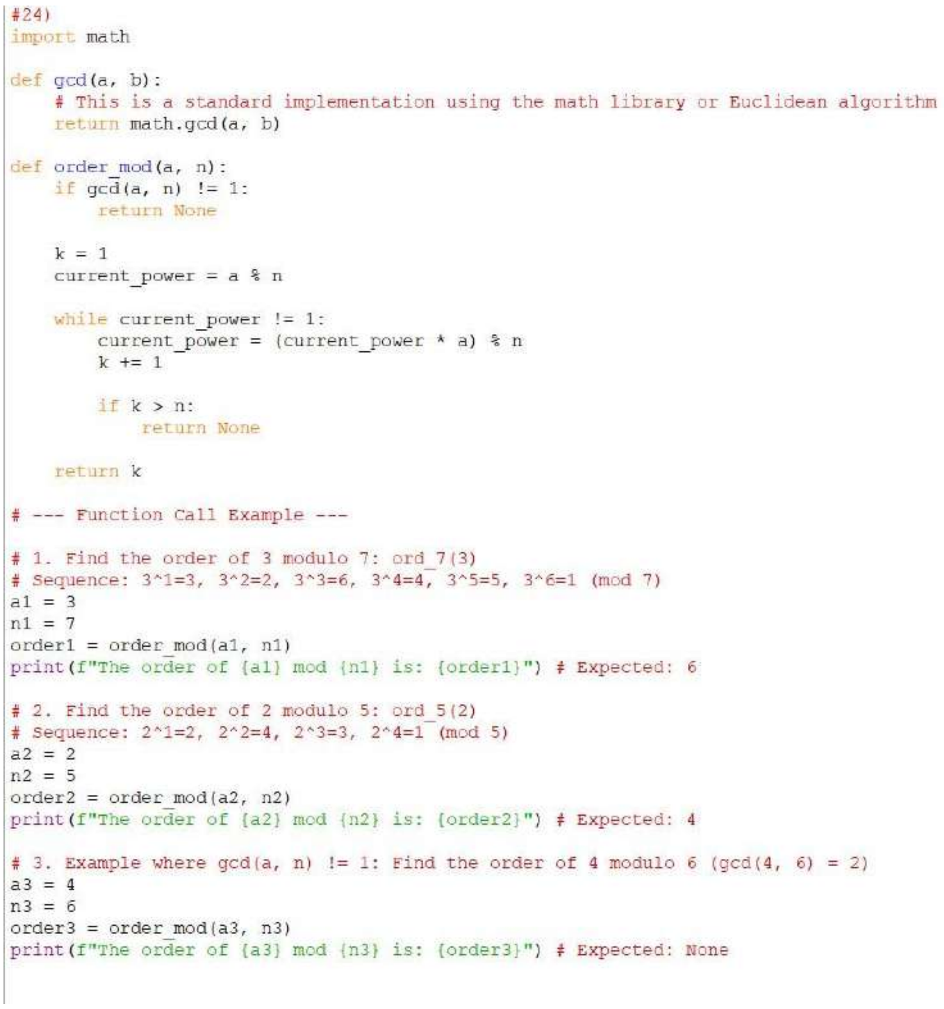
DIFFICULTY FACED BY STUDENT:

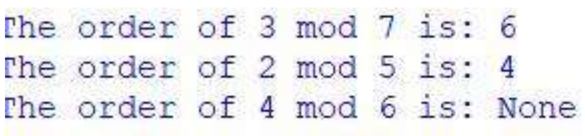
- Handling cases where a and n are not coprime, as the order may not be defined.
- Large values of n require optimizations in modular exponentiation to maintain efficiency.
- Using Euler's totient to limit possible orders requires an additional function or precomputation which adds complexity.
- Ensuring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


```python
#24)
import math

def gcd(a, b):
    # This is a standard implementation using the math library or Euclidean algorithm
    return math.gcd(a, b)

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

    k = 1
    current_power = a % n

    while current_power != 1:
        current_power = (current_power * a) % n
        k += 1

        if k > n:
            return None

    return k

# --- Function Call Example ---

# 1. Find the order of 3 modulo 7: ord_7(3)
# Sequence: 3^1=3, 3^2=2, 3^3=6, 3^4=4, 3^5=5, 3^6=1 (mod 7)
a1 = 3
n1 = 7
order1 = order_mod(a1, n1)
print(f"The order of {a1} mod {n1} is: {order1}") # Expected: 6

# 2. Find the order of 2 modulo 5: ord_5(2)
# Sequence: 2^1=2, 2^2=4, 2^3=3, 2^4=1 (mod 5)
a2 = 2
n2 = 5
order2 = order_mod(a2, n2)
print(f"The order of {a2} mod {n2} is: {order2}") # Expected: 4

# 3. Example where gcd(a, n) != 1: Find the order of 4 modulo 6 (gcd(4, 6) = 2)
a3 = 4
n3 = 6
order3 = order_mod(a3, n3)
print(f"The order of {a3} mod {n3} is: {order3}") # Expected: None
```


```
The order of 3 mod 7 is: 6
The order of 2 mod 5 is: 4
The order of 4 mod 6 is: None
```

Practical No: 30

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:

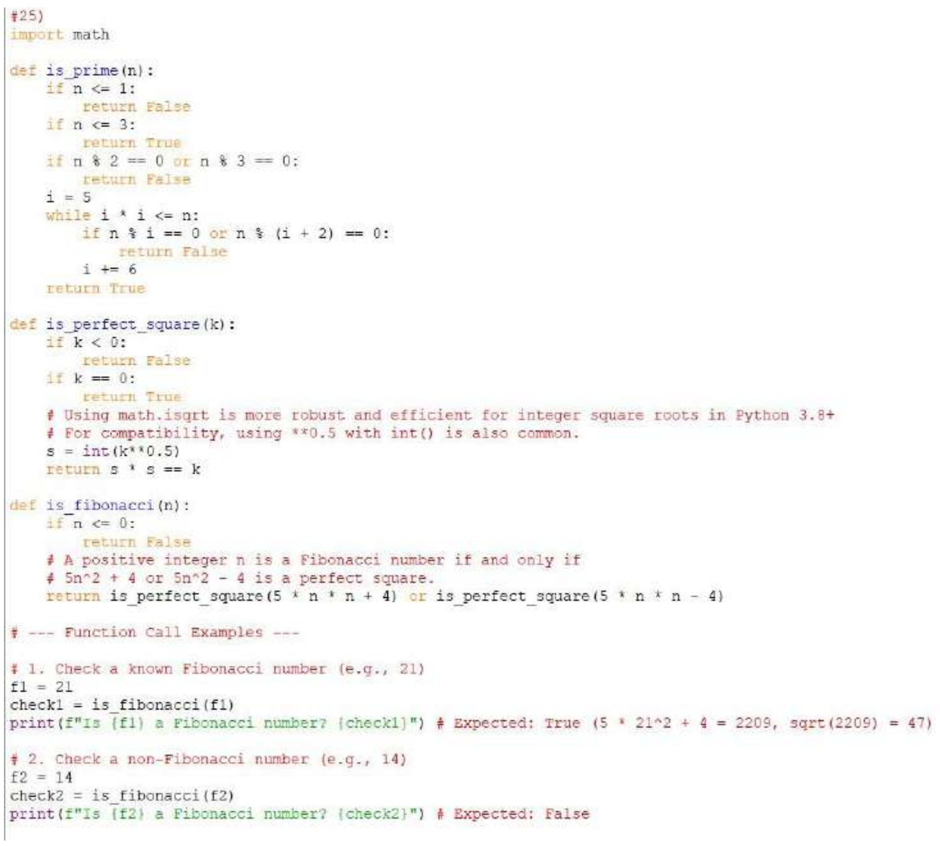
- The function quickly determines the status for any input n.
- For example, $is\_fibonacci\_prime(13)$ returns True because 13 is both a prime and a Fibonacci number.
- For $n = 12$, it returns False because although 12 is near a Fibonacci number, it is neither prime nor Fibonacci.

DIFFICULTY FACED BY STUDENT:

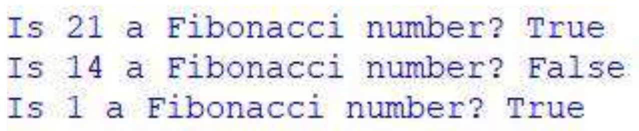
- Implementing a primality test efficient enough to handle large inputs robustly.
- Avoiding generation of the Fibonacci sequence via iteration or recursion to keep performance optimal.
- Verifying perfect squares without floating-point inaccuracies, ensuring correct detection of Fibonacci numbers.

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.



```python
#25)
import math

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

def is_perfect_square(k):
    if k < 0:
        return False
    if k == 0:
        return True
    # Using math.isqrt is more robust and efficient for integer square roots in Python 3.8+
    # For compatibility, using **0.5 with int() is also common.
    s = int(k**0.5)
    return s * s == k

def is_fibonacci(n):
    if n <= 0:
        return False
    # A positive integer n is a Fibonacci number if and only if
    # 5n^2 + 4 or 5n^2 - 4 is a perfect square.
    return is_perfect_square(5 * n * n + 4) or is_perfect_square(5 * n * n - 4)

# --- Function Call Examples ---

# 1. Check a known Fibonacci number (e.g., 21)
f1 = 21
check1 = is_fibonacci(f1)
print(f"Is {f1} a Fibonacci number? {check1}") # Expected: True (5 * 21^2 + 4 = 2209, sqrt(2209) = 47)

# 2. Check a non-Fibonacci number (e.g., 14)
f2 = 14
check2 = is_fibonacci(f2)
print(f"Is {f2} a Fibonacci number? {check2}") # Expected: False
```



```
Is 21 a Fibonacci number? True
Is 14 a Fibonacci number? False
Is 1 a Fibonacci number? True
```

Practical No: 31

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:

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

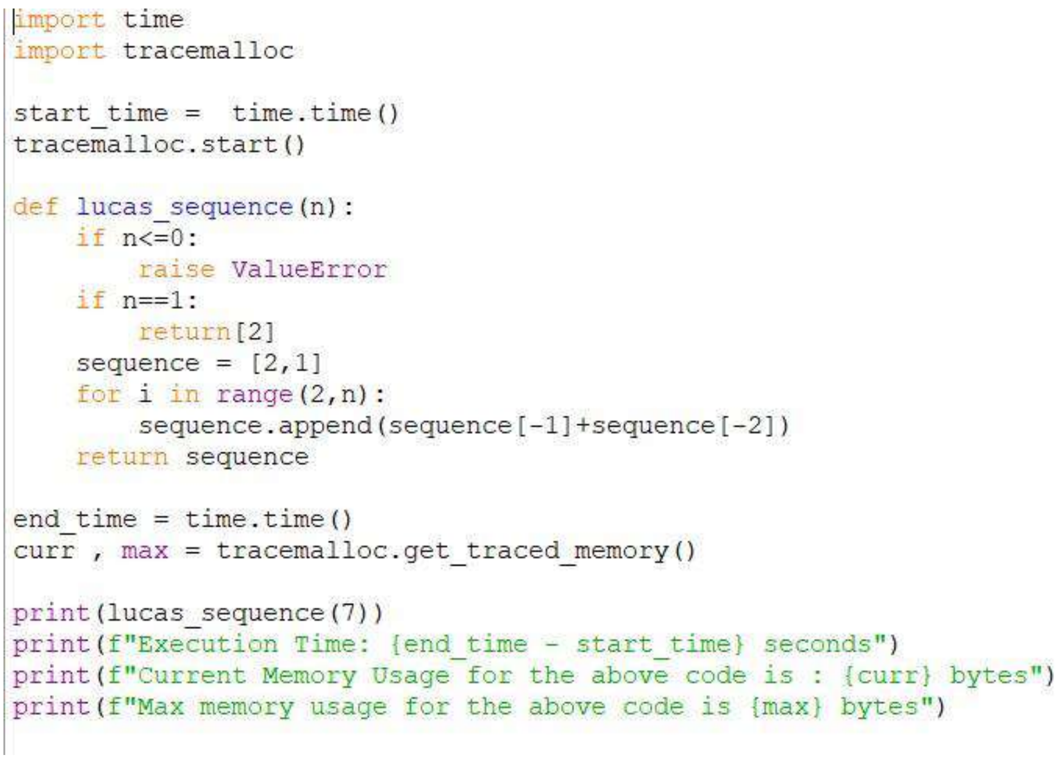
- Efficient for even large n, since the approach is linear in time.

