Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE10875
Name of Student	: Suyash Jain
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	Write a function called euler_phi(n) that calculates Euler's Totient Function, (n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which gcd(n, k) = 1).	27/09/2025	
2	Write a function called mobius(n) that calculates the Möbius function, μ(n). The function is defined as: μ(n) = 1 if n is a square-free positive integer with an even number of prime factors. μ(n) = -1 if n is a square-free positive integer with an odd number of prime factors. μ(n) = 0 if n has a squared prime factor.	27/09/2025	
3	Write a function called divisor_sum(n) that calculates the sum of all positive divisors of n (including 1 and n itself). This is often denoted by o(n).	27/09/2025	
4	Write a function called prime_pi(n) that approximates the prime-counting function, π(n). This function returns the number of prime numbers less than or equal to n.	27/09/2025	
5	Write a function called legendre_symbol(a, p) that calculates the Legendre symbol (a/p), which is a useful function in quadratic reciprocity. It is defined for an odd prime p and an integer a not divisible by p as: (a/p) = 1 if a is a quadratic residue modulo p (i.e., there exists an integer x such that x2 a (mod p)). (a/p) = -1 if a is a quadratic non-residue modulo p.	27/09/2025	

	You can calculate it using Euler's criterion: (a/p) a^((p-1)/2) mod p.		
6	Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).	04/10/2025	
7	Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.	04/10/2025	
8	Write a function mean_of_digits(n) that returns the average of all digits in a number.	04/10/2025	
9	Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	04/10/2025	
10	Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.	04/10/2025	
11	Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.	02/11/2025	
12	Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.	02/11/2025	
13	Write a function is_automorphic(n) that checks if a number's square ends with the number itself.	02/11/2025	
14	Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.	02/11/2025	
15	Write a function prime_factors(n) that returns the list of prime factors of a number.	02/11/2025	

16	Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.	09/11/2025	
17	Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.	09/11/2025	
18	Write a function is_mersenne_prime(p) that checks if 2p – 1 is a prime number (given that p is prime).	09/11/2025	
19	Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.	09/11/2025	
20	Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.	09/11/2025	
21	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	13/11/2025	
22	Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).	13/11/2025	
23	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	13/11/2025	
24	Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.	13/11/2025	
25	Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.	13/11/2025	

26	Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.	13/11/2025	
27	Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ ri mod mi.	13/11/2025	
28	Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p has a solution.	13/11/2025	
29	Write a function order_mod(a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.	13/11/2025	
30	Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.	13/11/2025	
31	Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2,1).	16/11/2025	
32	Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as ab where a > 0 and b > 1.	16/11/2025	
33	Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.	16/11/2025	
34	Write a function Polygonal Numbers polygonal_number(s,n) that returns the n-th s-gonal number.	16/11/2025	

35	Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies an−1 ≡ 1 mod n for all a coprime to n.	16/11/2025	
36	Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.	16/11/2025	
37	Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.	16/11/2025	
38	Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.	16/11/2025	
39	Write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers.	16/11/2025	

Practical No: 1

Date: 27/09/2025

TITLE: Write a function called euler_phi(n) that calculates Euler's Totient Function, (n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which gcd(n, k) = 1).

AIM/OBJECTIVE(s):

1. To understand the concept of coprime numbers and Euler's Totient Function.
2. To implement an efficient algorithm in Python to compute φ(n).
3. To apply the concept of the greatest common divisor (GCD) in determining coprimality.
4. To verify the correctness of the implemented function for various values of *n*.

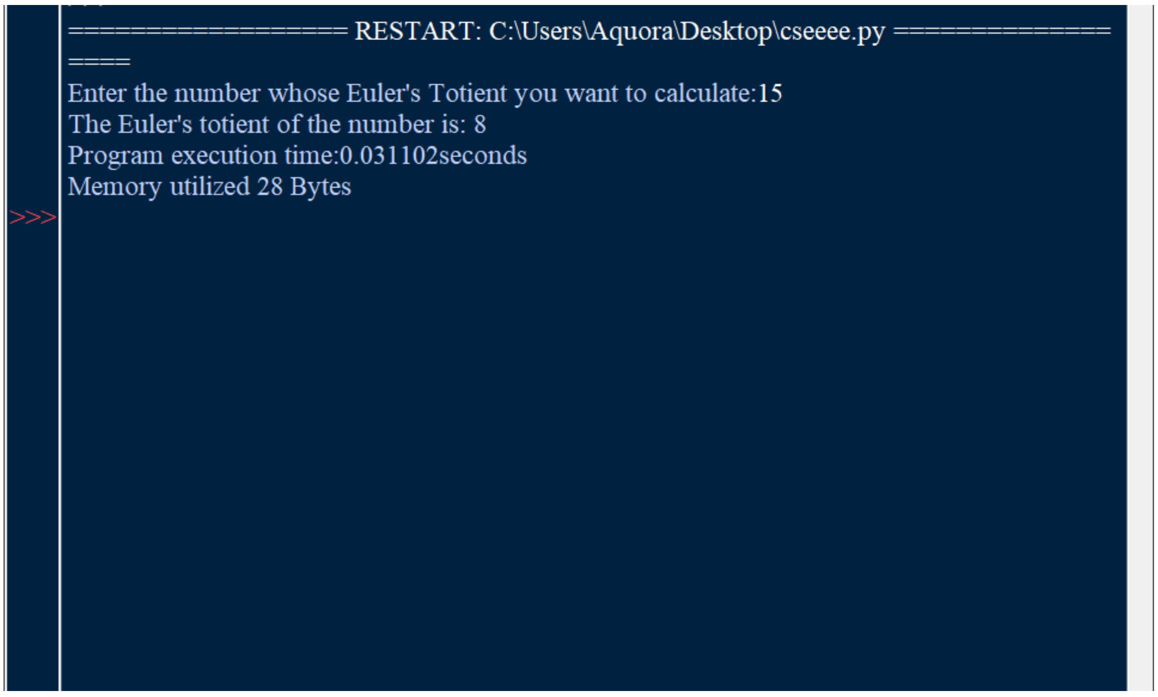
METHODOLOGY & TOOL USED:

```
import sys

import time

n=int(input("Enter the number whose Euler's Totient you want to calculate:"))

l1=list()

l2=list()

l3=list()

st=time.perf_counter()

def euler_phi(n):

    for i in range(1,n):

        l1.append(i)

    for j in l1:

        x=j
        y=n
        val=x%y
        while val!=0:
            val=x%y
            x,y=y,val
            l2.append(val)
        if l2[len(l2)-2]==1:
            l3.append(j)
    return len(l3)
a=euler_phi(n)
print("The Euler's totient of the number is:",a)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
print("Memory utilized",sys.getsizeof(euler_phi(n)),"Bytes")
```

BRIEF DESCRIPTION:

1. The function euler_phi(n) calculates Euler's Totient Function (φ(n)), which gives the count of integers less than or equal to *n* that are coprime to *n*.
2. It works by iterating through all numbers from 1 to *n* and checking which ones have a GCD of 1 with *n*.
3. Alternatively, it can use the prime factorization method for faster computation.
4. The function returns the total count of such numbers, which is the value of φ(n).

RESULTS ACHIEVED:



```
================== RESTART: C:\Users\Aquora\Desktop\cseeee.py ==============
====
Enter the number whose Euler's Totient you want to calculate:15
The Euler's totient of the number is: 8
Program execution time:0.031102seconds
Memory utilized 28 Bytes
>>>
```

DIFFICULTY FACED BY STUDENT: Employing the mathematical concept involved in calculation of Euler’s totient in Python.

SKILLS ACHIEVED:

1. Knowledge regarding python modules such as time and sys.
2. Concept of creating a list and appending values to it.
3. Concept of creating a user-defined function.
4. Concept of parameters and arguments in a function.
5. Traversing a list through for loop.
6. Mathematical concept relating to calculation of Euler’s totient of a number.

Practical No: 2

Date: 27/09/2025

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

μ(n) = 1 if n is a square-free positive integer with an even number of prime factors.

μ(n) = -1 if n is a square-free positive integer with an odd number of prime factors.

μ(n) = 0 if n has a squared prime factor.

AIM/OBJECTIVE(s):

1. To understand the concept of the Möbius function (μ(n)) and its significance in number theory.
2. To implement a Python function mobius(n) that calculates the value of μ(n) for a given positive integer *n*.
3. To identify whether a number is square-free and determine the count of its prime factors.
4. To apply the function in studying arithmetic functions, multiplicative functions, and other number-theoretic problems.

METHODOLOGY & TOOL USED:

```
import sys

import time

n=int(input("Enter the number whose mobius function you want to calculate:"))

l1=list()

l2=list()

c1=0

c2=0

st=time.perf_counter()
```

```
def mobius(n):
    x=2
    y=3
    while n%x==0:
        l1.append(x)
        n//=x
    while y*y<=n:
        while n%y==0:
            l1.append(y)
            n//=y
        y+=2
    if n>2:
        l1.append(n)
    for i in l1:
        if i not in l2:
            l2.append(i)
    if len(l2)!=len(l1):
        a=0
        return a
    else:
        if len(l2)%2==0:
            b=1
            return b
        else:
            b1=-1
            return b1
x=mobius(n)
et=time.perf_counter()
```

ent=et-st

print("The mobius value of the function is:",x)

print(f"Program execution time:{ent:4f}seconds")

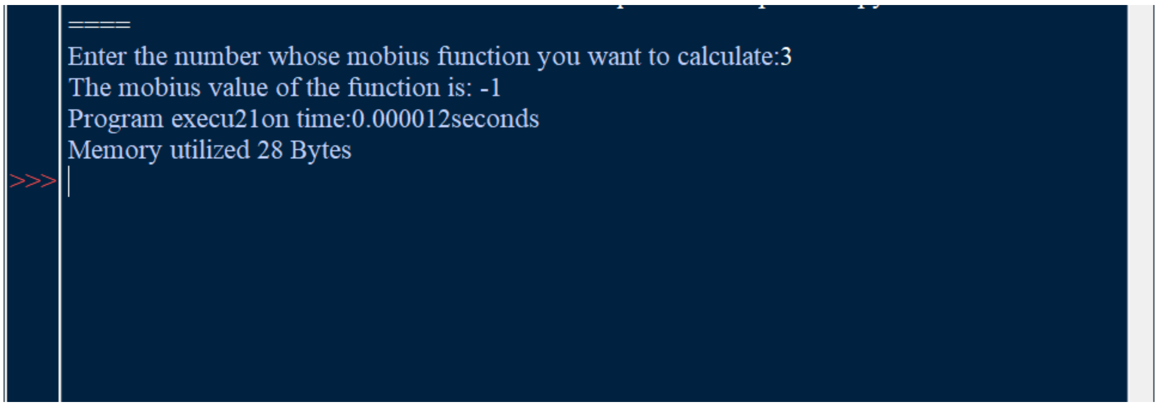
print("Memory utilized",sys.getsizeof(mobius(n)),"Bytes")

BRIEF DESCRIPTION:

The function mobius(n) calculates the Möbius function (μ(n)) for a given positive integer *n*.

1. It first checks whether *n* is square-free (i.e., not divisible by the square of any prime).
2. If *n* is square-free, it counts the number of distinct prime factors:
   - Returns 1 if the number of prime factors is even.
   - Returns -1 if the number of prime factors is odd.
3. If *n* has a squared prime factor, it returns 0.

RESULTS ACHIEVED:



```
====
Enter the number whose mobius function you want to calculate:3
The mobius value of the function is: -1
Program execu21on time:0.000012seconds
Memory utilized 28 Bytes
>>> |
```

DIFFICULTY FACED BY STUDENT: Identifying square free numbers, couting of the prime factors accurately and testing the efficiency of various codes against each other on the basis of space and time complexity.

SKILLS ACHIEVED:

1. Knowledge regarding python modules such as time and sys.
2. Concept of creating a list and appending values to it.
3. Concept of creating a user-defined function.
4. Prime number concept
5. Mathematical concepts pertaining to the calculation of the mobius function

Practical No: 3

Date: 27/09/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of divisors of a positive integer *n* and their sum.
2. To implement a Python function divisor_sum(n) that calculates the sum of all positive divisors of a given number, including 1 and *n* itself.
3. To apply the function in studying number-theoretic functions, perfect numbers, and other arithmetic properties.
4. To enhance problem-solving skills by efficiently iterating through divisors and handling larger numbers.

METHODOLOGY & TOOL USED:

```
import sys

import time

n=int(input("Enter the number whose factor sum you want to calculate:"))

st=time.perf_counter()

def divisor_sum(n):

    c=0

    for i in range(1,n+1):

        if n%i==0:

            c+=i

    return c

a=divisor_sum(n)
```

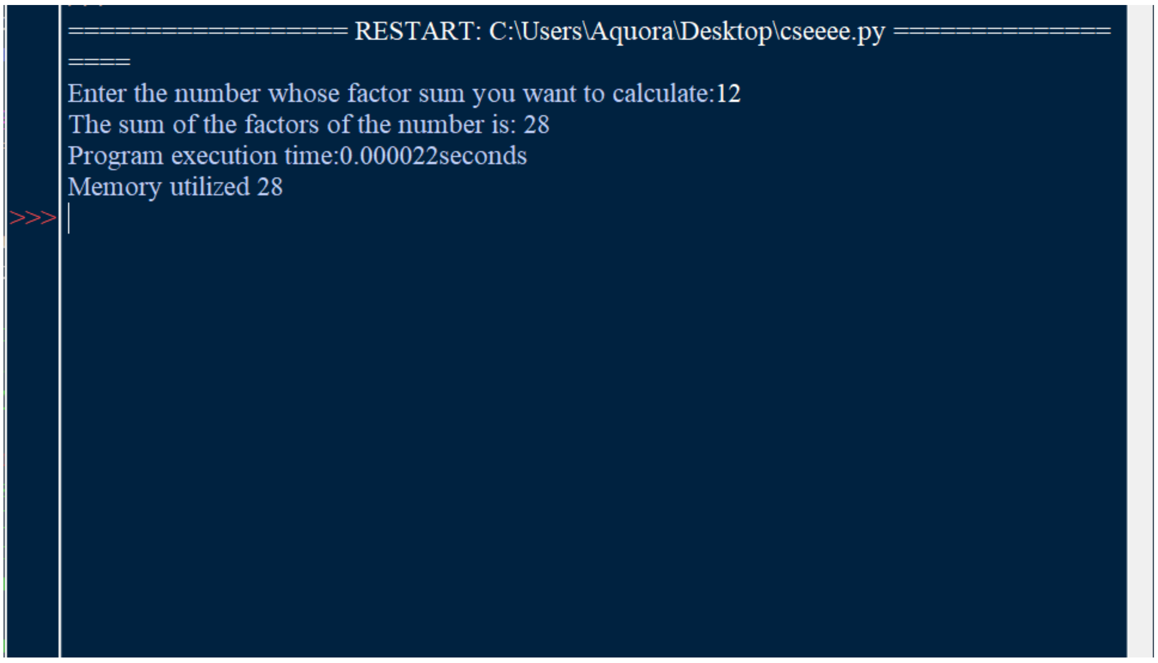
```
et=time.perf_counter()

ent=et-st

print("The sum of the factors of the number is:",a)

print(f"Program execution time:{ent:4f}seconds")

print("Memory utilized",sys.getsizeof(divisor_sum(n),"Bytes"))
```

BRIEF DESCRIPTION:

The function divisor_sum(n) calculates the sum of all **positive divisors** of a given positive integer *n*, including **1** and *n* itself.

1. The program iterates through all integers from 1 to *n* and checks which numbers divide *n* exactly (i.e., remainder = 0).
2. Each divisor is added to a running total to compute the sum.
3. The function finally returns this total, denoted as **σ(n)**, which is the sum of all divisors of *n*.

RESULTS ACHIEVED:



```
================== RESTART: C:\Users\Aquora\Desktop\cseeee.py ==============
=====
Enter the number whose factor sum you want to calculate:12
The sum of the factors of the number is: 28
Program execution time:0.000022seconds
Memory utilized 28
>>>
```

DIFFICULTY FACED BY STUDENT: Finding the divisors of the number, accidental double counting of factors while writing the code.

SKILLS ACHIEVED:

1. Knowledge regarding python modules such as time and sys.
2. Concept of creating a user-defined function.
3. Concept of finding factors of a number using looping constructs.

Practical No: 4

Date: 27/09/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of the prime-counting function π(n), which represents the number of prime numbers less than or equal to a given positive integer *n*.
2. To implement a Python function prime_pi(n) that calculates or approximates the value of π(n).
3. To apply the function in number theory, prime number analysis, and mathematical algorithms.
4. To enhance problem-solving skills by efficiently identifying prime numbers and counting them up to *n*.

METHODOLOGY & TOOL USED:

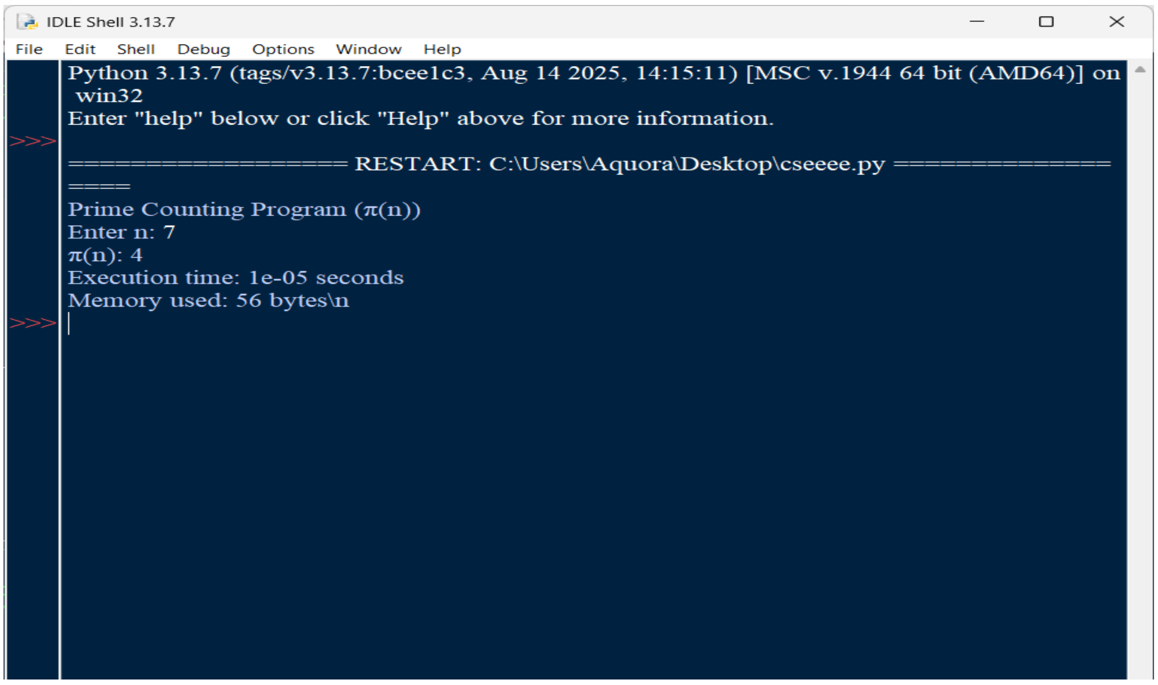
```
import time,sys

print("Prime Counting Program (π(n))")

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

def prime_pi(n):

    start=time.time()

    count=1

    for i in range(2,n+1):

        fact=True

        for j in range(2,n+1):

            if i%j==0:

                fact=False

                break
            if fact:
                count+=1
                break
    end = time.time()
    mem = sys.getsizeof(n) + sys.getsizeof(count)
    print("π(n):", count)
    print("Execution time:", round(end-start, 5), "seconds")
    print("Memory used:", mem, "bytes\\n")
prime_pi(n)
```

BRIEF DESCRIPTION:

The function prime_pi(n) calculates the prime-counting function (π(n)), which returns the number of prime numbers less than or equal to a given positive integer *n*.