DIFFICULTY FACED BY STUDENT:

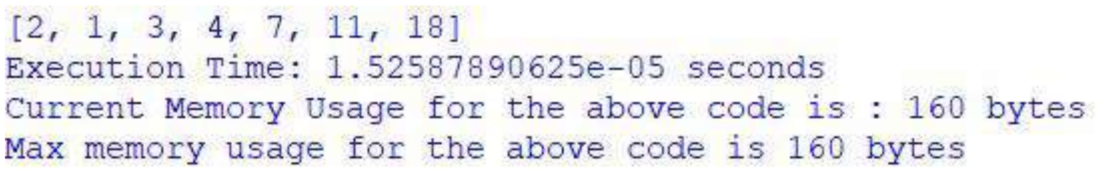
- Handling the base cases correctly as they differ from Fibonacci.
- Ensuring accuracy and efficiency without recursion overhead.
- Managing large values and integer overflow in some languages (Python inherently supports big integers).

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 tracemalloc

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

def lucas_sequence(n):
    if n<=0:
        raise ValueError
    if n==1:
        return[2]
    sequence = [2,1]
    for i in range(2,n):
        sequence.append(sequence[-1]+sequence[-2])
    return sequence

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(lucas_sequence(7))
print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
[2, 1, 3, 4, 7, 11, 18]
Execution Time: 1.52587890625e-05 seconds
Current Memory Usage for the above code is : 160 bytes
Max memory usage for the above code is 160 bytes
```

Practical No: 32

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:

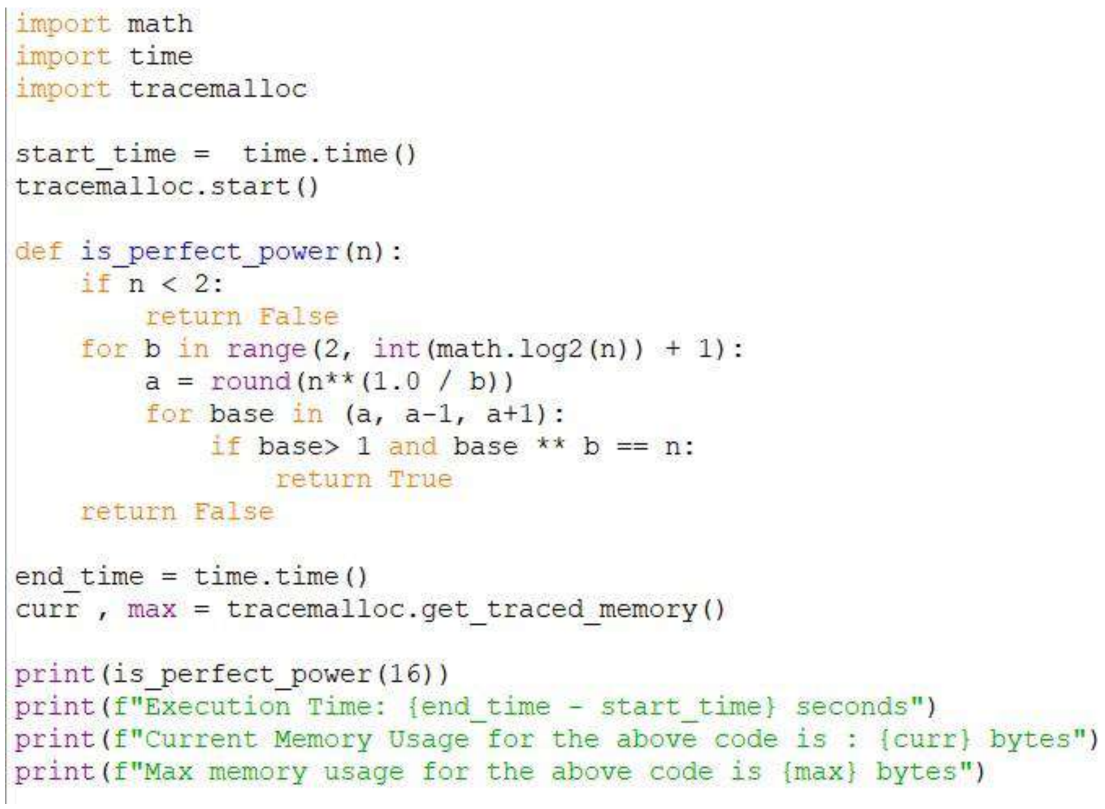
- The function correctly identifies perfect powers for various inputs, both small and large.
- For instance, is_perfect_power(64) returns True (since $2^6 = 64$), while is_perfect_power(70) returns False.
- Efficient checking is achieved by limiting the exponent range to $\log_2(n)$.

DIFFICULTY FACED BY STUDENT:

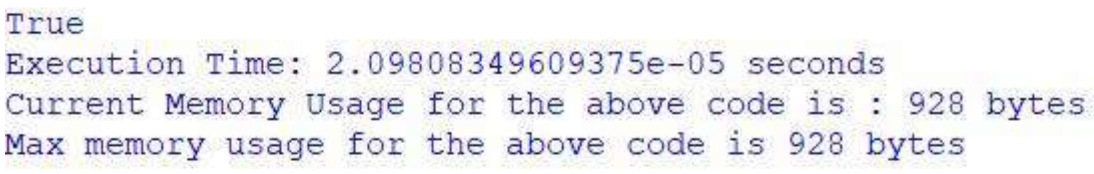
- Handling floating-point inaccuracies in root calculation.
- Efficiently iterating only up to $\log_2(n)$ exponents.
- Dealing with very large inputs where precision and performance are concerns.

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.



```python
import math
import time
import tracemalloc

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

def is_perfect_power(n):
    if n < 2:
        return False
    for b in range(2, int(math.log2(n)) + 1):
        a = round(n**(1.0 / b))
        for base in (a, a-1, a+1):
            if base> 1 and base ** b == n:
                return True
    return False

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(is_perfect_power(16))
print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
True
Execution Time: 2.09808349609375e-05 seconds
Current Memory Usage for the above code is : 928 bytes
Max memory usage for the above code is 928 bytes
```

Practical No: 33

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:

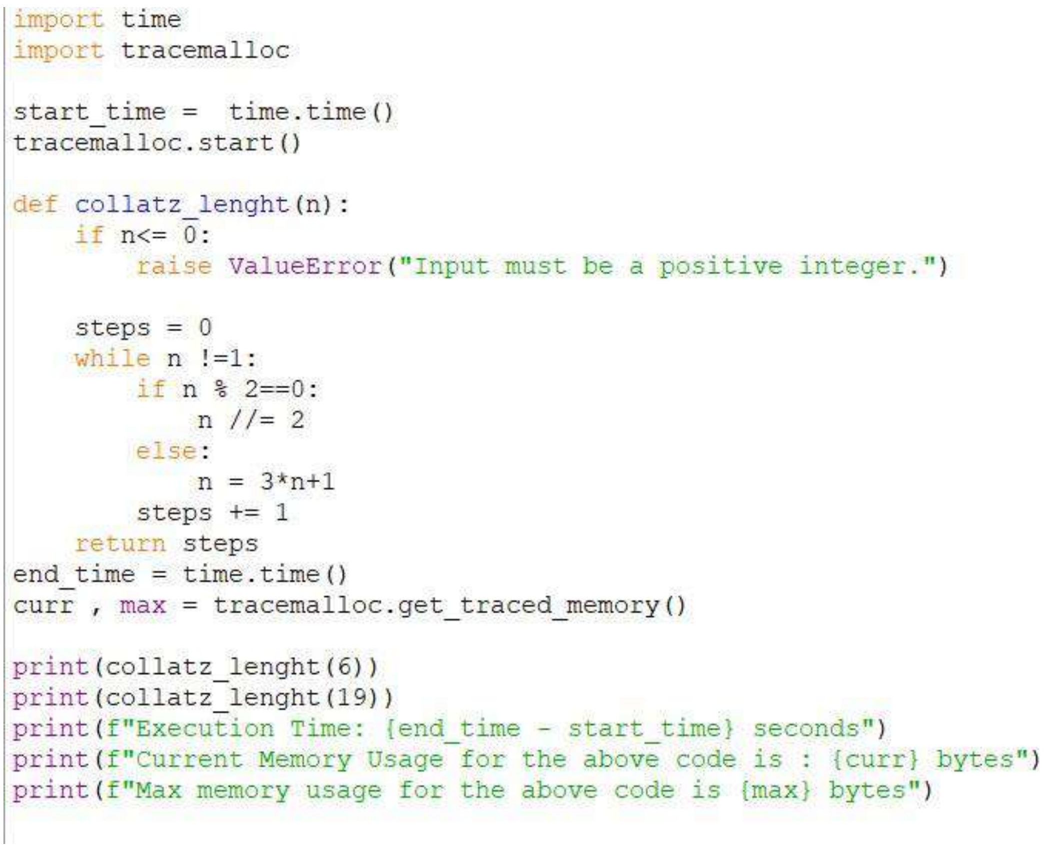
- The implemented function accurately returns the number of steps for various tested inputs.
- For example, collatz_length(6) returns 8 because the sequence from 6 is:
  $6 \rightarrow 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ (8 steps).
-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

DIFFICULTY FACED BY STUDENT:

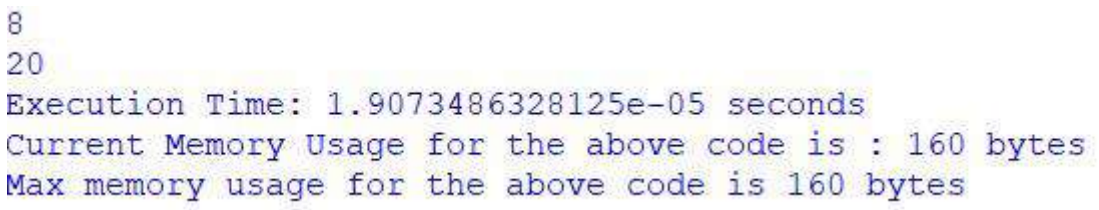
- Handling large values of n efficiently (can be mitigated using memoization).
- Avoiding infinite loops in theoretical edge cases if the conjecture were false (currently still unproven but widely believed true).
- Ensuring correct increment and termination conditions.

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 tracemalloc

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

def collatz_lenght(n):
    if n<= 0:
        raise ValueError("Input must be a positive integer.")

    steps = 0
    while n !=1:
        if n % 2==0:
            n //= 2
        else:
            n = 3*n+1
        steps += 1
    return steps
end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(collatz_lenght(6))
print(collatz_lenght(19))
print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
8
20
Execution Time: 1.9073486328125e-05 seconds
Current Memory Usage for the above code is : 160 bytes
Max memory usage for the above code is 160 bytes
```

Practical No: 34

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:

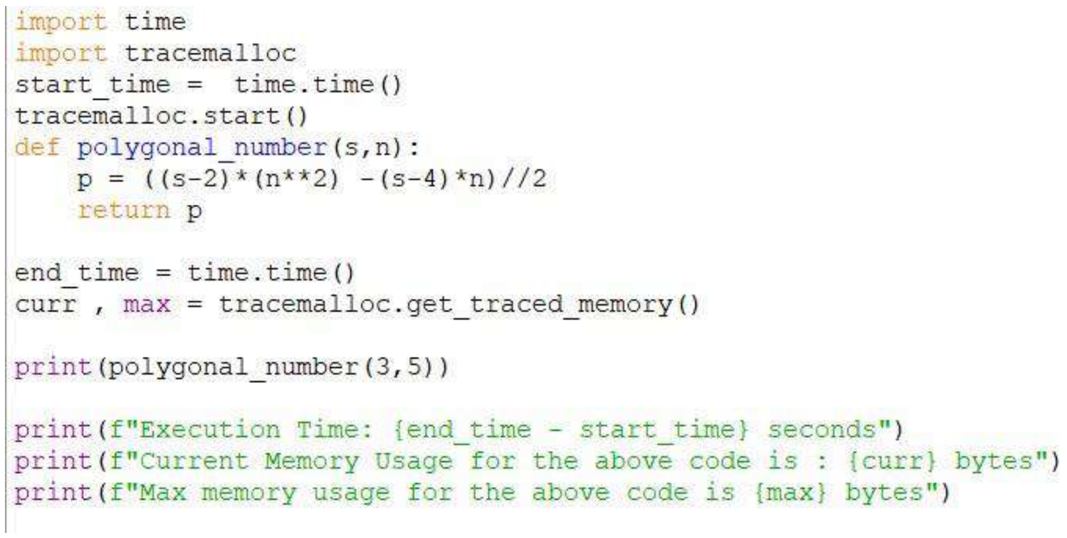
- For example, polygonal_number(3, 5) computes the 5th triangular number, yielding 15.
- The function returns correct polygonal numbers for any valid s and n, demonstrating generalization and formula correctness.
- Computational complexity is constant time $O(1)$, as it is a direct formula evaluation.