1. The program iterates through all numbers from 2 to *n* and checks whether each number is prime.
2. A prime-checking subroutine (e.g., trial division) is used to determine if a number has no divisors other than 1 and itself.
3. Each prime number found is counted, and the total count is returned as π(n).

RESULTS ACHIEVED:



```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
 win32
Enter "help" below or click "Help" above for more information.
>>>
================== RESTART: C:\Users\Aquora\Desktop\cseeee.py ===============
=====
Prime Counting Program (π(n))
Enter n: 7
π(n): 4
Execution time: 1e-05 seconds
Memory used: 56 bytes\n
>>>
```

DIFFICULTY FACED BY STUDENT: Employing the mathematical concept involved successfully and counting of prime numbers.

SKILLS ACHIEVED:

1. Knowledge regarding python modules such as time and sys.
2. Concept of traversing a loop
3. Concept of creating a user-defined function.
4. Concept of prime numbers.

Practical No: 5

Date: 27/09/2025

TITLE: Write a function called legendre_symbol(a, p) that calculates the Legendre symbol (a/p), which is a useful function in quadratic reciprocity. It is defined for an odd prime p and an integer a not divisible by p as:

(a/p) = 1 if a is a quadratic residue modulo p (i.e., there exists an integer x such that x2 a (mod p)).

(a/p) = -1 if a is a quadratic non-residue modulo p.

You can calculate it using Euler's criterion: (a/p) a^((p-1)/2) mod p.

AIM/OBJECTIVE(s):

1. To understand the concept of the Legendre symbol (a/p), which determines whether an integer a is a quadratic residue modulo an odd prime p.
2. To implement a Python function legendre_symbol(a, p) that calculates the Legendre symbol using Euler's criterion:

$$(a/p) \equiv a^{(p-1)/2} (\mathrm{mod} p)$$

3. To apply the function in studying quadratic residues, quadratic reciprocity, and other concepts in number theory and cryptography.
4. To develop efficient computation methods for modular exponentiation and prime-based arithmetic in Python.

METHODOLOGY & TOOL USED:

```
import time,sys

def is_prime(n):

    if n<=1:

        return False
    for i in range(2, int(n**0.5)+1):
        if n%i==0:
            return False
    return True

def modular_pow(a, b, m):
    result=1
    base=a%m
    exp=b
    while exp>0:
        if exp%2==1:
            result=(result*base)%m


        exp=exp//2
    return result

def legendre_symbol(a, p):
    start_time=time.time()
    power=(p-1)//2
    value=modular_pow(a,power,p)
    if value==p-1:
        symbol=-1
    elif value==1:
        symbol=1
    else:
        symbol=0
    end_time=time.time()
```

```
memory_used=sys.getsizeof(a)+sys.getsizeof(p)+sys.getsizeof(power)+sys.g
etsizeof(value)+sys.getsizeof(symbol)
    print("\\n--- Results ---")
    print("Legendre Symbol (a/p):", symbol)
    print("Execution Time (seconds):", end_time - start_time)
    print("Total Memory Used (bytes):", memory_used)
    print("-------------------------\\n")


while True:
    print("Enter 0 as 'a' to exit the program.\\n")
    n=int(input("Enter value of n: "))
    if n==0:
        print("Exiting program.")
        break
    p=int(input("Enter an odd prime p: "))
    if p%2==0 or not is_prime(p):
        print("Error: p must be an odd prime number. Please try
again.\\n")
        continue
    is_prime(n)
    modular_pow(n,b,m)
    legendre_symbol(n,p)
```

BRIEF DESCRIPTION:

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

SKILLS ACHIEVED:

Practical No: 6

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of **factorial**, denoted as *n!*, which is the product of all positive integers less than or equal to a given non-negative integer *n*.
2. To implement a Python function factorial(n) that calculates the factorial of a non-negative integer.
3. To develop problem-solving and programming skills by implementing iterative or recursive approaches for factorial computation.
4. To apply the factorial function in combinatorics, probability, and mathematical analysis.

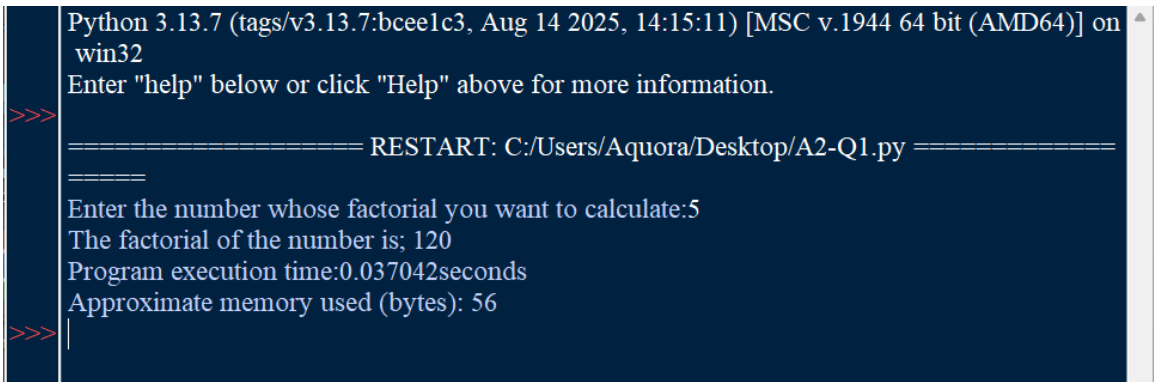
METHODOLOGY & TOOL USED:

```
import time,sys
n=int(input("Enter the number whose factorial you want to calculate:"))
st=time.perf_counter()
def factorial(n):
    if n<0:
        print("Negative integer, factorial calculation not possible!")
    else:
        fact=1
        while n!=0:
            fact*=n
            n-=1
    return fact
x=factorial(n)
print("The factorial of the number is;",x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(factorial(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. We create a counter termed as fact which has been assigened the value of 1.
2. Till the number is not equal to 0, we keep on multiplying it with fact thus forming a sequence as num*(num-1)*(num-2)….. till n=0 which translates to the factorial of a number in math.
3. If the number entered by the user is negative, factorial isn't possible.

RESULTS ACHIEVED:



```
    Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
     win32
    Enter "help" below or click "Help" above for more information.
>>>
    ================== RESTART: C:/Users/Aquora/Desktop/A2-Q1.py ============
    =====
    Enter the number whose factorial you want to calculate:5
    The factorial of the number is; 120
    Program execution time:0.037042seconds
    Approximate memory used (bytes): 56
>>> |
```

DIFFICULTY FACED BY STUDENT: Nil

SKILLS ACHIEVED:

1. Knowledge of Python modules such as time and sys.
2. Mathematical concept of factorial of a number.
3. Traversal and condition execution using while loop.
4. Testing various codes and finding out the most efficient one.

Practical No: 7

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of a **palindrome number**, which reads the same forwards and backwards.
2. To implement a Python function is_palindrome(n) that checks whether a given number is a palindrome.
3. To develop programming skills in number manipulation, string conversion, and algorithmic logic.
4. To apply the function in mathematical problem-solving, data validation, and programming exercises.

METHODOLOGY & TOOL USED:

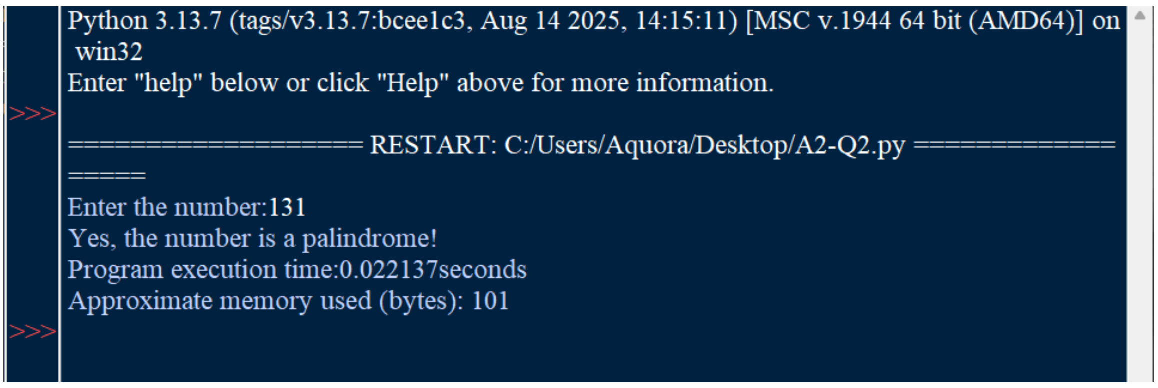
```
import time,sys

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

st=time.perf_counter()

def is_palindrome(n):

    st1=''

    for i in range(len(str(n))-1,-1,-1):

        st1+=str(n)[i]

    if st1==str(n):

        return "Yes, the number is a palindrome!"

    else:

        return "No, the number is not a palindrome!"

x=is_palindrome(n)

print(x)
```

```
et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(is_palindrome(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. Converting the integer number entered by the user into string.
2. We create an empty string to add characters to the same.
3. We start from te end of the number (now a string) adding each character one by one to the empty string.
4. The empty string now contains the number in backward reading format.
5. If the new string so created is equal to the number string, we can declare that the number entered by the user is a palindrome.

RESULTS ACHIEVED:



```
    Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
     win32
    Enter "help" below or click "Help" above for more information.
>>>
    =================== RESTART: C:/Users/Aquora/Desktop/A2-Q2.py =============
    ======
    Enter the number:131
    Yes, the number is a palindrome!
    Program execution time:0.022137seconds
    Approximate memory used (bytes): 101
>>>
```

DIFFICULTY FACED BY STUDENT: Employing the concept of string traversal and indexing.

SKILLS ACHIEVED:

1. The concept of palindrome.
2. Creating an empty string and adding characters to it.
3. Knowledge about modules of time and sys in Python.

Practical No: 8

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand how to work with the digits of a number individually.
2. To implement a Python function mean_of_digits(n) that calculates the average of all digits in a given number.
3. To develop programming skills in number manipulation, iteration, and arithmetic operations.

METHODOLOGY & TOOL USED:

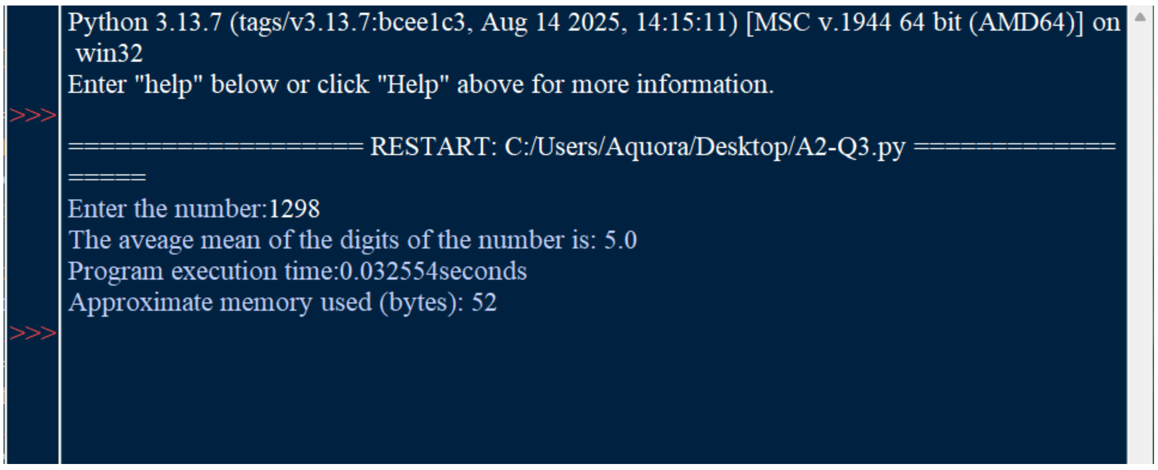
```
import time,sys

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

st=time.perf_counter()

def mean_of_digits(n):

    c=0

    for i in str(n):

        c+=int(i)

    return c/len(str(n))

x=mean_of_digits(n)

print("The aveage mean of the digits of the number is:",x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)
```

```
mem_count = sys.getsizeof(mean_of_digits(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. We convert the number entered by the user into string format so as to extract each digit of the same.
2. The each digit so extracted gets added to the counter variable c which has initially been assigned the value of 0.
3. Lastly we divide c now holding the sum of each digit of the number with the length of the number i.e the number of digits to obtain the average.

RESULTS ACHIEVED:



```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
win32
Enter "help" below or click "Help" above for more information.
>>>
=================== RESTART: C:/Users/Aquora/Desktop/A2-Q3.py =============
=====
Enter the number:1298
The aveage mean of the digits of the number is: 5.0
Program execution time:0.032554seconds
Approximate memory used (bytes): 52
>>>
```

DIFFICULTY FACED BY STUDENT: None as such.

SKILLS ACHIEVED:

1. Learning concepts of string and it's traversal
2. Using the inbuilt function len().
3. Type conversion.
4. Knowledge of Python modules; time and sys.
5. Mathematical concept of averages.

Practical No: 9

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To employ the knowledge of Python, it's vast collection of rich libraries and functions to create a function that satisfies the requirements.
2. To educate the students about the mathematical logic behind a digital root and write a code on the same.

METHODOLOGY & TOOL USED:

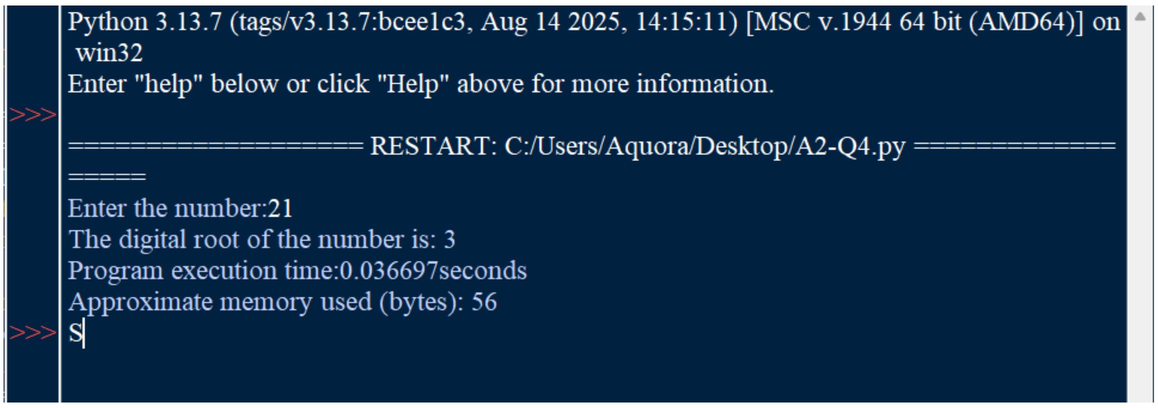
```
import time,sys

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

st=time.perf_counter()

def digital_root(n):

    if n==0:

        return 0

    return 1+(n-1)%9

x=digital_root(n)

print("The digital root of the number is:",x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(digital_root(n))

mem_total = mem_n + mem_count
```

```
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The mathematical concept behind the expression 1+(n-1)%9
2. If in case a number is divisible by 9 then, the digital root return with the value of 9 following the above expression.
3. The modulo operator evaluates the remainder of the division operation in Python.

RESULTS ACHIEVED:



```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
 win32
Enter "help" below or click "Help" above for more information.
>>>
=================== RESTART: C:/Users/Aquora/Desktop/A2-Q4.py =============
=====
Enter the number:21
The digital root of the number is: 3
Program execution time:0.036697seconds
Approximate memory used (bytes): 56
>>> S
```

DIFFICULTY FACED BY STUDENT: Nil

SKILLS ACHIEVED:

1. Mathematical concept of digital root.
2. Passing an argument/parameter to a function.
3. Knowledge of time and sys as python modules.

Practical No: 10

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of abundant numbers, which are numbers whose sum of proper divisors exceeds the number itself.
2. To implement a Python function is_abundant(n) that checks whether a given number *n* is abundant.
3. To develop programming skills in divisor calculation, iteration, and conditional logic.
4. To apply the function in number theory, particularly in the study of perfect, deficient, and abundant numbers.

METHODOLOGY & TOOL USED:

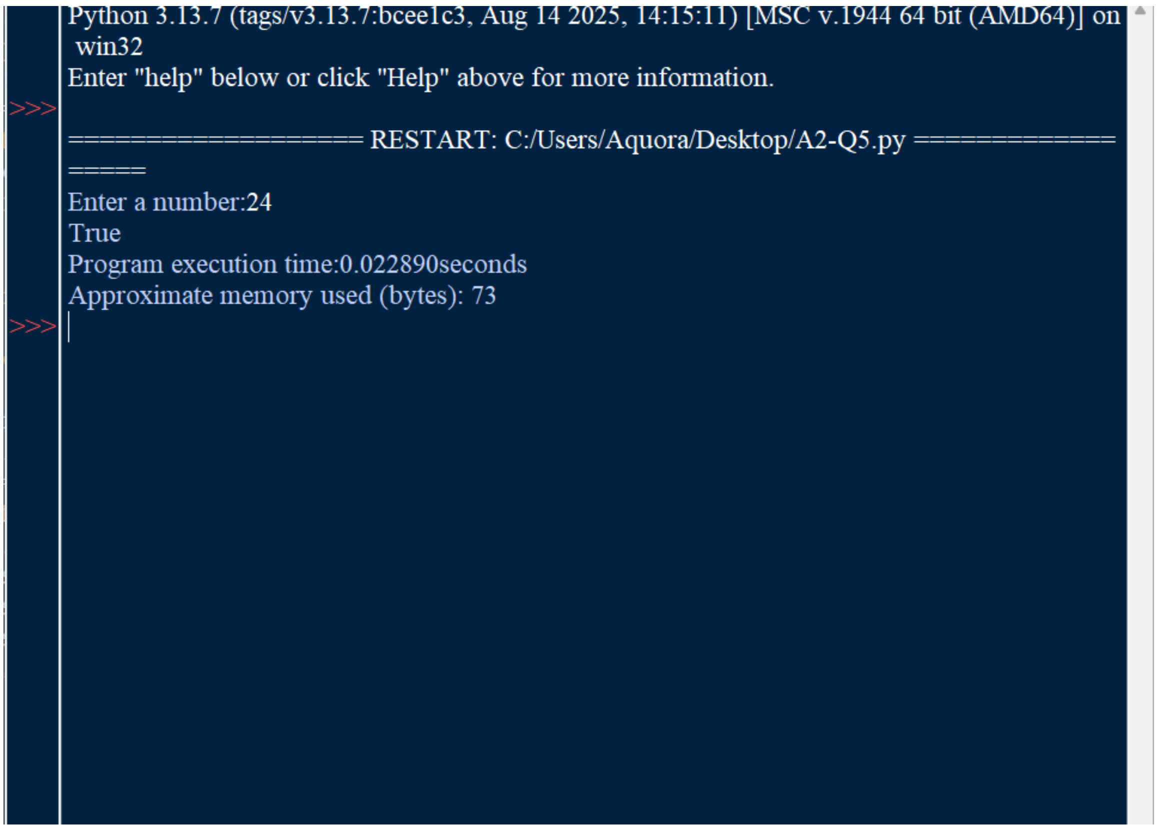
```
import time,sys

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

st=time.perf_counter()

def is_abundant(n):

    c=0

    for i in range(1,n):

        if n%i==0:

            c+=i

    if c>n:

        return "True"

    else:

        return "False"

x=is_abundant(n)
```

```
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(is_abundant(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The time and sys modules are helpful in the calculation of the execution time of the programme and the memory utilization respectively.
2. c is defined as a counter with the equivalent value of 0 which gets incremented each time a divisor of n is found
3. The for loop is employed to traverse numbers till n in order to check for divisors of n.
4. The return statements are helpful in producing the output as True or False.

RESULTS ACHIEVED:



```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)] on
 win32
Enter "help" below or click "Help" above for more information.
>>>
=================== RESTART: C:/Users/Aquora/Desktop/A2-Q5.py =============
======
Enter a number:24
True
Program execution time:0.022890seconds
Approximate memory used (bytes): 73
>>> |
```

DIFFICULTY FACED BY STUDENT: Minor problems encountered on implementing the logic and getting to an efficient code.

SKILLS ACHIEVED:

7. Knowledge regarding python modules such as time and sys.
8. Creating variables and assigning values.
9. Traversing a loop.
10. Returning a value in a fucntion

Practical No: 11

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of deficient numbers, which are numbers whose sum of proper divisors does not exceed the number itself.
2. To implement a Python function is_deficient(n) that checks whether a given number *n* is deficient.
3. To develop programming skills in divisor calculation, iteration, and conditional logic.
4. To apply the function in number theory, particularly in the study of perfect, deficient, and deficient numbers.

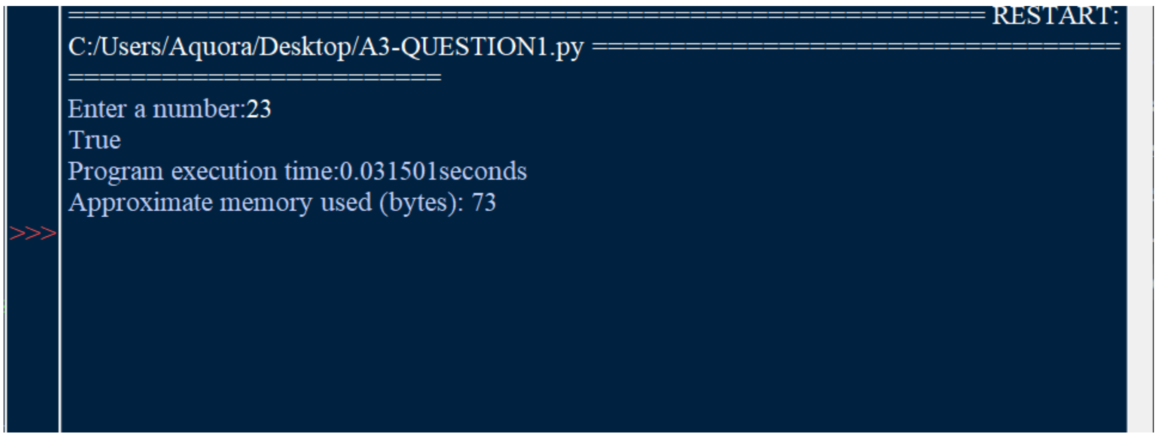
METHODOLOGY & TOOL USED:

```
import time,sys

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

st=time.perf_counter()

def is_deficient(n):

    c=0

    if n<0:

        return False

    for i in range(1,n):

        if n%i==0:

            c+=i

    if c<n:

        return True

    else:
        return False
x=is_deficient(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(is_deficientt(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

5. The time and sys modules are helpful in the calculation of the execution time of the programme and the memory utilization respectively.
6. c is defined as a counter with the equivalent value of 0 which gets incremented each time a divisor of n is found
7. The for loop is employed to traverse numbers till n in order to check for divisors of n.
8. The return statements are helpful in producing the output as True or False.

RESULTS ACHIEVED:



```
=============================================================== RESTART:
C:/Users/Aquora/Desktop/A3-QUESTION1.py ==================================
==========================
Enter a number:23
True
Program execution time:0.031501seconds
Approximate memory used (bytes): 73
>>>
```

DIFFICULTY FACED BY STUDENT: Minor problems encountered on implementing the logic and getting to an efficient code.

SKILLS ACHIEVED:

1. Concept of proper divisors.
2. Mathematical classification techniques.
3. Looping constructs and condition evaluation.
4. Programme efficiency awareness.
5. Boolean returns.

Practical No: 12

Date: 04/10/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of digit sum and its computation.
2. To practice type conversion (integer ↔ string) in Python.
3. To learn how to use loops or comprehensions to process each digit.
4. To develop logical reasoning by checking divisibility conditions.
5. To gain familiarity with returning Boolean values (True/False).

METHODOLOGY & TOOL USED:

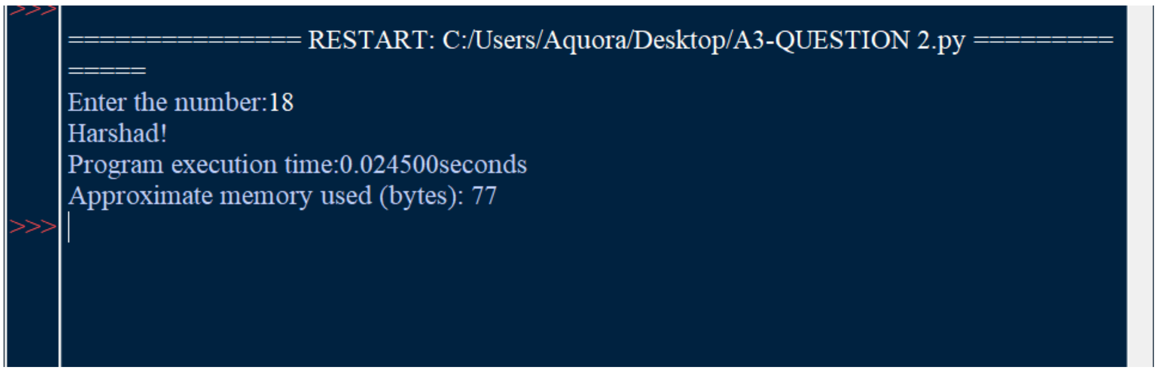
```
import time,sys
n=int(input("Enter the number:"))
st=time.perf_counter()
def is_harshad(n):
    c=0
    if n<0:
        return "Not Harshad!"
    for i in str(n):
        c+=int(i)
    if n%c==0:
        return "Harshad!"
    else:
        return "Not Harshad!"
```

```
x=is_harshad(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(is_harshad(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. A Harshad number (or Niven number) is a positive integer that is divisible by the sum of its digits.
2. In this program, the user inputs a number, and the function is_harshad(n) checks whether the number satisfies this condition.
3. The program works by finding the sum of all digits of the number and then checking if the number is divisible by that sum.
4. If it is, the function returns True; otherwise, it returns False.

RESULTS ACHIEVED:



```
>>> 
=============== RESTART: C:/Users/Aquora/Desktop/A3-QUESTION 2.py ==========
=====
Enter the number:18
Harshad!
Program execution time:0.024500seconds
Approximate memory used (bytes): 77
>>> |
```

DIFFICULTY FACED BY STUDENT: Slight problem encountered while validating the logic behind the code.

SKILLS ACHIEVED:

1. Mathematical reasoning:
   Understanding divisibility and number properties.
2. String manipulation and iteration:
   Converting a number to string to easily access digits.
3. Python list comprehensions:
   Writing concise code using sum(int(d) for d in str(n)).
4. Logical thinking:
   Structuring a program to test conditions and return appropriate results.
5. Error handling and edge-case thinking:
   Learning to handle inputs like 0 or negative numbers safely.

VIT BHOPAL

Practical No: 13

Date: 04/10/2025

TITLE: Write a function is_automorphic(n) that checks if a number's square ends with the number itself.

AIM/OBJECTIVE(s):

1. To understand the concept of Automorphic numbers.
2. To learn how to compute a number’s square in Python.
3. To apply string manipulation and use functions like .endswith().
4. To strengthen logical thinking and conditional testing.
5. To return Boolean results based on number properties.

METHODOLOGY & TOOL USED:

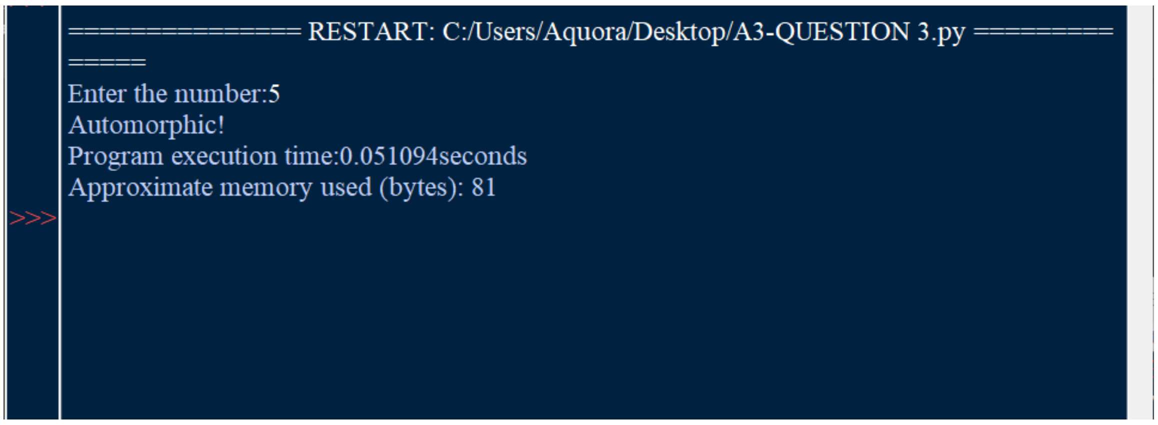
```
import time,sys

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

st=time.perf_counter()

def is_automorphic(n):

    x=n**2

    if n<0:

        return "Not Automorphic!"

    if str(x).endswith(str(n)):

        return "Automorphic!"

    else:

        return "Not Automorphic!"

x=is_automorphic(n)

print(x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")
```

```
mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(is_automorphic(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. An Automorphic number is a number whose square ends with the same digits as the number itself.
2. In this program, the function is_automorphic(n) checks this property and returns “Automorphic!” if the number is automorphic, else “Not Automorphic!”.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A3-QUESTION 3.py =========
=====
Enter the number:5
Automorphic!
Program execution time:0.051094seconds
Approximate memory used (bytes): 81
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Confusion in comparing the last digits of the square with the original number.
2. Difficulty in handling the comparison between numbers and strings (type mismatch).
3. Forgetting to convert both the square and the number to strings before comparison.
4. Misunderstanding the Automorphic condition (checking prefix instead of suffix).

SKILLS ACHIEVED:

1. Logical and analytical thinking through number pattern recognition.
2. Python string operations — especially .endswith() method.
3. Use of exponentiation (**) operator.
4. Understanding of type conversion between integers and strings.
5. Writing clean, efficient, and readable Python code with meaningful conditions.

Practical No: 14

Date: 04/10/2025

TITLE: Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.

AIM/OBJECTIVE(s):

1. To understand the concept of Pronic numbers and their mathematical pattern.
2. To practice loops and conditional statements in Python.
3. To apply logical reasoning to identify number relationships.
4. To improve skills in iteration and comparison.
5. To produce a Boolean output (True/False) based on a mathematical condition.

METHODOLOGY & TOOL USED:

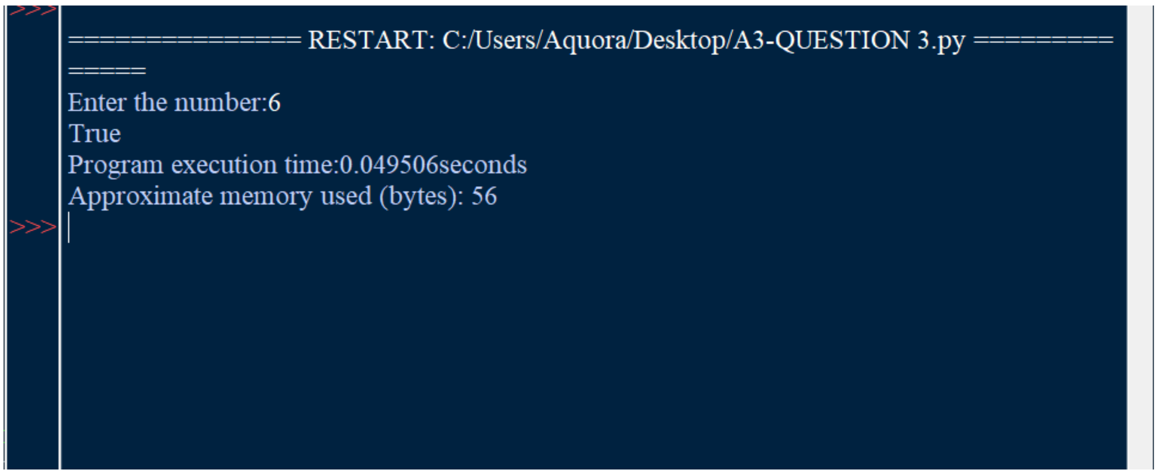
```
import time,sys
n=int(input("Enter the number:"))
st=time.perf_counter()
def is_pronic(n):
    if n < 0:
        return False
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False
x=is_pronic(n)
```

```
print(x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(is_pronic(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. A number n is pronic if there exists an integer k such that n = k × (k + 1)
2. The function is_pronic(n) checks whether a given number satisfies this property.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:/Users/Aquora/Desktop/A3-QUESTION 3.py =========
=====
Enter the number:6
True
Program execution time:0.049506seconds
Approximate memory used (bytes): 56
>>> |
```

DIFFICULTY FACED BY STUDENT:

1. Identifying how to generate consecutive integers systematically.
2. Confusion between multiplication vs addition when checking consecutive pairs.

SKILLS ACHIEVED:

1. Analytical thinking and pattern recognition in number theory.
2. Loop control and termination logic using while loops.
3. Conditional checking with logical operators in Python.
4. Efficient use of iteration to test mathematical relationships.
5. Ability to write clear and concise Python functions with Boolean returns.

Practical No: 15

Date: 04/10/2025

TITLE: Write a function prime_factors(n) that returns the list of prime factors of a number.

AIM/OBJECTIVE(s):

1. To understand the concept of prime factorization.
2. To learn how to check divisibility and primality of numbers.
3. To implement loops and conditional statements effectively.
4. To use integer division and modulus operators in Python.
5. To return results in a structured list format.

METHODOLOGY & TOOL USED:

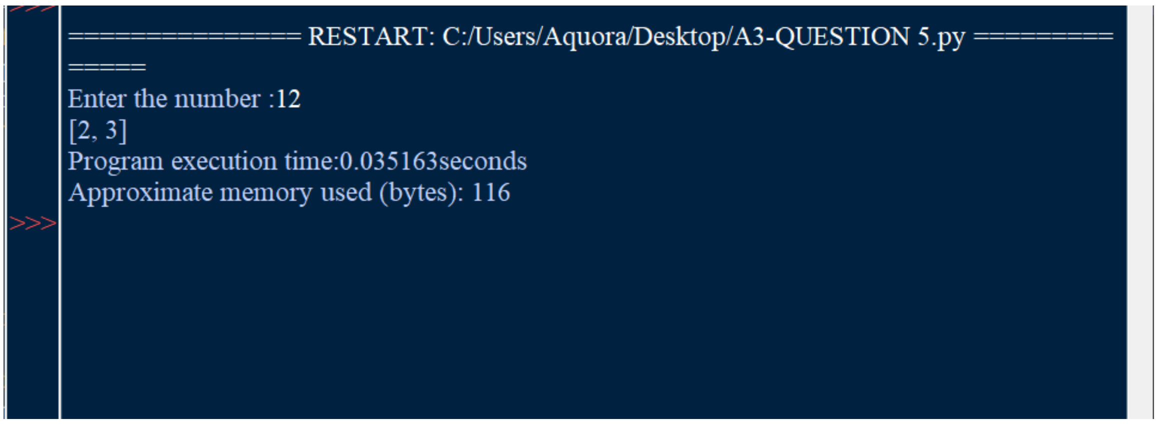
```
import time,sys
n=int(input("Enter the number :"))
st=time.perf_counter()
def prime_factors(n):
    prime = []
    i = 2
    while i * i <= n:
        if n % i == 0:
            prime.append(i)
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        prime.append(n)
    return prime
x=prime_factors(n)
```

```
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(prime_factors(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The prime factors of a number are the prime numbers that divide it exactly, without leaving any remainder.
2. In this program, the function prime_factors(n) finds and returns a list of all prime factors of a given number.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A3-QUESTION 5.py =========
=====
Enter the number :12
[2, 3]
Program execution time:0.035163seconds
Approximate memory used (bytes): 116
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Handling cases like 1, 0, or negative numbers appropriately.
2. Forgetting to remove duplicate primes or reduce the number after division.
3. Logical errors in loop termination (not stopping when i * i > n).
4. Mixing up integer division (//) and normal division (/), leading to float results.

SKILLS ACHIEVED:

1. Understanding prime numbers and factorization principles.
2. Using while loops efficiently for iterative division.
3. Applying conditional logic for checking divisibility.
4. Practicing list operations and appending results dynamically.
5. Developing efficient and logical problem-solving strategies in Python.

Practical No: 16

Date: 09/11/2025

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

AIM/OBJECTIVE(s):

1. To understand the concept of prime factorization.
2. To learn how to check divisibility and primality of numbers.
3. To implement loops and conditional statements effectively.
4. To use integer division and modulus operators in Python

METHODOLOGY & TOOL USED:

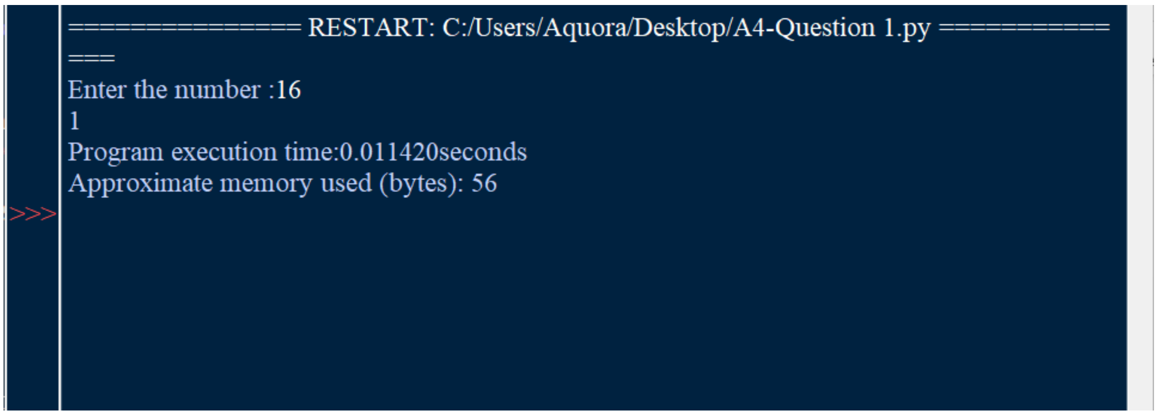
```
import time,sys