DIFFICULTY FACED BY STUDENT:

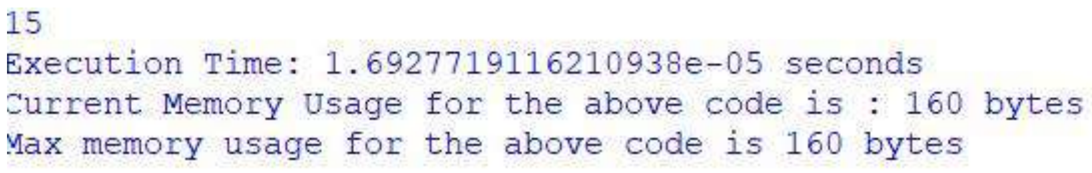
- Ensuring formula correctness for different polygonal orders.
- Handling invalid inputs such as $s < 3$ or $n \leq 0$.
- Understanding geometric interpretation underlying the algebraic formula.

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.



```
import time
import tracemalloc
start_time =  time.time()
tracemalloc.start()
def polygonal_number(s,n):
    p = ((s-2)*(n**2) -(s-4)*n)//2
    return p

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(polygonal_number(3,5))

print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
15
Execution Time: 1.6927719116210938e-05 seconds
Current Memory Usage for the above code is : 160 bytes
Max memory usage for the above code is 160 bytes
```

Practical No: 35

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:

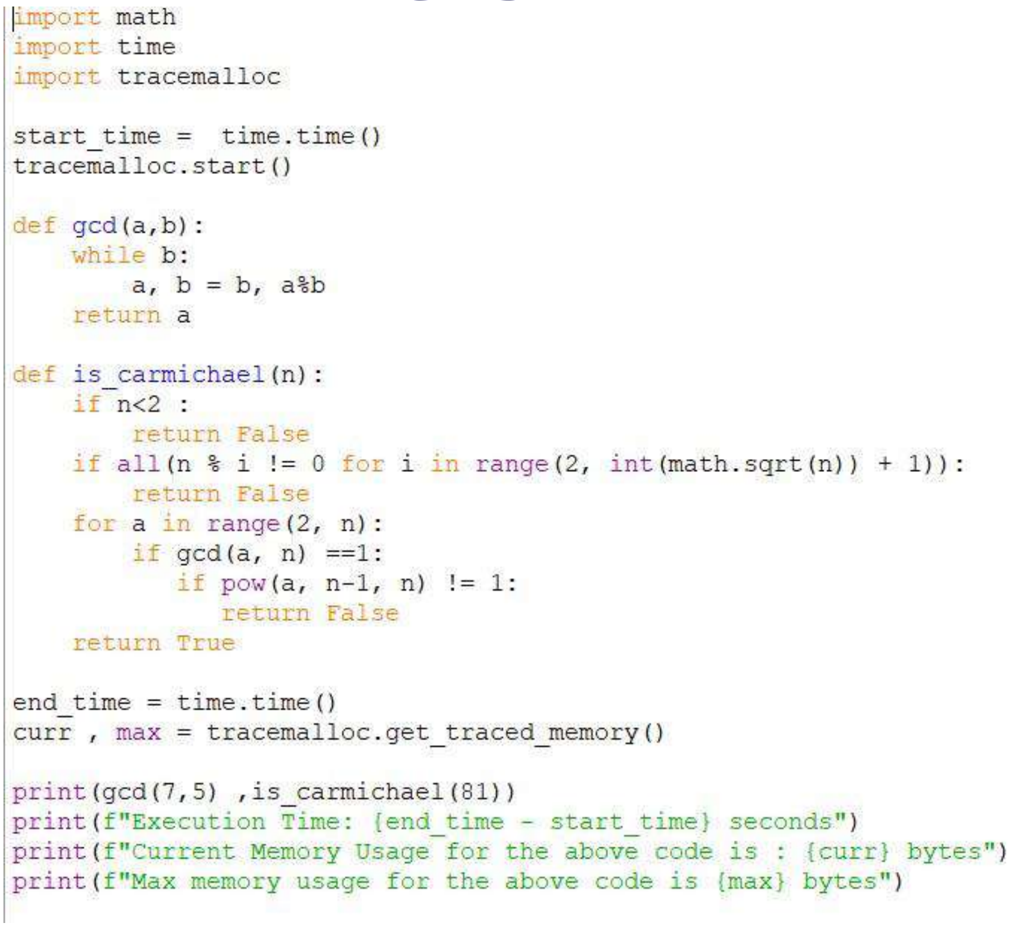
- The function correctly identifies known Carmichael numbers such as 561, 1105, and 1729.
- It efficiently distinguishes non-Carmichael composites and primes.
- Optimized implementations using Korselt's criterion are computationally faster for larger inputs.

DIFFICULTY FACED BY STUDENT:

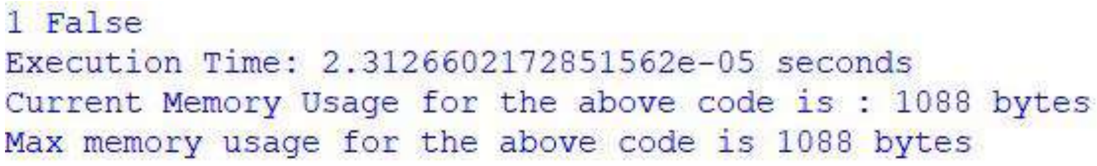
- Factorizing n efficiently to test Korselt's criterion for large numbers.
- Performing modular exponentiation for many bases if brute force used, leading to performance issues.
- Managing corner cases of small composite numbers and primes.

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.



```python
import math
import time
import tracemalloc

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

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

def is_carmichael(n):
    if n<2 :
        return False
    if all(n % i != 0 for i in range(2, int(math.sqrt(n)) + 1)):
        return False
    for a in range(2, n):
        if gcd(a, n) ==1:
           if pow(a, n-1, n) != 1:
               return False
    return True

end_time = time.time()
curr , max = tracemalloc.get_traced_memory()

print(gcd(7,5) ,is_carmichael(81))
print(f"Execution Time: {end_time - start_time} seconds")
print(f"Current Memory Usage for the above code is : {curr} bytes")
print(f"Max memory usage for the above code is {max} bytes")
```



```
1 False
Execution Time: 2.3126602172851562e-05 seconds
Current Memory Usage for the above code is : 1088 bytes
Max memory usage for the above code is 1088 bytes
```

Practical No: 36

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 $x = a^d \text{mod } n$ 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, $x \equiv n - 1 (\text{mod} n)$, 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:

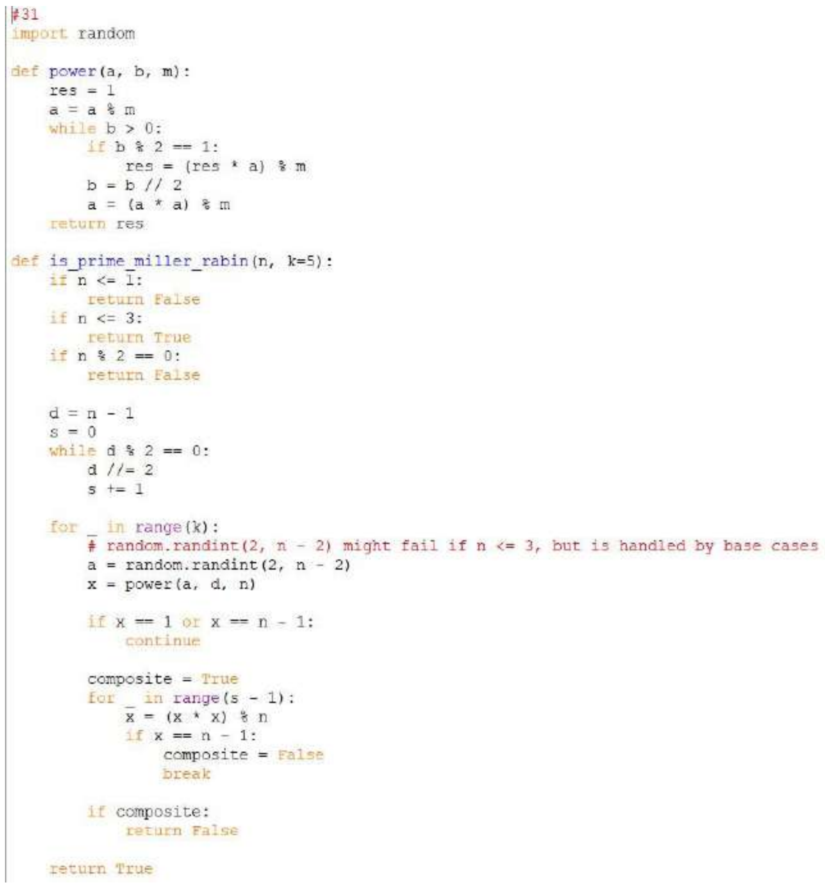
- Correctly identifies primes and composite numbers with a high degree of confidence.
- Suitable for very large numbers where deterministic tests are impractical.
- Reliable with a sufficient number of rounds k (commonly 5-10).

DIFFICULTY FACED BY STUDENT:

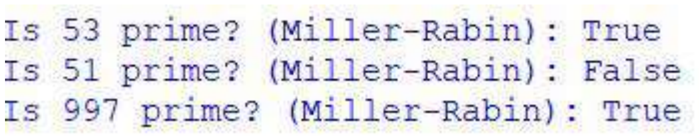
- Implementing modular exponentiation correctly and efficiently.
- Managing the probabilistic nature and interpreting results as "probably prime" rather than guaranteed prime.
- Choosing a sufficient number k of testing rounds balancing speed and accuracy.

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.



```python
#31
import random

def power(a, b, m):
    res = 1
    a = a % m
    while b > 0:
        if b % 2 == 1:
            res = (res * a) % m
        b = b // 2
        a = (a * a) % m
    return res

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

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

    for _ in range(k):
        # random.randint(2, n - 2) might fail if n <= 3, but is handled by base cases
        a = random.randint(2, n - 2)
        x = power(a, d, n)

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

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

        if composite:
            return False

    return True
```



```
Is 53 prime? (Miller-Rabin): True
Is 51 prime? (Miller-Rabin): False
Is 997 prime? (Miller-Rabin): True
```

Practical No: 37

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:

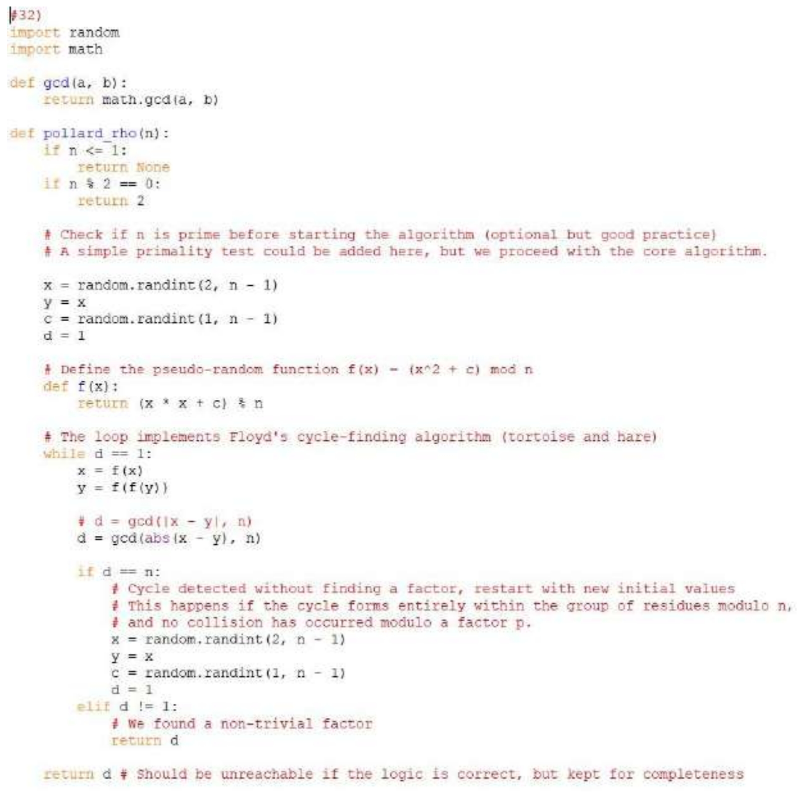
- The function returns a non-trivial factor of n when one exists.
- It works efficiently for numbers with small factors and often finds factors quickly.
- For prime n, it returns n itself or must be handled separately.

DIFFICULTY FACED BY STUDENT:

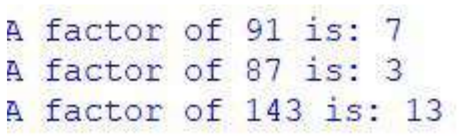
- Handling failure cases where the algorithm cycles without finding a factor (requires parameter tuning).
- Ensuring performance on large composite numbers with large prime factors.
- Avoiding infinite loops and ensuring termination.

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.



```python
#32)
import random
import math

def gcd(a, b):
    return math.gcd(a, b)

def pollard_rho(n):
    if n <= 1:
        return None
    if n % 2 == 0:
        return 2

    # Check if n is prime before starting the algorithm (optional but good practice)
    # A simple primality test could be added here, but we proceed with the core algorithm.

    x = random.randint(2, n - 1)
    y = x
    c = random.randint(1, n - 1)
    d = 1

    # Define the pseudo-random function f(x) - (x^2 + c) mod n
    def f(x):
        return (x * x + c) % n

    # The loop implements Floyd's cycle-finding algorithm (tortoise and hare)
    while d == 1:
        x = f(x)
        y = f(f(y))

        # d = gcd(|x - y|, n)
        d = gcd(abs(x - y), n)

        if d == n:
            # Cycle detected without finding a factor, restart with new initial values
            # This happens if the cycle forms entirely within the group of residues modulo n,
            # and no collision has occurred modulo a factor p.
            x = random.randint(2, n - 1)
            y = x
            c = random.randint(1, n - 1)
            d = 1
        elif d != 1:
            # We found a non-trivial factor
            return d

    return d # Should be unreachable if the logic is correct, but kept for completeness
```



```
A factor of 91 is: 7
A factor of 87 is: 3
A factor of 143 is: 13
```

Practical No: 38

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.

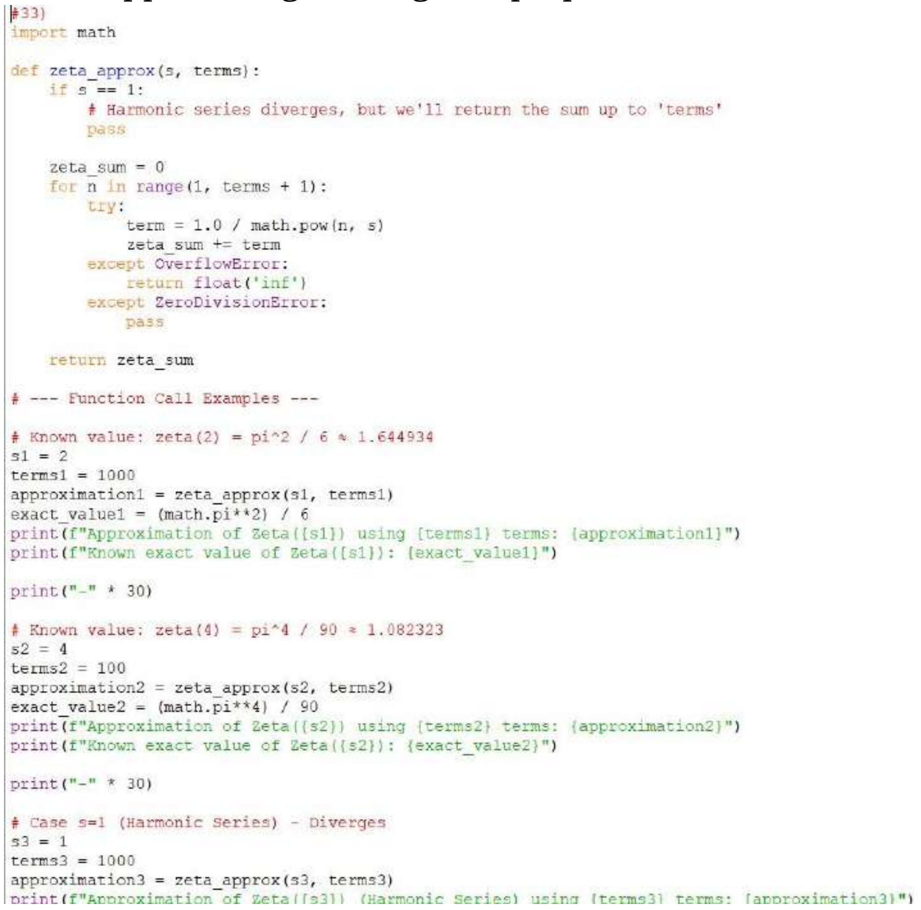
DIFFICULTY FACED BY STUDENT:

- 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.