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

```
st=time.perf_counter()
def count_distinct_prime_factors(n):
    c=0
    i=2
    while i * i<=n:
        if n % i==0:
            c += 1
            while n % i==0:
                n //= i
        i += 1
    if n > 1:
        c+=1
    return c
x=count_distinct_prime_factors(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(count_distinct_prime_factors(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The function determines how many distinct prime numbers divide the input number n.
2. It does this by repeatedly dividing n by potential prime factors starting from 2, counting each distinct factor only once.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A4-Question 1.py ===========
===
Enter the number :16
1
Program execution time:0.011420seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Ensuring duplicate prime factors were not counted multiple times (e.g., counting 2 only once in $2^3 \times 3$).
2. Handling large numbers efficiently without generating all primes up to n.
3. Avoiding infinite loops or incorrect termination conditions when reducing n progressively.

SKILLS ACHIEVED:

1. Understanding of prime factorization and trial division.
2. Enhanced logical thinking for loop control and conditional checks.
3. Improved ability to optimize code for mathematical computations.
4. Strengthened grasp of number theory concepts and Python programming fundamentals.

Practical No: 17

Date: 09/11/2025

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

AIM/OBJECTIVE(s):

1. To create a Python function is_prime_power(n) that determines whether a given integer can be expressed as p^k.
2. To ensure that $p$is a prime number and $k \geq 1$.
3. To apply mathematical and logical reasoning for checking prime powers efficiently.
4. To enhance understanding of number theory and Python programming concepts through implementation.

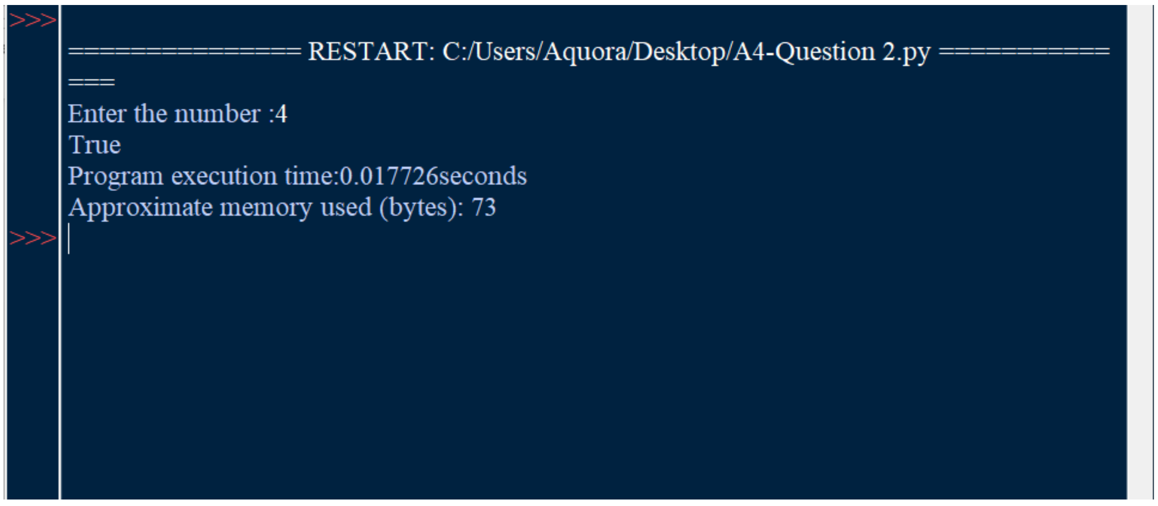
METHODOLOGY & TOOL USED:

```
import time,sys

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

st=time.perf_counter()

def is_prime_power(n):

    a=0

    if n < 2:

        return False

    for i in range(2, int(n ** 0.5) + 1):

        if n % i == 0:

            a+=1

    if n < 2:

        return False

    for k in range(1, int(n ** 0.5) + 2):

        p = round(n ** (1 / k))
        if p ** k==n and a==0:
            return "False"
    return "True"
x=is_prime_power(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(is_prime_power(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The function determines if a number is a prime power — that is, whether it can be written as a single prime raised to some positive integer exponent.

RESULTS ACHIEVED:



```
>>> 
================ RESTART: C:/Users/Aquora/Desktop/A4-Question 2.py ===========
===
Enter the number :4
True
Program execution time:0.017726seconds
Approximate memory used (bytes): 73
>>> 
```

DIFFICULTY FACED BY STUDENT:

1. Handling floating-point precision when computing roots for different values of k.
2. Ensuring that only one prime factor defines the number, avoiding composite bases.
3. Managing edge cases like n = 1 or small numbers (since $1 = p^0$ is not valid).

SKILLS ACHIEVED:

1. Deepened understanding of prime numbers and exponentiation relationships.
2. Enhanced ability to use mathematical reasoning to verify number properties.
3. Improved skills in Python functions, loops, and mathematical computation.
4. Strengthened knowledge of algorithmic efficiency and precision handling.

Practical No: 18

Date: 09/11/2025

TITLE: Write a function is_mersenne_prime(p) that checks if 2p –1 is a prime number (given that p is prime).

AIM/OBJECTIVE(s):

1. To create a single-function program is_mersenne_prime(p) that checks whether $2^p - 1$is a prime number, given that $p$is prime.
2. To understand and implement the concept of Mersenne primes in Python.
3. To improve mathematical logic and coding efficiency in prime number computations.

METHODOLOGY & TOOL USED:

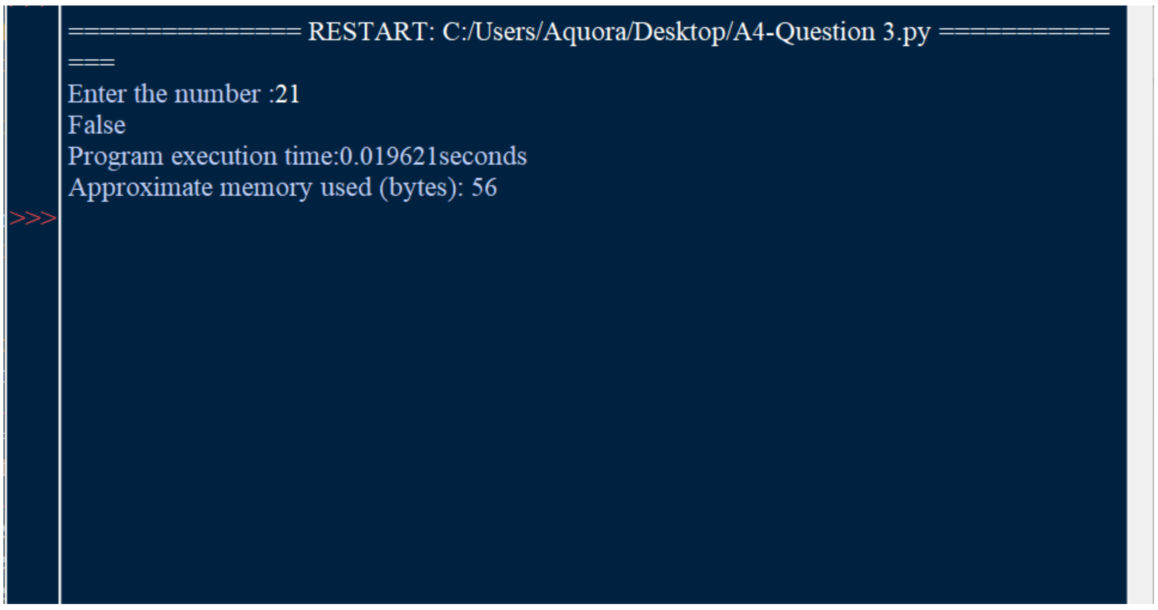
```
import time,sys

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

st=time.perf_counter()

def is_mersenne_prime(p):

    if p < 2:

        return False

    for i in range(2, int(p ** 0.5) + 1):

        if p % i == 0:

            return False

    mersenne = 2 ** p - 1

    for i in range(2, int(mersenne ** 0.5) + 1):

        if mersenne % i == 0:

            return False

    return True
```

```
x=is_mersenne_prime(n)

print(x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(is_mersenne_prime(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The function takes an integer p as input and determines if it forms a Mersenne prime.
A Mersenne prime is of the form $M_p = 2^p - 1$, where both $p$and $M_p$are prime.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A4-Question 3.py ===========
===
Enter the number :21
False
Program execution time:0.019621seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Avoiding the use of multiple helper functions while keeping the logic clear.
2. Handling large exponentiation for higher values of p without performance issues.
3. Ensuring both checks (for p and 2^p - 1) are implemented correctly in a single loop structure.

SKILLS ACHIEVED:

1. Stronger understanding of prime checking algorithms.
2. Knowledge of Mersenne numbers and their mathematical significance.
3. Enhanced ability to structure logic efficiently within a single function.
4. Improved problem-solving and computational thinking in Python.

Practical No: 19

Date: 09/11/2025

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

AIM/OBJECTIVE(s):

1. To develop a Python function twin_primes(limit) that finds all twin prime pairs up to a given number.
2. To enhance understanding of prime numbers and their relationships.
3. To improve logical reasoning and loop-based programming skills.
4. To efficiently generate and verify twin primes without using extra helper functions.

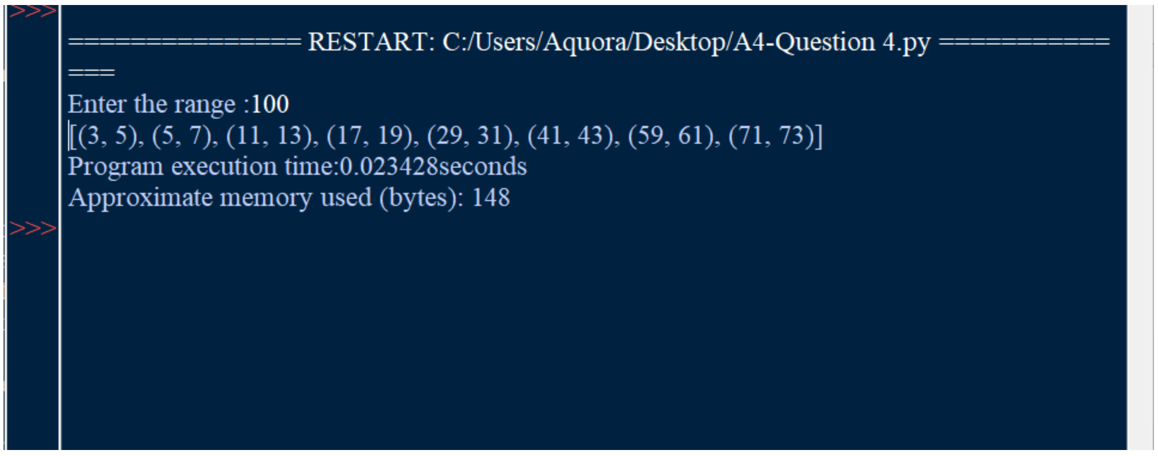
METHODOLOGY & TOOL USED:

```
import time,sys

n=int(input("Enter the range :"))

st=time.perf_counter()

def twin_primes(n):

    t=[]

    for i in range(2, n- 1):

        prime1=True

        for j in range(2, int(i ** 0.5) + 1):

            if i % j == 0:

                prime1=False

                break

        prime2= True

        for j in range(2, int((i + 2) ** 0.5) + 1):

            if (i + 2) % j == 0:
                prime2=False
                break
        if prime1 and prime2 and (i + 2) <=n:
            t.append((i, i + 2))
    return t
x=twin_primes(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(twin_primes(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. Twin primes are pairs of prime numbers that differ by exactly 2. For example, (3, 5), (5, 7), (11, 13), and (17, 19) are all twin prime pairs.
2. The function checks all numbers up to the specified limit, identifies which numbers are prime, and then finds pairs of primes that have a difference of 2.

RESULTS ACHIEVED:



```
>>>
    =============== RESTART: C:/Users/Aquora/Desktop/A4-Question 4.py ===========
    ===
    Enter the range :100
    [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
    Program execution time:0.023428seconds
    Approximate memory used (bytes): 148
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Ensuring the prime check function was efficient enough for larger limits.
2. Avoiding repetitive calculations when checking consecutive primes.
3. Handling edge cases like small limits (e.g., limit < 3, where no twin primes exist).
4. Keeping the code concise while maintaining readability.

SKILLS ACHIEVED:

1. Deep understanding of prime number properties and twin prime concepts.
2. Improved ability to write nested functions and iterate logically through number ranges.
3. Enhanced problem-solving through efficient mathematical checks.
4. Better grasp of Python list operations, loops, and function structuring.

Practical No: 20

Date: 09/11/2025

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

AIM/OBJECTIVE(s):

1. To write a Python function count_divisors(n) that calculates how many positive divisors a given number has.
2. To understand and apply the concept of factors and divisibility in mathematics.
3. To enhance problem-solving and loop-based logic-building skills.
4. To implement the program using a single function without helper functions.

METHODOLOGY & TOOL USED:

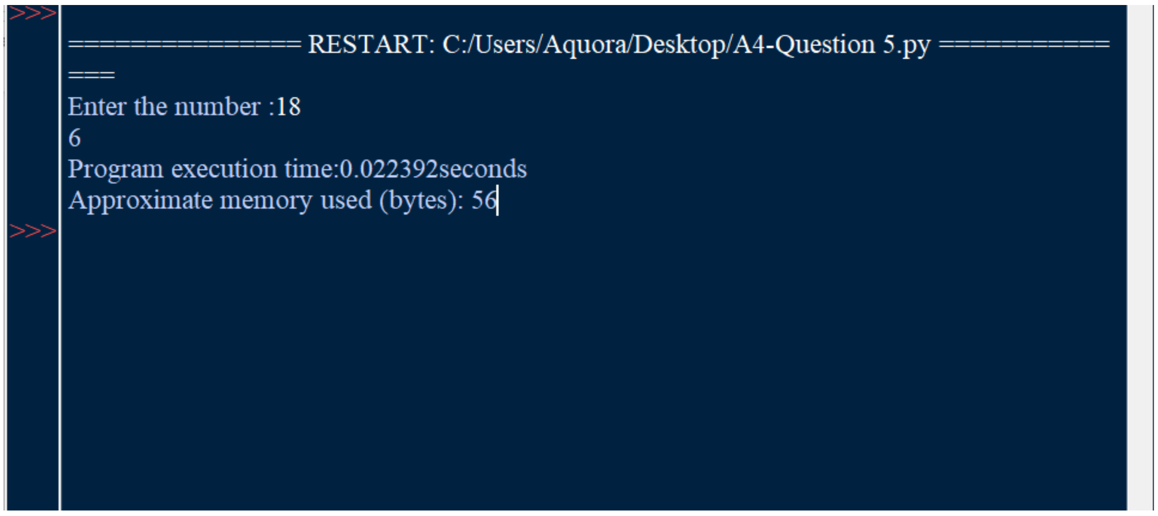
```
import time,sys

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

st=time.perf_counter()

def count_divisors(n):

    c=0

    i=1

    while i * i<=n:

        if n % i==0:

            if i * i==n:

                c+=1

            else:

                c+=2

        i+=1

    return c
```

```
x=count_divisors(n)

print(x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(count_divisors(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

Every positive integer has certain numbers that divide it completely, known as its divisors or factors.
The function counts how many such positive integers divide n exactly (i.e., with remainder 0).

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:/Users/Aquora/Desktop/A4-Question 5.py ===========
===
Enter the number :18
6
Program execution time:0.022392seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Ensuring no double counting of divisors for perfect squares (e.g., 36 → 6 counted once).
2. Optimizing the loop to run only up to √n for better performance.
3. Avoiding logic errors when differentiating between divisor pairs.
4. Handling small values like n = 1 correctly.

SKILLS ACHIEVED:

1. Strengthened understanding of divisibility and factors in number theory.
2. Improved proficiency in loop design and conditional logic.
3. Learned to optimize mathematical computations using square root limits.
4. Enhanced ability to write concise and efficient single-function programs in Python.

Practical No: 21

Date: 13/11/2025

TITLE: Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

AIM/OBJECTIVE(s):

1. Identify all integers that divide the given number without leaving a remainder.
2. Accumulate these divisors to obtain their total sum.
3. Handle edge cases (such as n = 1, which has no proper divisors).
4. Understand the relevance of such a function in determining special numbers like perfect numbers, amicable pairs, and deficient numbers, which are based on the concept of aliquot sums.

METHODOLOGY & TOOL USED:

```
import time,sys
n=int(input("Enter the number:"))
st=time.perf_counter()
def aliquot_sum(n):
    c=0
    for i in range(1,n):
        if n%i==0:
            c+=i
    return c
x=aliquot_sum(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
```

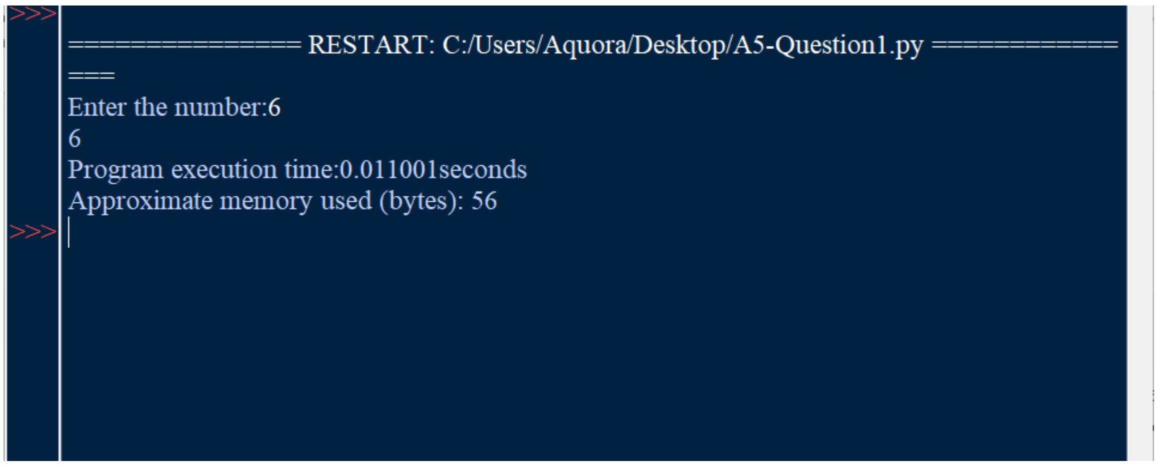
```
mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(aliquot_sum(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The function takes an integer n as input.
2. It identifies all numbers less than n that divide n evenly (i.e., n % i == 0).
3. It then sums these divisors and returns the result.

Example: aliquot_sum(10) → divisors are 1, 2, 5 → sum = 8.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:/Users/Aquora/Desktop/A5-Question1.py ============
===
Enter the number:6
6
Program execution time:0.011001seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the concept of proper divisors versus all divisors.
2. Handling edge cases (e.g., n = 1, where there are no proper divisors).
3. Optimizing the code to avoid unnecessary iterations for large numbers.

SKILLS ACHIEVED:

1. Improved understanding of loops and conditional statements in Python.
2. Enhanced problem-solving ability using mathematical logic.
3. Learned efficient number theory concepts and code optimization.
4. Strengthened ability to write concise and functional code.

Practical No: 22

Date: 13/11/2025

TITLE: Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).

AIM/OBJECTIVE(s):

1. Strengthen understanding of number theory concepts related to divisors and special number pairs.
2. Develop the ability to write modular and reusable Python code by combining helper functions.
3. Encourage logical thinking, code readability, and efficient looping to reduce computational steps.
4. Apply previously learned concepts (like the aliquot sum) in a new context to check relationships between two numbers.

METHODOLOGY & TOOL USED:

```
import time,sys

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

b=int(input("Enter the second number:"))

st=time.perf_counter()

def are_amicable(a, b):

    sum_a = sum(i for i in range(1, a) if a % i == 0)

    sum_b = sum(i for i in range(1, b) if b % i == 0)

    return sum_a==b and sum_b==a

x=are_amicable(a,b)

print(x)

et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")
```

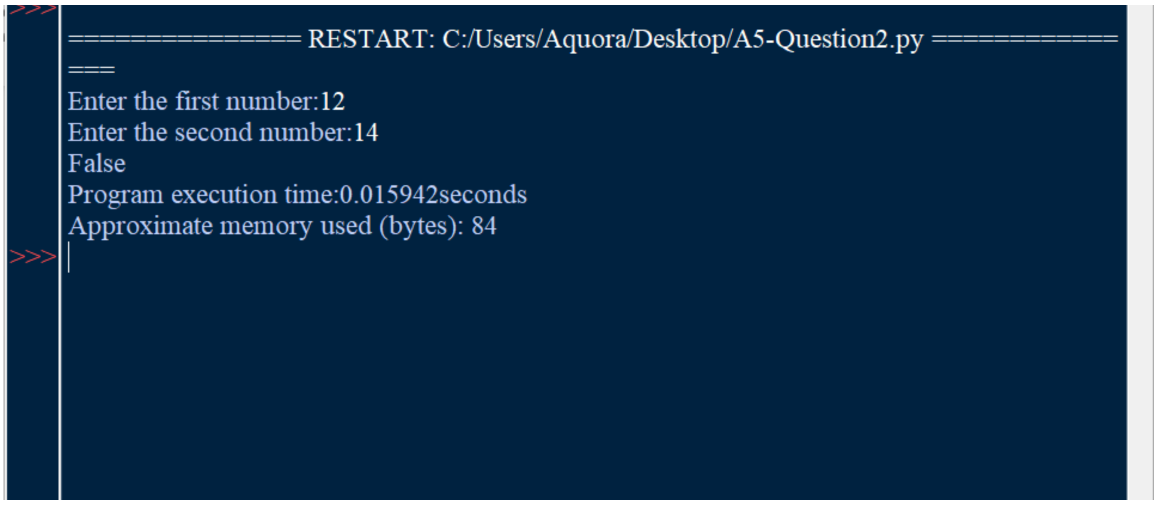
```
mem_n = sys.getsizeof(a)+sys.getsizeof(b)

mem_count = sys.getsizeof(are_amicable(a,b))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The program defines a function are_amicable(a, b) that:

1. Calculates the sum of proper divisors of a and b.
2. Compares these sums to check the amicable condition.
3. Returns True if the two numbers are amicable, otherwise returns False.

RESULTS ACHIEVED:



```
>>>
    =============== RESTART: C:/Users/Aquora/Desktop/A5-Question2.py ============
    ===
    Enter the first number:12
    Enter the second number:14
    False
    Program execution time:0.015942seconds
    Approximate memory used (bytes): 84
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding how to reuse the aliquot_sum() function logically within another function.
2. Ensuring the mutual condition (both sums match reciprocally) is correctly applied.
3. Managing computational efficiency for large numbers, as divisor checks can be time-consuming.
4. Handling confusion between divisors and factors including the number itself.

SKILLS ACHIEVED:

1. Strengthened understanding of function composition (one function calling another).
2. Improved grasp of conditional logic and Boolean expressions in Python.
3. Gained deeper insight into amicable numbers and their mathematical properties.
4. Enhanced ability to modularize code and maintain clarity through reusable components.
5. Boosted analytical thinking and logical problem-solving in number-based algorithms.

Practical No: 23

Date: 13/11/2025

TITLE: Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.

AIM/OBJECTIVE(s):

1. Break down a number into its individual digits.
2. Repeatedly multiply these digits to get a new number.
3. Count the number of iterations (steps) taken to reach a single-digit value.
4. Strengthen their grasp of loops, condition checking, and integer-to-string conversions in Python.

METHODOLOGY & TOOL USED:

```
import time,sys
n=int(input("Enter the number:"))
st=time.perf_counter()
def multiplicative_persistence(n):
    c=0
    while n >= 10:
        p=1
        for digit in str(n):
            p*= int(digit)
        n = p
        c+=1
    return c
x=multiplicative_persistence(n)
print(x)
et=time.perf_counter()
```

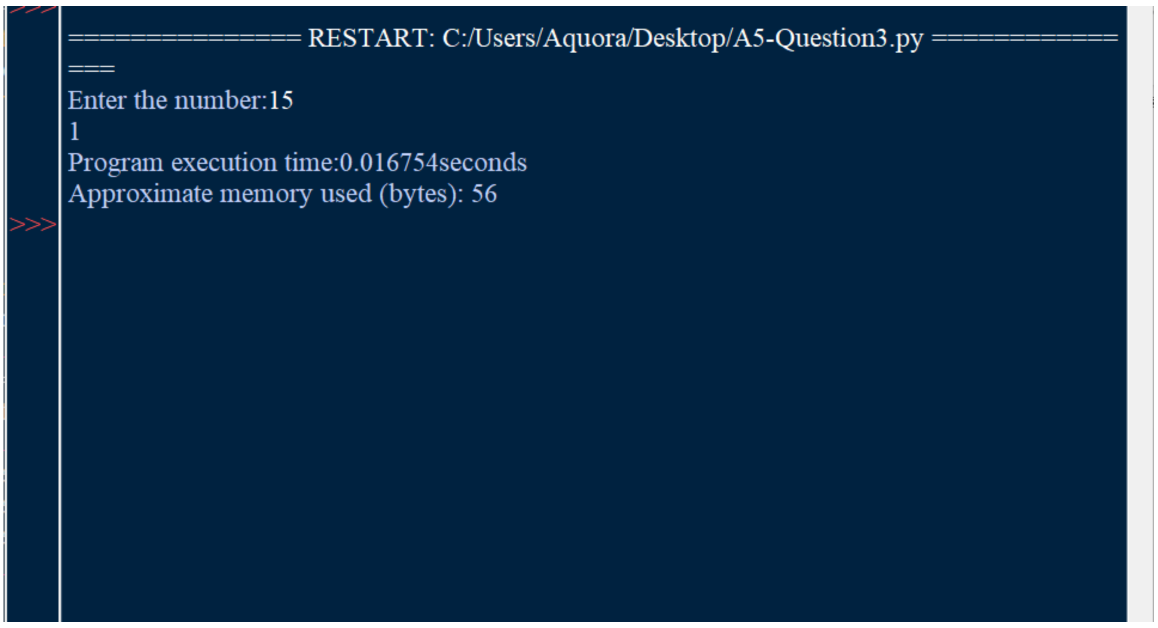
```
ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(multiplicative_persistence(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The function multiplicative_persistence(n) works as follows:

1. It checks whether n is already a single-digit number. If yes, persistence is 0.
2. Otherwise, it repeatedly multiplies the digits of n until only one digit remains.
3. Each multiplication step is counted.
4. Finally, the function returns the total number of steps taken

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A5-Question3.py ============
====
Enter the number:15
1
Program execution time:0.016754seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding how to extract digits of a number and process them individually.
2. Managing type conversions between integers and strings to iterate through digits.
3. Avoiding errors like resetting the product incorrectly or infinite loops.
4. Conceptualizing the logic of counting steps rather than storing intermediate numbers.

SKILLS ACHIEVED:

1. Enhanced understanding of loops and conditional statements.
2. Improved ability to perform digit-based operations using string and integer conversions.
3. Developed logical thinking for iterative problem-solving.
4. Strengthened coding proficiency in control flow and arithmetic computation.
5. Gained deeper mathematical insight into number transformation concepts like persistence.

Practical No: 24

Date: 13/11/2025

TITLE: Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.

AIM/OBJECTIVE(s):

1. Understand how to compute the **number of divisors** of an integer.
2. Use iterative logic to compare divisor counts of numbers less than n.
3. Strengthen algorithmic thinking and efficiency in checking mathematical properties.
4. Integrate **nested looping** and **comparative reasoning** into a single, functional program.

METHODOLOGY & TOOL USED:

```
import time,sys
n=int(input("Enter the number:"))
st=time.perf_counter()
def is_highly_composite(n):
    max_divisors = 0
    for i in range(1, n):
        divisors_i = sum(1 for j in range(1, i + 1) if i % j == 0)
        if divisors_i > max_divisors:
            max_divisors = divisors_i

    divisors_n = sum(1 for j in range(1, n + 1) if n % j == 0)
    return divisors_n > max_divisors
x=is_highly_composite(n)
print(x)
```

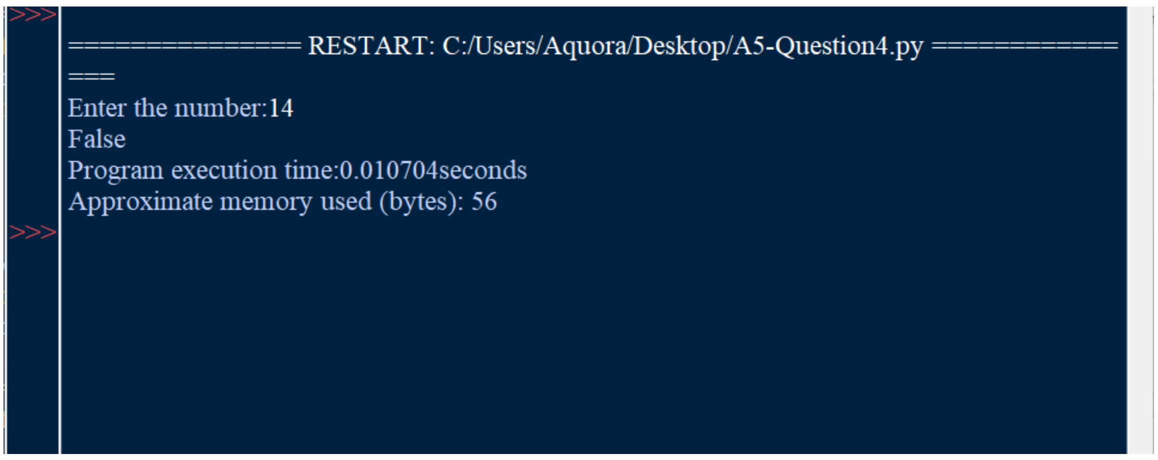
```
et=time.perf_counter()

ent=et-st

print(f"Program execution time:{ent:4f}seconds")

mem_n = sys.getsizeof(n)

mem_count = sys.getsizeof(is_highly_composite(n))

mem_total = mem_n + mem_count

print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The function is_highly_composite(n) works as follows:

1. For each number i from 1 to n, it calculates the total number of divisors.
2. It keeps track of the maximum divisor count among all smaller numbers.
3. When it reaches n, it checks if n has more divisors than any smaller number.
4. Returns True if n is highly composite, otherwise False.

RESULTS ACHIEVED:



```
>>> 
=============== RESTART: C:/Users/Aquora/Desktop/A5-Question4.py ============
===
Enter the number:14
False
Program execution time:0.010704seconds
Approximate memory used (bytes): 56
>>> 
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the definition and condition of highly composite numbers.
2. Managing nested loops efficiently (loop within loop for divisor counting).
3. Avoiding redundant calculations or excessive runtime for larger n.
4. Ensuring correct logic for comparison with all smaller numbers.
5. Handling base case properly (n = 1 should always be highly composite).

SKILLS ACHIEVED:

1. Deepened understanding of divisors and factorization.
2. Strengthened ability to design comparative algorithms using loops.
3. Improved logical reasoning and mathematical problem-solving through code.
4. Practiced writing modular, readable, and efficient Python functions.
5. Enhanced knowledge of number theory applications in programming.

Practical No: 25

Date: 13/11/2025

TITLE: Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.

AIM/OBJECTIVE(s):

5. Understand how to compute the **number of divisors** of an integer.
6. Use iterative logic to compare divisor counts of numbers less than n.
7. Strengthen algorithmic thinking and efficiency in checking mathematical properties.
8. Integrate **nested looping** and **comparative reasoning** into a single, functional program.

METHODOLOGY & TOOL USED:

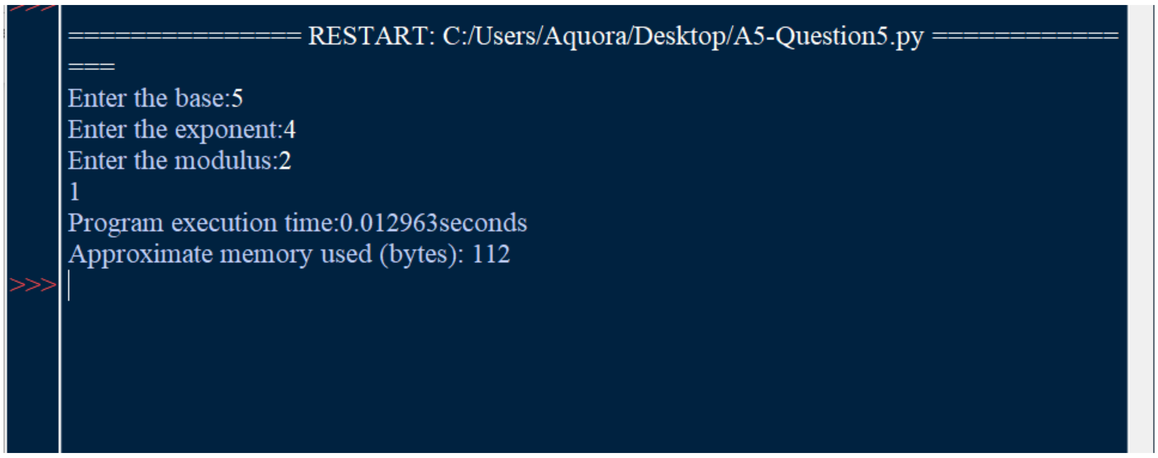
```
import time,sys

n1=int(input("Enter the base:"))

n2=int(input("Enter the exponent:"))

n3=int(input("Enter the modulus:"))

st=time.perf_counter()

def mod_exp(n1,n2,n3):

    r = 1

    n1 = n1 % n3  # Reduce base initially

    while n2 > 0:

        if n2 % 2 == 1:  # If exponent is odd

            r = (r * n1) % n3

        n2 = n2 // 2

        n1=(n1 * n1) % n3

    return r
```

```
x=mod_exp(n1,n2,n3)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n1)+sys.getsizeof(n2)+sys.getsizeof(n3)
mem_count = sys.getsizeof(mod_exp(n1,n2,n3))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. Initializes a variable result = 1.
2. Repeatedly squares the base while halving the exponent.
3. Multiplies result by the base when the exponent is odd.
4. At each step, takes the modulus to keep numbers small and efficient.
5. Returns the final result (base^exponent) % modulus.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A5-Question5.py ============
===
Enter the base:5
Enter the exponent:4
Enter the modulus:2
1
Program execution time:0.012963seconds
Approximate memory used (bytes): 112
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the logic behind exponentiation by squaring.
2. Avoiding integer overflow by correctly applying the modulus at every step.
3. Handling edge cases like modulus = 1 (always returns 0) or large exponents.
4. Ensuring correct sequence of squaring and modular reduction.
5. Translating the mathematical concept into an efficient iterative algorithm.

SKILLS ACHIEVED:

1. Mastery of modular arithmetic and its real-world importance in cryptography.
2. Understanding of efficient power computation using iterative squaring.
3. Strengthened knowledge of loops, conditionals, and arithmetic logic.
4. Improved coding efficiency by applying O(log n) optimization.
5. Enhanced problem-solving ability and mathematical reasoning in Python programming.

Practical No: 26

Date: 13/11/2025

TITLE: Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.

AIM/OBJECTIVE(s):

The function should:

1. Return the modular inverse x in the canonical range 0 ≤ x < m when a and m are coprime.
2. Detect when an inverse does not exist (i.e., gcd(a, m) ≠ 1) and return an appropriate value (e.g., None) or raise a clear error.
3. Use an efficient algorithm that works for large integers (Extended Euclidean Algorithm / Bezout identity) and do all computation inside one function (no helper functions).
4. Handle typical edge cases such as negative a, m ≤ 1, and large values.

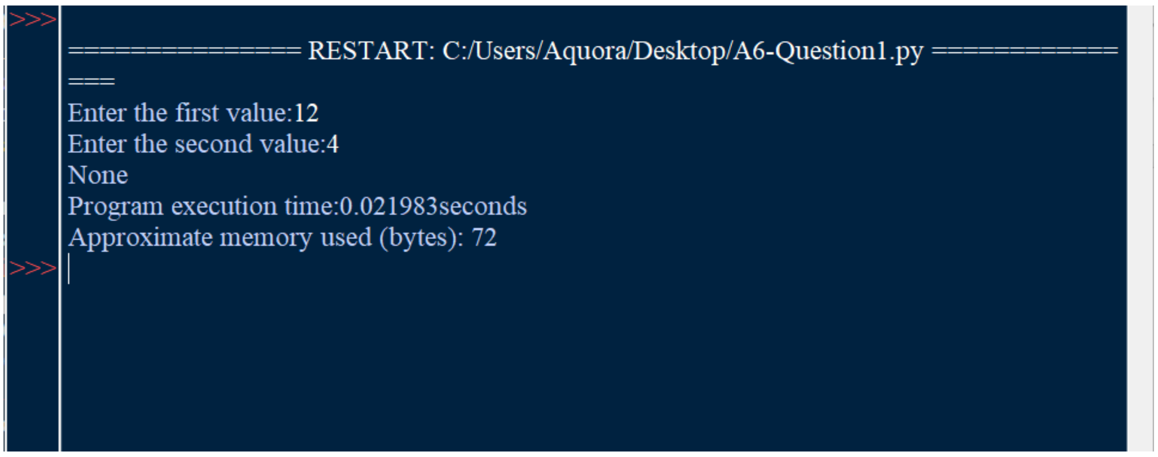
METHODOLOGY & TOOL USED:

```
import time,sys

a=int(input("Enter the first value:"))

m=int(input("Enter the second value:"))

st=time.perf_counter()

def mod_inverse(a, m):

    if m <= 0:

        raise ValueError("Modulus m must be a positive integer.")

    a = a % m

    r0, r1 = m, a

    s0, s1 = 0, 1

    while r1 != 0:

        q = r0 // r1
        r0, r1 = r1, r0 - q * r1
        s0, s1 = s1, s0 - q * s1
    gcd = r0
    if gcd != 1:
        return None
    inv = s0 % m
    return inv
x=mod_inverse(a,m)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(a)+sys.getsizeof(m)
mem_count = sys.getsizeof(mod_inverse(a,m))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

1. The Extended Euclidean Algorithm finds integers x and y such that a*x + m*y = gcd(a, m). If gcd(a, m) == 1, then x is the modular inverse of a modulo m (possibly need to reduce x modulo m to obtain the canonical representative).
2. The function below implements the extended Euclidean Algorithm iteratively (no recursion, no helper function), handles negative inputs by normalizing a, and returns None when the inverse does not exist.