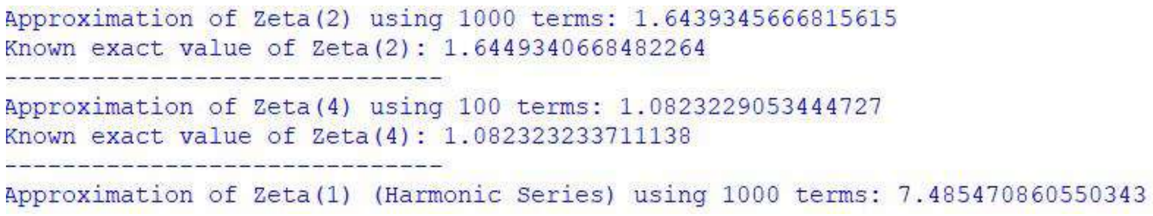
SKILLS ACHIEVED:

- Understanding infinite series approximations of special functions.

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



```
#33)
import math

def zeta_approx(s, terms):
    if s == 1:
        # Harmonic series diverges, but we'll return the sum up to 'terms'
        pass

    zeta_sum = 0
    for n in range(1, terms + 1):
        try:
            term = 1.0 / math.pow(n, s)
            zeta_sum += term
        except OverflowError:
            return float('inf')
        except ZeroDivisionError:
            pass

    return zeta_sum

# --- Function Call Examples ---

# Known value: zeta(2) = pi^2 / 6 ≈ 1.644934
s1 = 2
terms1 = 1000
approximation1 = zeta_approx(s1, terms1)
exact_value1 = (math.pi**2) / 6
print(f"Approximation of Zeta({s1}) using {terms1} terms: {approximation1}")
print(f"Known exact value of Zeta({s1}): {exact_value1}")

print("-" * 30)

# Known value: zeta(4) = pi^4 / 90 ≈ 1.082323
s2 = 4
terms2 = 100
approximation2 = zeta_approx(s2, terms2)
exact_value2 = (math.pi**4) / 90
print(f"Approximation of Zeta({s2}) using {terms2} terms: {approximation2}")
print(f"Known exact value of Zeta({s2}): {exact_value2}")

print("-" * 30)

# Case s=1 (Harmonic Series) - Diverges
s3 = 1
terms3 = 1000
approximation3 = zeta_approx(s3, terms3)
print(f"Approximation of Zeta({s3}) (Harmonic Series) using {terms3} terms: {approximation3}")
```



```
Approximation of Zeta(2) using 1000 terms: 1.6439345666815615
Known exact value of Zeta(2): 1.6449340668482264
------------------------------
Approximation of Zeta(4) using 100 terms: 1.0823229053444727
Known exact value of Zeta(4): 1.082323233711138
------------------------------
Approximation of Zeta(1) (Harmonic Series) using 1000 terms: 7.485470860550343
```

Practical No: 39

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.

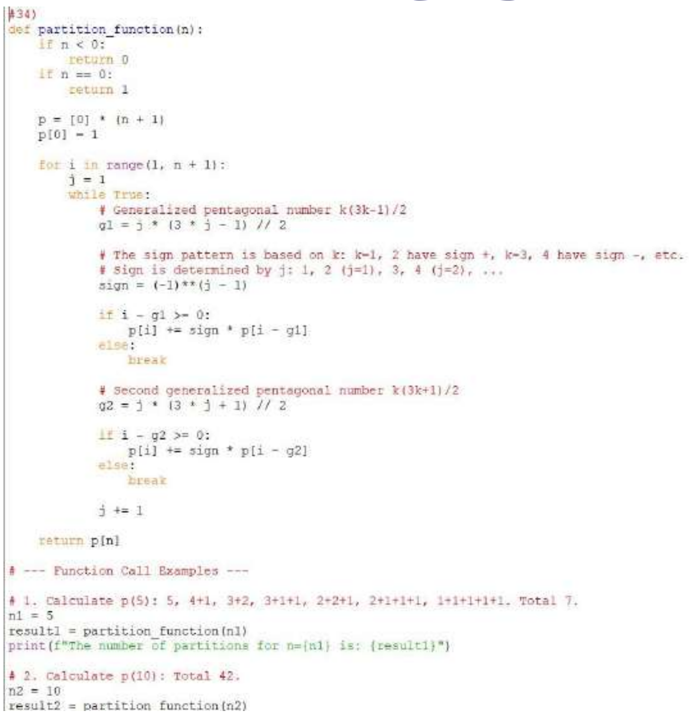
- For example, partition_function(5) returns 7, representing the partitions
- $5; 4 + 1; 3 + 2; 3 + 1 + 1; 2 + 2 + 1; 2 + 1 + 1 + 1; 1 + 1 + 1 + 1 + 1$
- Time complexity is $O(n^2)$, which is feasible for moderate sized n.

DIFFICULTY FACED BY STUDENT:

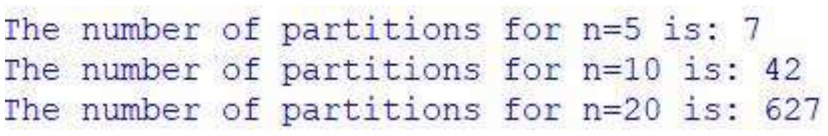
- Understanding the nontrivial recurrence or combinatorial logic behind partitions.
- Efficiently implementing DP to avoid exponential complexity.
- Memory management and performance for large n.

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
#34)
def partition_function(n):
    if n < 0:
        return 0
    if n == 0:
        return 1

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

    for i in range(1, n + 1):
        j = 1
        while True:
            # Generalized pentagonal number k(3k-1)/2
            g1 = j * (3 * j - 1) // 2

            # The sign pattern is based on k: k=1, 2 have sign +, k=3, 4 have sign -, etc.
            # Sign is determined by j: 1, 2 (j=1), 3, 4 (j=2), ...
            sign = (-1)**(j - 1)

            if i - g1 >= 0:
                p[i] += sign * p[i - g1]
            else:
                break

            # Second generalized pentagonal number k(3k+1)/2
            g2 = j * (3 * j + 1) // 2

            if i - g2 >= 0:
                p[i] += sign * p[i - g2]
            else:
                break

            j += 1

    return p[n]

# --- Function Call Examples ---

# 1. Calculate p(5): 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1. Total 7.
n1 = 5
result1 = partition_function(n1)
print(f"The number of partitions for n={n1} is: {result1}")

# 2. Calculate p(10): Total 42.
n2 = 10
result2 = partition function(n2)
```



```
The number of partitions for n=5 is: 7
The number of partitions for n=10 is: 42
The number of partitions for n=20 is: 627
```