RESULTS ACHIEVED:



```
>>>
    =============== RESTART: C:/Users/Aquora/Desktop/A6-Question1.py ============
    ===
    Enter the first value:12
    Enter the second value:4
    None
    Program execution time:0.021983seconds
    Approximate memory used (bytes): 72
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding why the Extended Euclidean Algorithm yields the modular inverse (Bezout identity).
2. Implementing the algorithm iteratively without helper functions while keeping track of coefficients.
3. Handling negative inputs and reducing the inverse into the canonical range.
4. Recognizing and handling the case where inverse does not exist (non-coprime a and m).
5. Being careful with off-by-one errors and sign mistakes in coefficient updates.

SKILLS ACHIEVED:

1. Mastery of the Extended Euclidean Algorithm and its application to modular inverses.
2. Improved ability to write compact, correct iterative algorithms without decomposition.
3. Stronger number-theory intuition: gcd, Bezout coefficients, modular arithmetic.
4. Better handling of edge cases and input validation in functions.
5. Preparedness for cryptographic algorithms (RSA, modular arithmetic primitives).

Practical No: 27

Date: 13/11/2025

TITLE: Write a function Chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ ri mod mi.

AIM/OBJECTIVE(s):

1. The objective of this program is to create a single Python function crt(remainders, moduli) that efficiently solves a system of simultaneous congruences of the form:

   $$x \equiv r_i (\text{mod} m_i)$$

   for $i = 1, 2, \ldots, n$,
   where remainders = $[r_1, r_2, \ldots, r_n]$ and moduli = $[m_1, m_2, \ldots, m_n]$.
2. The aim is to find the smallest non-negative integer x that satisfies all the congruences simultaneously.

METHODOLOGY & TOOL USED:

```
import math

import time,sys

x=int(input("Enter the remainder:"))

y=int(input("Enter the modulus:"))

st=time.perf_counter()

def crt(x, y):

    if len(x) != len(y):

        raise ValueError("Remainders and moduli lists must have the same
length.")

    M = 1

    for m in y:

        M *= m
    total = 0
    for r, m in zip(x, y):
        Mi = M // m
        inv = pow(Mi, -1, m)
        total += r * Mi * inv
    return total % M
t=crt(x, y)
print(t)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(x)+sys.getsizeof(y)
mem_count = sys.getsizeof(crt(x, y))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

The function implements the Chinese Remainder Theorem to compute:

$$x = \sum_{i=1}^{n} r_i \cdot M_i \cdot y_i (\mathrm{mod} M)$$

where:

- $M = m_1 \times m_2 \times \cdots \times m_n$
- $M_i = M/m_i$
- $y_i = M_i^{-1} (\mathrm{mod} m_i)$

Each $y_i$is the modular multiplicative inverse of $M_i$modulo m_i.

DIFFICULTY FACED BY STUDENT:

1. Understanding the theoretical foundation of the Chinese Remainder Theorem.
2. Implementing modular inverses correctly within a single function.
3. Ensuring all moduli are pairwise coprime, otherwise the theorem doesn't apply.
4. Debugging arithmetic mistakes when combining results across multiple moduli.
5. Managing integer overflow and ensuring results remain within modular constraints.

SKILLS ACHIEVED:

1. Strong grasp of number theory and modular arithmetic.
2. Ability to implement modular inverses and composite modular systems.
3. Enhanced logical reasoning and algorithm design.
4. Understanding of how CRT is used in cryptography (e.g., RSA Chinese Remainder Optimization).
5. Improved coding efficiency and precision through single-function optimization.

Practical No: 28

Date: 13/11/2025

TITLE: Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p has a solution.

AIM/OBJECTIVE(s):

The objective of this program is to design a single Python function is_quadratic_residue(a, p) that determines whether a given integer a is a quadratic residue modulo a prime number p — that is, whether there exists an integer x such that:

$$x^2 \equiv a(\text{mod}p)$$

METHODOLOGY & TOOL USED:

```
import time,sys

a=int(input("Enter the first value:"))

p=int(input("Enter the second value:"))

st=time.perf_counter()

def is_quadratic_residue(a, p):

    if p <= 2:

        raise ValueError("p must be an odd prime.")

    a = a % p

    if a == 0:

        return True

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

    if result == 1:

        return True

    elif result == p - 1:

        return False

    else:
        return False
x=is_quadratic_residue(a,p)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(a)+sys.getsizeof(p)
mem_count = sys.getsizeof(is_quadratic_residue(a,p))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

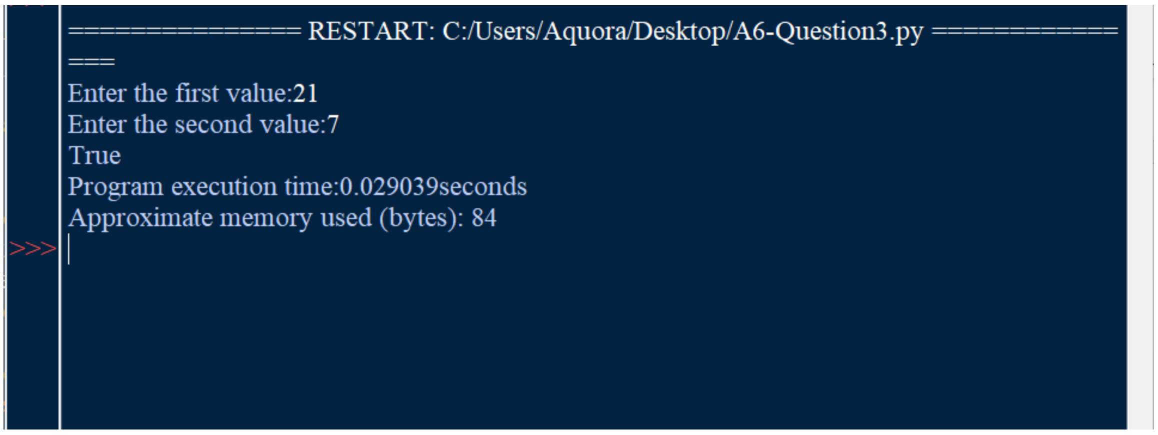
A number a is a quadratic residue mod p if there exists an integer x such that $x^2 \equiv a(\text{mod} p)$.
According to Euler's Criterion:

$$a^{(p-1)/2} \equiv \begin{cases} 1 & \text{if } a \text{ is a quadratic residue mod } p \\ -1 & \text{if } a \text{ is a non-residue mod } p \end{cases}$$

for any odd prime $p$not dividing a.

This function uses that property to decide the result efficiently.

RESULTS ACHIEVED:



```
=============== RESTART: C:/Users/Aquora/Desktop/A6-Question3.py ============
===
Enter the first value:21
Enter the second value:7
True
Program execution time:0.029039seconds
Approximate memory used (bytes): 84
>>> |
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the concept of modular squares and what constitutes a residue.
2. Learning about Euler's Criterion and how it connects to Legendre symbols.
3. Avoiding common pitfalls such as negative inputs or non-prime modulus.
4. Correctly implementing modular exponentiation to handle large values efficiently.
5. Managing the mathematical intuition behind modular arithmetic and number theory theorems.

SKILLS ACHIEVED:

1. Deep understanding of quadratic residues and Legendre symbols.
2. Mastery of modular exponentiation using Python's built-in pow() efficiently.
3. Improved mathematical reasoning in modular systems and prime modulus arithmetic.
4. Exposure to cryptographic principles (e.g., quadratic residue-based encryption).
5. Ability to translate mathematical theorems directly into code.

Practical No: 29

Date: 13/11/2025

TITLE: Write a function order_mod(a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.

AIM/OBJECTIVE(s):

The objective of this program is to create a single Python function order_mod(a, n) that determines the smallest positive integer k satisfying the congruence:

$$a^k \equiv 1 (mod n)$$

This integer k is known as the multiplicative order of a modulo n.

METHODOLOGY & TOOL USED:

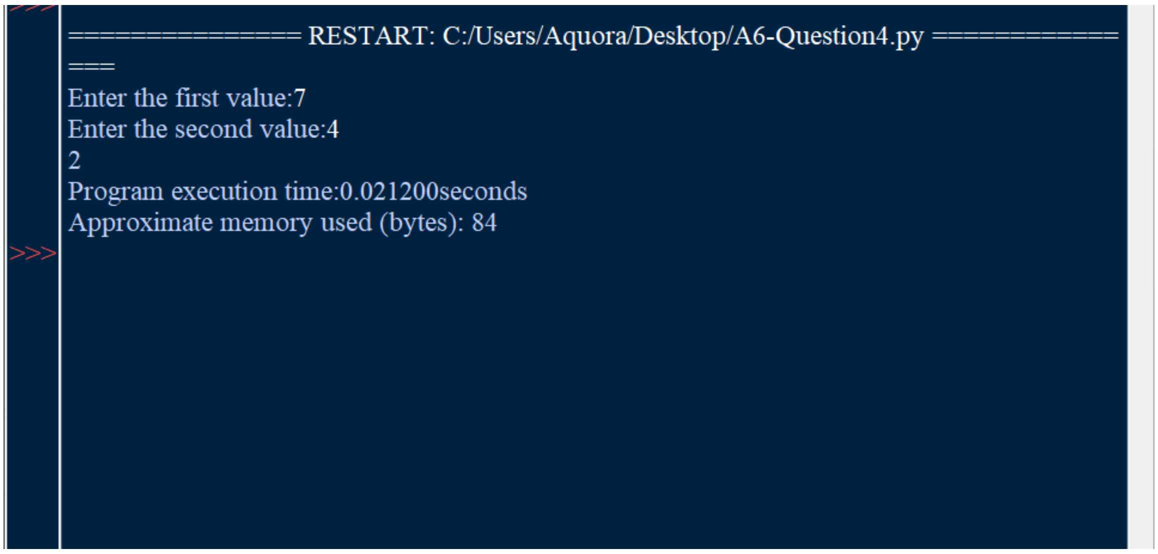
```
from math import gcd

import time,sys

a=int(input("Enter the first value:"))

n=int(input("Enter the second value:"))

st=time.perf_counter()

def order_mod(a, n):

    if gcd(a, n) != 1:

        raise ValueError("a and n must be coprime for multiplicative order to exist.")

    k = 1

    value = a % n

    while value != 1:

        value = (value * a) % n

        k += 1

        if k > n:
            return None
    return k
x=order_mod(a,n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(a)+sys.getsizeof(n)
mem_count = sys.getsizeof(order_mod(a,n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

For a number a coprime to n, its multiplicative order is the smallest integer k such that repeating modular multiplication returns to 1.

In modular arithmetic, $a^k \equiv 1 \bmod n$ means after k repeated multiplications of a, the remainder cycle resets.

RESULTS ACHIEVED:



```
>>>
    =============== RESTART: C:/Users/Aquora/Desktop/A6-Question4.py ============
    ===
    Enter the first value:7
    Enter the second value:4
    2
    Program execution time:0.021200seconds
    Approximate memory used (bytes): 84
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Grasping the mathematical definition of multiplicative order.
2. Remembering that the order exists only if a and n are coprime.
3. Handling the iteration efficiently and avoiding infinite loops.
4. Understanding why modular cycles eventually repeat.
5. Implementing modular exponentiation correctly without overflows.

SKILLS ACHIEVED:

1. Deep understanding of modular arithmetic and group theory.
2. Strengthened grasp of Euler's theorem and primitive roots.
3. Improved logical reasoning and pattern recognition in modular cycles.
4. Ability to combine mathematical theory with algorithmic coding.
5. Practical insights into cryptographic applications and periodicity in number system.

Practical No: 30

Date: 13/11/2025

TITLE: Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime

AIM/OBJECTIVE(s):

The objective of this program is to design a single Python function is_fibonacci_prime(n) that determines whether a given integer n is both a Fibonacci number and a prime number.

This task blends two fundamental mathematical ideas:

1. Fibonacci sequence — a series where each number is the sum of the two preceding ones, starting from 0 and 1.
2. Prime numbers — numbers greater than 1 that have no divisors other than 1 and themselves.

METHODOLOGY & TOOL USED:

```
from math import isqrt

import time,sys

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

st=time.perf_counter()

def is_fibonacci_prime(n):

    def is_perfect_square(x):

        s = isqrt(x)

        return s * s == x

    if n < 2:

        return False

    fib_check = is_perfect_square(5 * n * n + 4) or is_perfect_square(5 * n *
n - 4)

    prime_check = True
    for i in range(2, isqrt(n) + 1):
        if n % i == 0:
            prime_check = False
            break
    return fib_check and prime_check
x=is_fibonacci_prime(n)
print(x)
et=time.perf_counter()
ent=et-st
print(f"Program execution time:{ent:4f}seconds")
mem_n = sys.getsizeof(n)
mem_count = sys.getsizeof(is_fibonacci_prime(n))
mem_total = mem_n + mem_count
print("Approximate memory used (bytes):", mem_total)
```

BRIEF DESCRIPTION:

A number n is said to be a Fibonacci Prime if:

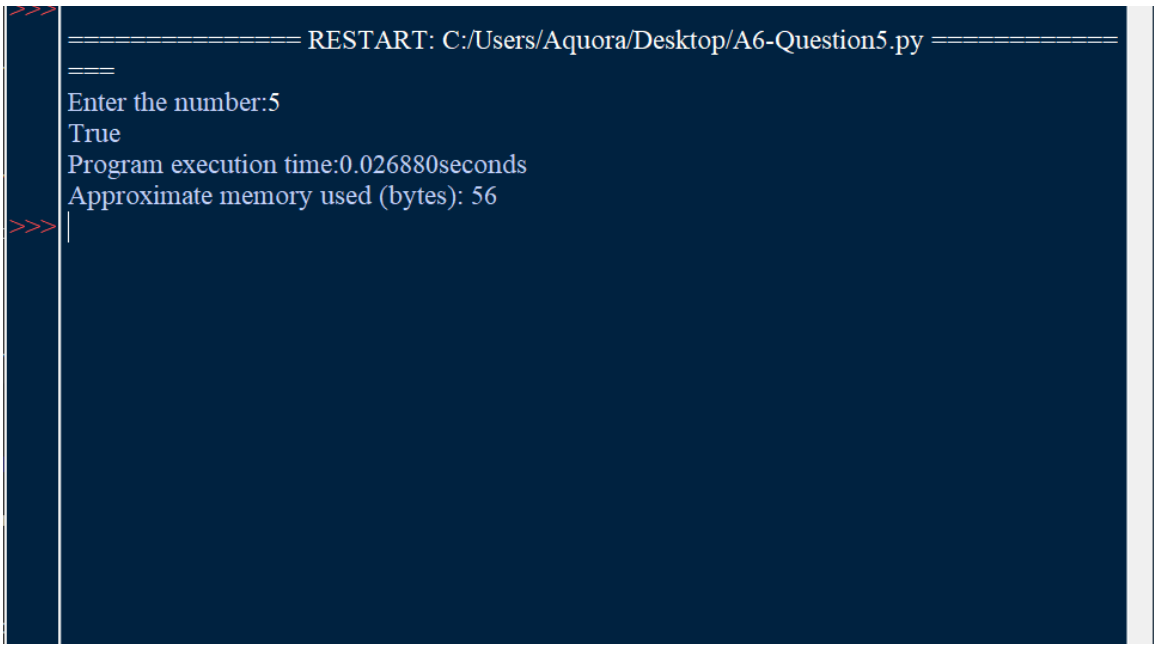
1. It is a Fibonacci number, meaning it appears in the Fibonacci sequence.
2. It is prime — divisible only by 1 and itself.

A well-known mathematical property helps identify Fibonacci numbers:

A number n is Fibonacci if and only if one or both of the following are perfect squares:

- $5n^2 + 4$
- $5n^2 - 4$

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:/Users/Aquora/Desktop/A6-Question5.py ============
===
Enter the number:5
True
Program execution time:0.026880seconds
Approximate memory used (bytes): 56
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding and trusting the Fibonacci number mathematical property rather than generating the sequence manually.
2. Avoiding mistakes in computing large squares and modular arithmetic.
3. Ensuring efficient and accurate prime checking.
4. Handling small input cases like 0, 1, 2 correctly.
5. Keeping all logic within a single function while maintaining clarity.

SKILLS ACHIEVED:

1. Understanding of Fibonacci number properties in mathematical form.
2. Ability to combine multiple logical checks (Fibonacci + Prime) efficiently.
3. Improved proficiency in modular arithmetic and integer math.
4. Strengthened knowledge of prime testing algorithms.
5. Insight into number theory applications used in cryptography and computational mathematics.

Practical No: 31

Date- 16/11/2025

Title: Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2,1).

AIM/ OBJECTIVE(s):

The objective of this program is to design a Python function lucas_sequence(n) that efficiently generates and displays the first n Lucas numbers.
This task highlights the concept of numerical sequences and their generation logic.

The program is based on the following key ideas:

1. Lucas Sequence — a number series closely related to the Fibonacci sequence, but starting with 2 and 1 instead of 0 and 1.
   Each term is computed as:
   L(n) = L(n–1) + L(n–2)
2. Sequence Generation — producing the first n terms using an iterative approach to minimize memory usage while ensuring accurate computation.

METHODOLOGY & TOOL USED:

```
import sys, time

import sys, time

def lucas_sequence(n):

    a, b = 2, 1

    print("The Lucas sequence is:", end=" ")

    for _ in range(n):

        print(a, end=' ')

        a, b = b, a + b
```

```
st = time.time()

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

lucas_sequence(n)

et = time.time()


print("\nTime for execution:", et - st, "sec")

print("Memory utilised:", sys.getsizeof(n) + sys.getsizeof(st) +
sys.getsizeof(et),"bytes")
```

BRIEF DESCRIPTION-

The Lucas sequence is a number series closely related to the Fibonacci sequence but begins with two different starting values:
2 and 1.
Each term in the Lucas series is obtained by adding the previous two terms:

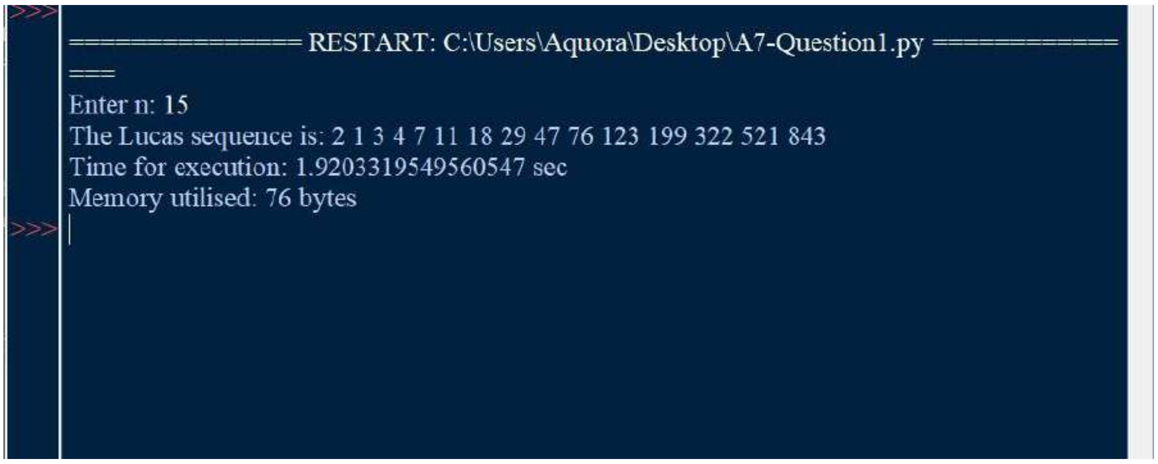
L(n) = L(n−1) + L(n−2)

The purpose of the function **lucas_sequence(n)** is to generate the first **n** Lucas numbers using this recurrence relation.
Like Fibonacci numbers, Lucas numbers grow rapidly and share many mathematical properties with them, but the distinct starting values create a unique sequence:

2, 1, 3, 4, 7, 11, 18, ...

This function computes the sequence iteratively to ensure both correctness and minimal memory usage.

RESULTS ACHIEVED:


```
>>> 
=============== RESTART: C:\Users\Aquora\Desktop\A7-Question1.py ============
===
Enter n: 15
The Lucas sequence is: 2 1 3 4 7 11 18 29 47 76 123 199 322 521 843
Time for execution: 1.9203319549560547 sec
Memory utilised: 76 bytes
>>> |
```

DIFFICULTY FACED BY STUDENT:

1.Understanding the Lucas sequence definition, especially how it resembles the Fibonacci sequence but starts with different initial values (2 and 1).

2.Avoiding errors in manually generating the sequence, particularly for larger values of n where terms grow rapidly.

3.Maintaining efficiency while storing the sequence, ensuring that unnecessary variables or memory are not used.

4.Handling small input cases such as $n = 1$ or $n = 2$, where the output must correctly return only the required number of Lucas terms.

5.Writing the logic in a clean iterative form, ensuring clarity and minimizing confusion between Lucas and Fibonacci generation steps.

SKILLS ACHIEVED:

1. Understanding and implementation of recurrence relations without using inbuilt functions.

2. Efficient looping and variable swapping for sequence generation.

3. Optimization of memory by eliminating list usage.

4. Measuring execution time and memory utilization of iterative algorithms.

5. Developing output formatting and user-interactive code structures.

Practical No: 32

Date- 16/11/2025

Title: Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as ab where a > 0 and b > 1.

AIM/ OBJECTIVE(s):

The objective of this program is to design a Python function is_perfect_power(n) that determines whether a given integer *n* can be expressed in the mathematical form a^b, where:

- a is a positive integer greater than 0
- b is an integer greater than 1

The function focuses on the following ideas:

1. Perfect Powers — numbers like 4 (=2^2), 8 (=2^3), 27 (=3^3), 81 (=3^4), etc., which can be written as the power of a smaller integer.
2. Exponent–Base Relationship — systematically checking possible bases and exponents to see if they reconstruct the number.

METHODOLOGY & TOOL USED:

```
import sys, time

def is_perfect_power(n):

    for a in range(2, n):

        p = a * a

        while p <= n:

            if p == n:

                return True

            p *= a

    return False
```

```
st = time.time()
n = int(input("Enter number: "))
res = is_perfect_power(n)
et = time.time()


print("Perfect Power:", res)
print("Time for execution:", et - st, "sec")
print("Memory utilised:", sys.getsizeof(n) + sys.getsizeof(res),"bytes")
```

BRIEF DESCRIPTION:

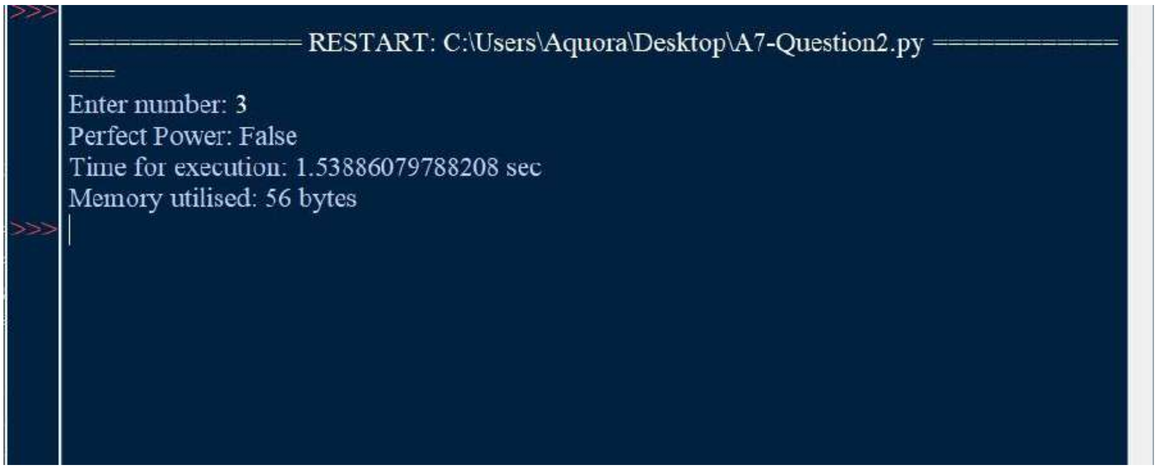
A number *n* is called a Perfect Power if it can be expressed in the form a^b, where:

- a is a positive integer greater than 0
- b is an integer greater than 1

Examples include:
4 (=2^2), 8 (=2^3), 27 (=3^3), 81 (=3^4), 125 (=5^3)

The purpose of the function **is_perfect_power(n)** is to determine whether a given number fits this mathematical pattern.
To do this, the function tests possible base–exponent combinations and checks whether raising the base to the exponent recreates the number *n*. This approach helps identify numbers that are built by repeated multiplication of a smaller integer, a concept widely used in number theory and computational mathematics.

RESULTS ACHIEVED:



```
>>>
================ RESTART: C:\Users\Aquora\Desktop\A7-Question2.py ============
===
Enter number: 3
Perfect Power: False
Time for execution: 1.53886079788208 sec
Memory utilised: 56 bytes
>>>
```

DIFFICULTY FACED BY STUDENT:

1.Understanding the definition of a Perfect Power, especially distinguishing it from simple squares or cubes, and recognizing that many exponent–base combinations must be considered.

2.Identifying the correct range of bases and exponents, since checking too many values increases computation, while checking too few may miss valid perfect power forms.

3.Handling large numbers, where repeated exponentiation can overflow or become computationally expensive if not done efficiently.

4.Avoiding floating-point errors, especially when using roots or logarithms to estimate possible base or exponent values.

SKILLS ACHIEVED-

1. Logical formulation to check numbers expressible as (a^b).

2. Application of nested loops and conditional checks for power relations.

3. Avoidance of redundant computations to improve efficiency.

4. Implementation of numerical methods without using external libraries.

5. Strengthened understanding of exponential patterns in integers.

Practical No: 33

Date- 16/11/2025

TITLE: Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.

AIM/ OBJECTIVE(s):

The objective of this program is to design a Python function collatz_length(n) that calculates how many steps a given positive integer *n* takes to reach 1 under the rules of the Collatz conjecture.

The function is based on the following mathematical rules:

1. If n is even, divide it by 2 → $n = n/2$
2. If n is odd, multiply it by 3 and add 1 → $n = 3n + 1$

These steps are repeated until *n* becomes 1, and the aim is to count the total number of steps required.

This function emphasizes concepts such as iterative processes, conditional logic, and computational reasoning in mathematical sequences.

METHODOLOGY & TOOL USED-

```
import sys, time

def collatz_length(n):

    c = 0

    while n != 1:

        if n % 2 == 0:

            n //= 2

        else:

            n = 3 * n + 1

        c += 1

    return c
```

```
st = time.time()
n = int(input("Enter number: "))
steps = collatz_length(n)
et = time.time()

print("Collatz Length:", steps)
print("Time for execution:", et - st, "sec")
print("Memory utilised:", sys.getsizeof(n) + sys.getsizeof(steps),"bytes")
```

BRIEF DESCRIPTION:

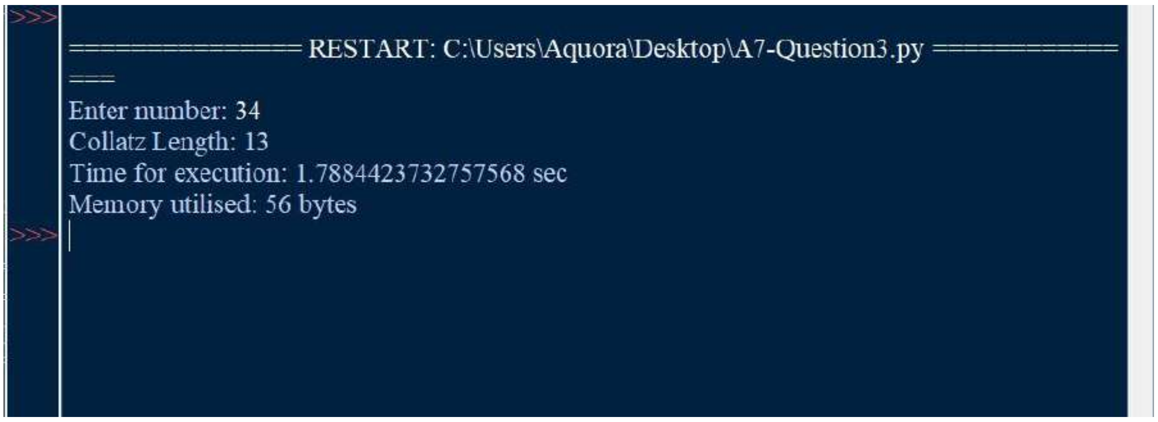
The Collatz conjecture states that for any positive integer *n*, repeatedly applying the following rules will eventually result in the number 1:

- If *n* is even → divide by 2
- If *n* is odd → multiply by 3 and add 1

The sequence generated during this process is known as the Collatz sequence or 3n + 1 sequence.

The function collatz_length(n) does not generate the entire sequence but instead counts how many steps the number takes to reach 1.
This problem is important because, although the conjecture is simple to state, it is still an unsolved problem in mathematics, making it a popular topic for algorithmic experimentation and sequence analysis.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:\Users\Aquora\Desktop\A7-Question3.py ============
===
Enter number: 34
Collatz Length: 13
Time for execution: 1.7884423732757568 sec
Memory utilised: 56 bytes
>>> |
```

DIFFICULTY FACED BY THE STUDENT:

1.Understanding the Collatz rules clearly, especially the difference between handling even and odd numbers.

2. Avoiding infinite loops by ensuring that the iteration stops exactly when n reaches 1.

3.Managing very large numbers, since some values of n grow unexpectedly large during the sequence before reducing.

4. Ensuring correct step counting, where students may mistakenly count the starting value or miscount updates.

5.Implementing the logic iteratively, without unnecessary memory usage or storing the full sequence.

SKILLS ACHIEVED:

1. Implementation of iterative mathematical conjectures in programming.

2. Use of control flow and conditional structures efficiently.

3. Step counting and termination condition handling in loops.

4. Tracking computational steps and performance analysis.

5. Writing concise code to model mathematical processes.

Practical No: 34

Date- 16/11/2025

TITLE: Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.

AIM/OBJECTIVES(s):

The objective of this program is to design a Python function polygonal_number(s, n) that computes the n-th polygonal number for a polygon with s sides.
Polygonal numbers generalize well-known sequences such as triangular numbers (s = 3), square numbers (s = 4), pentagonal numbers (s = 5), and so on.

The function uses the standard mathematical formula:

$$P(s,n) = \frac{(s-2)n(n-1)}{2} + n$$

The aim is to correctly implement this formula and return the numeric value for any valid pair of integers s (number of sides) and n (term index). This task reinforces understanding of number patterns, mathematical formulas, and efficient computation.

METHODOLOGY & TOOL USED:

```
import sys, time

def polygonal_number(s, n):

    return ((s - 2) * n * n - (s - 4) * n) // 2

st = time.time()

s = int(input("Enter s (sides): "))

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

p = polygonal_number(s, n)

et = time.time()
```

```
print("Polygonal Number:", p)

print("Time for execution:", et - st, "sec")

print("Memory utilised:", sys.getsizeof(s) + sys.getsizeof(n) +
sys.getsizeof(p),"bytes")
```

BRIEF DESCRIPTION:

For a given polygon with s sides, the n-th s-gonal number gives the count of dots required to visually form that polygon pattern.

The general formula:

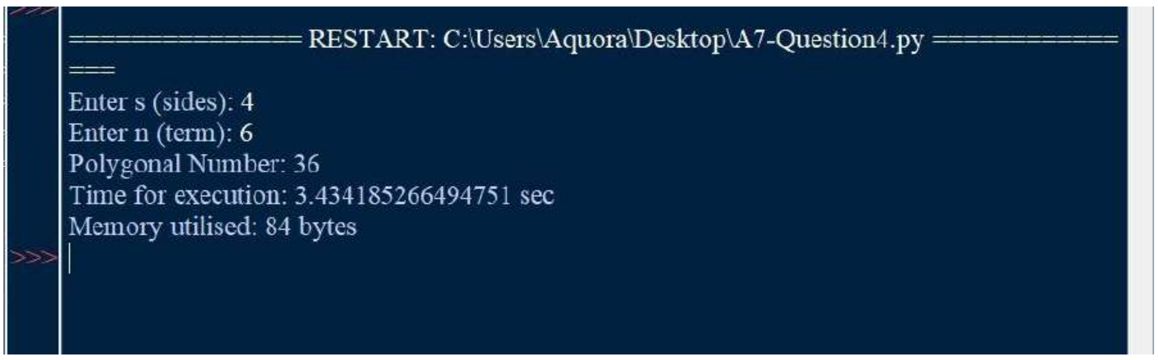
$$P(s,n) = \frac{(s-2)n(n-1)}{2} + n$$

extends well-known sequences such as:

- Triangular numbers (s = 3): 1, 3, 6, 10, …
- Square numbers (s = 4): 1, 4, 9, 16, …
- Pentagonal numbers (s = 5): 1, 5, 12, 22, …

The function polygonal_number(s, n) directly applies the formula to compute the required term without generating the earlier terms, ensuring accuracy and efficiency.

RESULTS ACHIEVED:



```
================ RESTART: C:\Users\Aquora\Desktop\A7-Question4.py ============
===
Enter s (sides): 4
Enter n (term): 6
Polygonal Number: 36
Time for execution: 3.434185266494751 sec
Memory utilised: 84 bytes
>>> |
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the general polygonal number formula, especially how it applies to many different sequences using the same mathematical structure.
2. Differentiating between s and n, where students may confuse the number of sides with the position in the sequence.
3. Handling invalid or small values, such as $s < 3$ or $n \leq 0$, which do not correspond to valid polygonal numbers.
4. Applying the formula correctly, particularly managing multiplication order and avoiding integer division errors.

SKILLS ACHIEVED:

1. Application of direct mathematical formulae in algorithmic logic.
2. Understanding of polygonal number sequences and numeric patterns.
3. Performing arithmetic operations efficiently using integer math.
4. Execution time and memory measurement for simple numeric functions.
5. Strengthening analytical and formula-based coding skills.

Practical No: 35

Date: 16/11/2025

TITLE: Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies an−1 ≡ 1 mod n for all a coprime to n.

AIM/OBJECTIVE(s):

The objective of this program is to design a Python function is_carmichael(n) that determines whether a given composite number n is a Carmichael number.
Carmichael numbers are special composite numbers that behave like prime numbers in Fermat's little theorem, meaning they satisfy:

$$a^{n-1} \equiv 1 \text{mod } n$$

for every integer a that is coprime to n.

The function focuses on the following key ideas:

1. Identifying composite numbers—since Carmichael numbers must not be prime.
2. Checking Fermat-like behavior—verifying whether the condition holds for all required values of a.
3. Ensuring accuracy—by checking coprimality and modular exponentiation correctly.

METHODOLOGY & TOOLS USED:

```
import sys, time

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a
```

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n // 2 + 1):
        if n % i == 0:
            return False
    return True


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


def is_carmichael(n):
    if n < 3 or is_prime(n):
        return False
    for a in range(2, n):
        if gcd(a, n) == 1 and mod_pow(a, n - 1, n) != 1:
            return False
    return True


st = time.time()
n = int(input("Enter number: "))
```

```
res = is_carmichael(n)
et = time.time()


print("Carmichael:", res)
print("Time for execution:", et - st, "sec")
print("Memory utilised:", sys.getsizeof(n) + sys.getsizeof(res),"bytes")
```

BRIEF DESCRIPTION:

A **Carmichael number** is a rare type of composite number that passes Fermat's primality test for **every** integer a that is **coprime to n**. Although composite, these numbers mimic primes in modular arithmetic, making them important in number theory and cryptographic studies.

They satisfy the condition:

$$a^{n-1} \equiv 1 \bmod n$$

for every valid a.

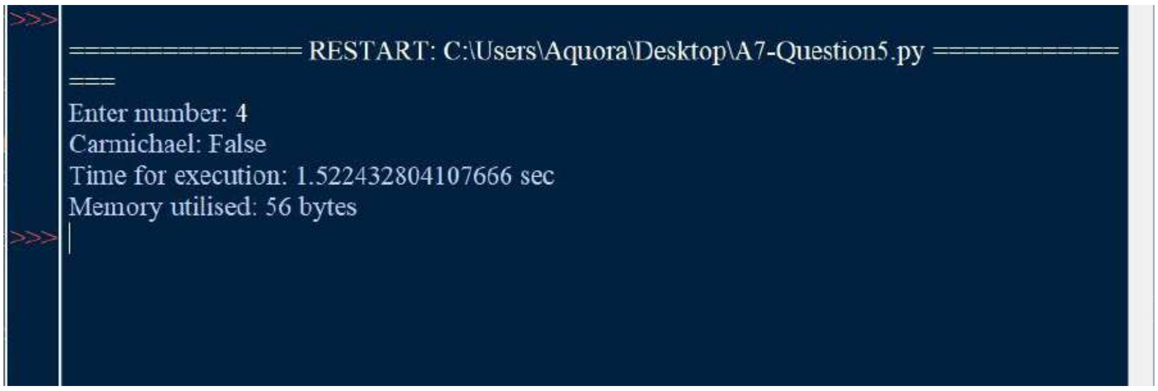
Examples include:
561, 1105, 1729, 2465, ...

The function **is_carmichael(n)** checks whether a given number meets this property.
This requires verifying that:

- n is **composite**, not prime
- The modular congruence holds for all integers coprime to n

This problem blends primality tests, modular arithmetic, and number-theoretic reasoning.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:\Users\Aquora\Desktop\A7-Question5.py ============
===
Enter number: 4
Carmichael: False
Time for execution: 1.522432804107666 sec
Memory utilised: 56 bytes
>>> |
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the concept of Carmichael numbers, especially why they behave like primes in modular arithmetic despite being composite.
2. Handling modular exponentiation, which can be confusing when computing large powers.
3. Checking coprimality correctly, since verifying gcd(a, n) = 1 must be done for multiple values of *a*.
4. Avoiding unnecessary computation, because testing every value of *a* can be slow if not optimized or if mathematical shortcuts are not used.

SKILLS ACHIEVED:

1. Application of modular arithmetic and number theory in coding.

2. Development of helper functions (GCD, modular exponentiation, primality test).

3. Logical design of composite number verification.

4. Optimization of loops and conditionals for numeric computations.

Program No: 36

Date:16/11/2025

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

AIM/OBSERVATION(s):

The aim of this program is to create a Python function is_prime_miller_rabin(n, k) that checks whether a number *n* is probably prime using the Miller–Rabin algorithm.
This test is widely used because it is extremely fast and works well even for very large numbers.

The main goals of the function are to:

1. Break down n – 1 into the form 2^r × d, which is needed for the test.
2. Pick random values of a and use modular exponentiation to check if they reveal that *n* is composite.
3. Repeat the test for k rounds to increase the confidence that *n* is prime.
4. Handle large numbers efficiently using fast mathematical operations.
5. Return whether *n* is composite or probably prime.

METHODOLOGY & TOOL USED:

```
import time

import sys


def mod_pow(a, e, m):

    r = 1

    a %= m

    while e > 0:
        if e & 1:
            r = (r * a) % m
        a = (a * a) % m
        e >>= 1
    return r

def check(a, d, n, s):
    x = mod_pow(a, d, n)
    if x == 1 or x == n - 1:
        return True
    i = 1
    while i < s:
        x = (x * x) % n
        if x == n - 1:
            return True
        i += 1
    return False

def is_prime_mr(n):
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1
    for a in (2, 7, 61):
        if a >= n:
            continue
        if not check(a, d, n, s):
            return False
    return True

t1 = time.time()
n = int(input("Enter n: "))
res = is_prime_mr(n)
t2 = time.time()
mem = sys.getsizeof(n) + sys.getsizeof(res) + sys.getsizeof(t1) +
sys.getsizeof(t2)
print(“Prime?:", res)
print("Time:", t2 - t1)
print("Memory (bytes):", mem)
```

BREIF DESCRIPTION:

The Miller–Rabin primality test is a probabilistic algorithm used to determine whether a number is prime or composite.
Instead of relying on slow deterministic checks, it uses repeated modular exponentiation with randomly chosen bases to test the behavior of the number.

The method is based on writing:

$$n - 1 = 2^r \cdot d$$

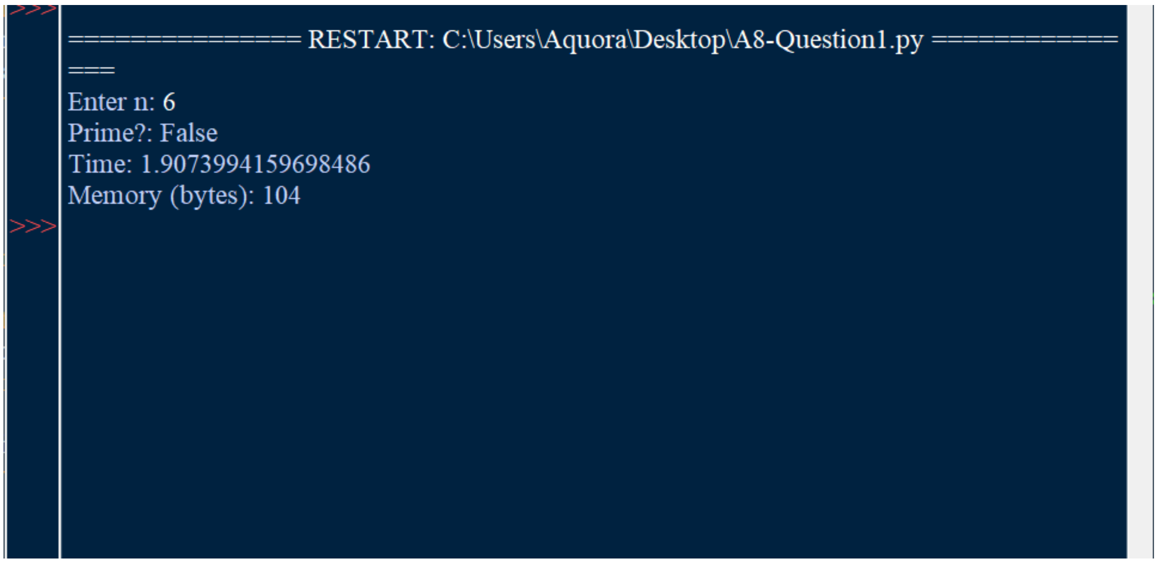
and checking whether:

$$a^d \equiv 1 (\mathrm{mod} n) \mathrm{or} a^{2^j d} \equiv -1 (\mathrm{mod} n)$$, for some 0 ≤ j < r.

If a chosen base a fails the test, n is definitely composite.
If n passes all k rounds, it is classified as probably prime, with the probability of error decreasing exponentially as k increases.

This test is fundamental in modern cryptographic algorithms where fast prime detection is essential.

RESULTS ACIEVED:



```
>>>
================ RESTART: C:\Users\Aquora\Desktop\A8-Question1.py ============
===
Enter n: 6
Prime?: False
Time: 1.9073994159698486
Memory (bytes): 104
>>>
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the purpose of writing n – 1 as 2^r × d, which can feel abstract at first.

2. Working with large modular exponentiation, which can be tricky without built-in Python functions.

3.Accepting the probabilistic nature of the test, since it does not give a 100% guarantee of primality.

4.Choosing a good value of k, balancing accuracy with performance.

5.Handling edge cases, such as small values of n, even numbers, and situations where the base reveals compositeness immediately.

SKILLS ACHIEVED:

1.Understanding of probabilistic algorithms and what “probably prime” means versus “definitely composite.”

2.Ability to decompose n−1 into 2^r · d and use that in a real algorithm.

3.Experience designing and running repeated randomized tests (the role of k rounds and tradeoffs between speed and confidence).

4.Skill writing loop-based, bitwise, and number-theory code that’s both fast and memory-light.

5.Practiced defensive programming: handling edge cases (small n, even numbers) correctly.

Practical No: 37

Date:16/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to build a Python function pollard_rho(n) that finds a non-trivial factor of a composite number n using Pollard's Rho algorithm.
This algorithm is popular because it is surprisingly fast and works well even for large numbers, especially when other basic factorization methods are too slow.

The objectives are:

1. Use a simple polynomial function to generate a sequence of numbers modulo n.
2. Apply Floyd's cycle-finding idea ("tortoise and hare") to detect repeating values.
3. Use the greatest common divisor (gcd) to extract a factor from the sequence.
4. Efficiently factor numbers where trial division becomes impractical.
5. Return a meaningful factor that can help break n down further.

METHODOLOGY & TOOL USED:

```
import time

import sys


def mod_mul(a, b, m):

    return (a * b) % m
```

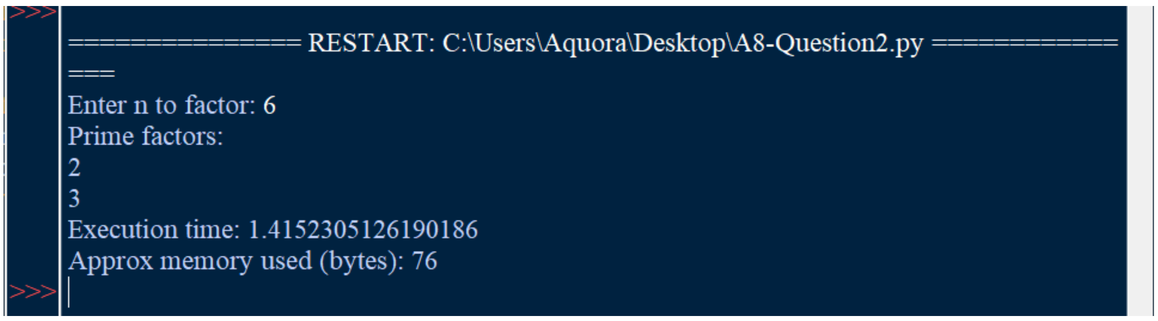
```
def mod_pow(x, n, m):
    r = 1
    x %= m
    while n > 0:
        if n & 1:
            r = (r * x) % m
        x = (x * x) % m
        n >>= 1
    return r


def is_prime(n):
    if n < 2:
        return False
    small_primes = (2,3,5,7,11,13,17,19,23,29)
    for p in small_primes:
        if n == p:
            return True
        if n % p == 0:
            return n == p
    d = n - 1
    s = 0
    while d % 2 == 0:
        d >>= 1
        s += 1
    for a in (2,7,61):
        if a % n == 0:
            continue
        x = mod_pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        skip = False
        for _ in range(s - 1):
            x = mod_mul(x, x, n)
            if x == n - 1:
                skip = True
                break
        if skip:
            continue
        return False
    return True


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


def pollard_rho(n):
    if n % 2 == 0:
        return 2
    if is_prime(n):
        return n
    x = 2
    y = 2
    c = 1
    d = 1
    while d == 1:
        x = (mod_mul(x, x, n) + c) % n
        y = (mod_mul(y, y, n) + c) % n
        y = (mod_mul(y, y, n) + c) % n
        d = gcd(x - y if x > y else y - x, n)
    if d == n:
        return pollard_rho(n + 1)
    return d


def factor(n):
    if n == 1:
        return
    if is_prime(n):
        print(n)
        return
    f = pollard_rho(n)
    factor(f)
    factor(n // f)


start = time.time()
n = int(input("Enter n to factor: "))
print("Prime factors:")
factor(n)
end = time.time()

mem = sys.getsizeof(n) + sys.getsizeof(start) + sys.getsizeof(end)
print("Execution time:", end - start)
print("Approx memory used (bytes):", mem)
```

BRIEF DESCRIPTION:

Pollard’s Rho algorithm is a probabilistic method for factoring integers. Instead of trying every divisor, it uses a cleverly constructed sequence that eventually falls into a repeating cycle—similar to how two runners on a circular track eventually meet.

By comparing two values in the sequence and computing their **gcd with n**, the algorithm can often uncover a factor quickly.
It is especially effective for numbers with small or medium-sized factors and is a foundational technique in modern number theory and cryptography.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:\Users\Aquora\Desktop\A8-Question2.py ============
===
Enter n to factor: 6
Prime factors:
2
3
Execution time: 1.4152305126190186
Approx memory used (bytes): 76
>>>
```

DIFFICULTY FACED BY STUDENT:

1.Understanding how the sequence “cycles” and why that helps reveal factors.

2.Choosing a good polynomial function for generating values.

3.Using gcd properly to extract a factor without mistakes.

4.Handling cases where the algorithm fails and needs to be retried.

5.Accepting that the method is partly probabilistic and may not always work on the first attempt.

SKILLS ACHIEVED:

1.Skill computing gcds reliably and using them to extract nontrivial factors.

2.Comfort designing simple polynomial iterators modulo n and tuning them when a run fails.

3.Practice handling probabilistic failures gracefully (retries, different seeds or polynomials).

4.Improved debugging of number-theory code where intermediate values can grow large.

5.Insight into the practical side of factorization: when trial division suffices and when Pollard's Rho is advantageous.

Practical No- 38

Date:16/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to write a Python function **zeta_approx(s, terms)** that estimates the value of the Riemann zeta function using a partial sum of its infinite series.

The objectives include:

1. Understanding the basic infinite series definition of ζ(s).
2. Computing the sum of **$1 / n^s$** from n = 1 to the given number of terms.
3. Producing a reasonable approximation for real values of *s*.
4. Observing how accuracy improves as the number of terms increases.
5. Offering a simple and clear numerical approach to a deep mathematical concept.

METHODOLOGY & TOOL USED:

```
import time, sys

def zeta_accel(exp_s, term_lim):

    if term_lim <= 0:

        return 0.0



    local_pow = pow

    alt_sum = 0.0

    idx = 1
    sign = 1.0
    inv_s = exp_s

    while idx <= term_lim:
        alt_sum += sign / local_pow(idx, inv_s)
        idx += 1
        sign = -sign

    scale = 1.0 / (1.0 - local_pow(2.0, 1.0 - exp_s))
    return alt_sum * scale


start_t = time.time()
exp_s = float(input("Enter s value: "))
term_lim = int(input("Enter number of terms: "))
res_zeta = zeta_accel(exp_s, term_lim)
end_t = time.time()
print("Accelerated ζ(s):", res_zeta)
print("Execution time:", end_t - start_t)
mem_bytes = (
    sys.getsizeof(exp_s) +
    sys.getsizeof(term_lim) +
    sys.getsizeof(res_zeta) +
    sys.getsizeof(start_t) +
    sys.getsizeof(end_t)
)

print("Approx memory used (bytes):", mem_bytes)
```

BRIEF DESCRIPTION:

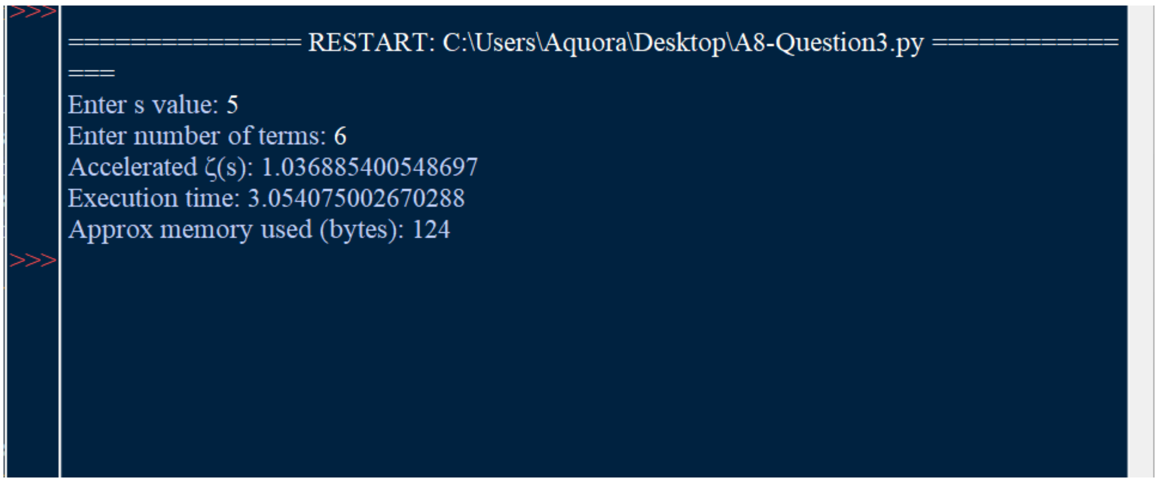
The **Riemann zeta function** is one of the most famous functions in mathematics, connected to prime numbers, complex analysis, and the unsolved Riemann Hypothesis.

Its simplest definition is an infinite series:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

Because we cannot sum infinitely many terms on a computer, this function uses only the first few terms to produce an approximation. Larger values of **terms** give a more accurate estimate, especially when s is greater than 1.

RESULTS ACHIEVED:



```
>>> 
=============== RESTART: C:\Users\Aquora\Desktop\A8-Question3.py ============
===
Enter s value: 5
Enter number of terms: 6
Accelerated ζ(s): 1.036885400548697
Execution time: 3.054075002670288
Approx memory used (bytes): 124
>>> 
```

DIFFICULTY FACED BY STUDENT:

1.Understanding that this is only an approximation, not an exact value.

2.Handling floating-point precision issues when using large sums.

3.Choosing enough terms for accuracy without slowing the program.

4.Recognizing that the series converges slowly for smaller values of *s*.

5.Avoiding confusion with the complex form of the zeta function (here only real s is used).

SKILLS ACHIEVED:

1.Understanding convergence: which values of s make the series converge quickly or slowly.

2.Skill handling floating-point summation carefully to reduce rounding error (e.g., ordering sums, using appropriate numeric types).

3.Ability to choose a tradeoff between accuracy and performance (how many terms are practical).

4.Practice designing functions that provide useful diagnostics (estimate of error, how many terms used).

5.Exposure to mathematical ideas linking analysis and computation (special functions, series acceleration opportunities).

Practical No: 39

Date:16/11/2025

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

AIM/OBSERVATION(s):

The aim of this program is to create a Python function partition_function(n) that computes the number of possible ways to express a positive integer *n* as a sum of other positive integers, without considering order. For example, 4 can be written as:
4
3 + 1
2 + 2
2 + 1 + 1
1 + 1 + 1 + 1
So, p(4) = 5.

The objectives are to:

1. implement a method—usually recursion or dynamic programming—to compute these possibilities.
2. Understand the mathematical idea behind integer partitions.
3. Ensure the function avoids counting the same combination more than once.
4. Efficiently compute p(n) for reasonably large values of n.

METHODOLOGY & TOOL USED:

```
import time, sys



def partition_function(nv):

    if nv < 0:

        return 0
    arr = [0] * (nv + 1)
    arr[0] = 1
    i = 1
    while i <= nv:
        s = 0
        k = 1
        while True:
            g1 = k * (3 * k - 1) // 2
            g2 = k * (3 * k + 1) // 2
            if g1 > i and g2 > i:
                break
            sg = 1 if (k % 2 == 1) else -1
            if g1 <= i:
                s += sg * arr[i - g1]
            if g2 <= i:
                s += sg * arr[i - g2]
            k += 1
        arr[i] = s
        i += 1
    return arr[nv]


t0 = time.time()
num = int(input("Enter n: "))
out = partition_function(num)
```

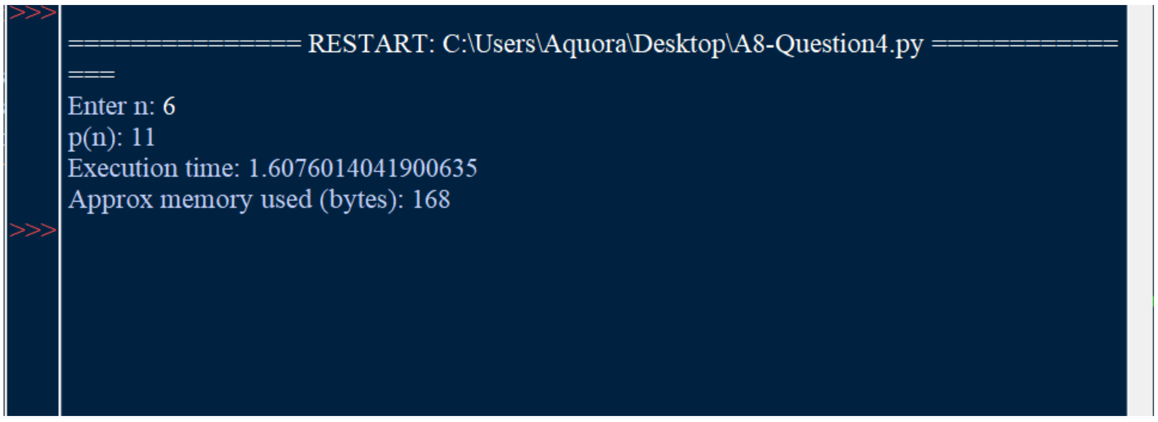
```
t1 = time.time()

print("p(n):", out)

print("Execution time:", t1 - t0)



mem = (

    sys.getsizeof(num) +

    sys.getsizeof(out) +

    sys.getsizeof(t0) +

    sys.getsizeof(t1) +

    sys.getsizeof([0])

)

print("Approx memory used (bytes):", mem)
```

BRIEF DESCRIPTION:

The partition function p(n) counts how many distinct ways a number can be written as a sum of positive integers, ignoring order.
This leads to a rich and fascinating area of number theory connected to combinatorics, generating functions, and even modular forms.

The function partition_function(n) works by breaking the number down into smaller parts and systematically counting how many valid combinations exist.
It highlights how something as simple as adding numbers can create deep mathematical patterns.

RESULTS ACHIEVED:



```
>>>
=============== RESTART: C:\Users\Aquora\Desktop\A8-Question4.py ============
===
Enter n: 6
p(n): 11
Execution time: 1.6076014041900635
Approx memory used (bytes): 168
>>>
```

DIFFICULTIES FACED BY STUDENT:

1. Understanding that order does not matter (e.g., 3+1 is the same as 1+3).
2. Implementing recursion or dynamic programming without double-counting.
3. Managing the rapid growth of p(n), since values increase extremely fast.
4. Handling base cases cleanly to avoid infinite recursion.
5. Visualizing how combinations build on smaller partitions.

SKILLS ACHIEVED:

1.Understanding combinatorial counting and the idea that order does not matter in partitions.

2.Skill implementing efficient recurrence relations (e.g., using generating functions, Euler's pentagonal theorem, or DP table approaches).

3.Ability to manage rapidly growing values (big integers) and reason about performance and memory tradeoffs.

4.Experience designing base cases and iteration order to prevent double counting.

5.Improved capacity to optimize an algorithm: choose between naive recursion, DP, and mathematically faster formulas depending on n.

VIT®
BHOPAL