Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BCE10884

Name of Student : DEEPTASUNDAR MOHANTY

Course Name :Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name :SCAI

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Euler’s Totient function		
2	Mobius function		
3	Sum of divisors		
4	Prime numbers count		
5	Legendre symbol		
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

VIT BHOPAL

Practical No: 1

Date: ___________

TITLE: **Euler's Totient function**

AIM/OBJECTIVE(s): To write a function called euler_phi(n) that calculates Euler's Totient Function, φ(n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which ged(n, k) = 1).

METHODOLOGY & TOOL USED: tool used: VS CODE

The algorithm begins by initializing result = n, then iteratively checks possible primes from 2 up to √n. Whenever a prime p divides n, the algorithm reduces result by result / p and continues dividing n by p to remove all multiples of that factor. If, after the loop, the remaining n is greater than 1, it is treated as a prime factor and applied to the formula. The final value of result is φ(n).

BRIEF DESCRIPTION: The function euler_phi(n) calculates how many numbers from 1 to n are coprime with n, meaning they share no common divisor larger than 1. This count is important in number theory, especially in modular arithmetic and cryptography. The function works by finding the prime factors of n and applying Euler's formula to efficiently determine how many integers remain relatively prime to it. This avoids the slower method of checking every number individually.

RESULTS ACHIEVED:

```
================================================================
Enter a positive integer: 8
number of integers up to 8 that are coprime with 8 are 4

--- Performance Metrics ---
Time taken (excluding input): 0.011667 seconds
Memory utilized (Final RSS): 26857472 bytes
```

DIFFICULTY FACED BY STUDENT: The student initially struggled with understanding why factoring n is essential and how Euler's product formula arises from the principle of excluding multiples of prime factors. Distinguishing between repeated prime factors and distinct prime factors caused confusion at first, as the formula only uses each prime once. Another challenge was correctly handling the final leftover value of n after the loop, which represents a prime factor larger than $\sqrt{n}$. Ensuring efficiency and avoiding unnecessary gcd computations also required thoughtful implementation.

CONCLUSION: Completing the euler_phi(n) function gave the student a solid grasp of how arithmetic functions relate to prime factorizations. It demonstrated that mathematical formulas often provide far more efficient algorithms than brute-force checking. Despite initial difficulties with understanding the theory and handling edge cases, the student successfully built a fast and accurate implementation of Euler's Totient Function, gaining confidence in both number theory concepts and algorithmic problem-solving.

Practical No: 2

Date: ___________

TITLE: Mobius function

AIM/OBJECTIVE(s): To 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.

METHODOLOGY & TOOL USED: tool used: VS CODE

To implement the mobius(n) function, the student used an approach based on prime factorization and square-free checking. The algorithm starts by iterating through integers from 2 up to √n and checking whether each one divides n. If a divisor appears twice (i.e., n is divisible by p^2), the function immediately returns 0 because the number is not square-free. If the divisor appears exactly once, it is counted as one prime factor. After processing all numbers up to √n, if the remaining value of n is greater than 1, it is an additional prime factor. Finally, if the count of distinct prime factors is even, the function returns 1; if odd, it returns –1. This method efficiently checks all conditions of the Möbius function.

BRIEF DESCRIPTION: The function mobius(n) determines whether a number is square-free and how many distinct prime factors it has, then returns a value based on these properties. It outputs 1 for square-free numbers with an even number of prime factors, –1 for square-free numbers with an odd number of prime factors, and 0 if the number contains any repeated prime factor. This function is important in number theory, particularly in multiplicative functions and the study of the Möbius inversion formula.

RESULTS ACHIEVED:

```
================================================================================
Enter a positive integer: 4
The value of mu(4) is: 0

--- Performance Metrics ---
Time taken (excluding input): 0.011485 seconds
Memory utilized (Final RSS): 26746880 bytes
>>> |
```

DIFFICULTY FACED BY STUDENT: The student initially found it challenging to distinguish between detecting prime factors and detecting *squared* prime factors, especially when implementing efficient division checks. It was also confusing at first to understand why the Möbius function only cares about *distinct* primes and not their multiplicities beyond one. Managing the loop boundaries up to √n and correctly updating the remaining value of n required careful attention. The student also struggled briefly with the conceptual reasoning behind why square-free numbers behave differently in the output of μ(n).

CONCLUSION: Through implementing the mobius(n) function, the student gained a deeper understanding of multiplicative arithmetic functions and the significance of square-free numbers in number theory. The task strengthened the student's comfort with prime factorization techniques and conditional logic based on mathematical definitions. Despite the initial confusion around detecting squared factors and counting distinct primes, the student ultimately produced a correct and efficient implementation, reinforcing both theoretical knowledge and programming skills.

Practical No: 3

Date: ___________

TITLE: SUM OF DIVISORS

AIM/OBJECTIVE(s): To 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).

METHODOLOGY & TOOL USED: tool used:VS CODE

To implement the divisor_sum(n) function, the student used an efficient method based on iterating only up to √n rather than checking every number up to n. For each integer i from 1 to √n, the algorithm checks whether i divides n. If it does, both i and its complementary divisor n/i are added to the running total, unless they are the same (in the case of a perfect square). This allows the function to compute the sum of all positive divisors in O(√n) time. Proper handling of divisor pairs ensures accuracy while significantly reducing computation.

BRIEF DESCRIPTION: The function divisor_sum(n) returns the sum of all positive divisors of a number, including both 1 and n. Instead of checking every possible divisor, the function leverages the fact that divisors come in pairs around √n, making the computation much faster. This sum, often denoted σ(n), is widely used in number theory, especially when studying perfect numbers, abundant numbers, and multiplicative functions.

RESULTS ACHIEVED:

```
... Enter a positive integer :12
    The sum of the divisors of 12 is 28
```

DIFFICULTY FACED BY STUDENT: The student initially struggled with understanding why checking only up to √n is sufficient and how divisor pairs work. Another challenge was avoiding double-counting in cases where n is a perfect square, since the square root divisor should only be added once. Implementing the logic cleanly without unnecessary computations required careful thought. Additionally, understanding the broader meaning of σ(n) in number theory helped clarify why the function must include both 1 and n as divisors.

CONCLUSION: Implementing the divisor_sum(n) function allowed the student to appreciate how mathematical insights can significantly optimize algorithms. Despite early difficulties with divisor logic and edge cases involving perfect squares, the student successfully produced an efficient and correct implementation. The task deepened their understanding of divisor functions, computational complexity, and the practical use of mathematical structure in programming.

Practical No: 4

Date: ___________

TITLE: Prime-counting function

AIM/OBJECTIVE(s):To 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.

METHODOLOGY & TOOL USED: tool used: VS CODE

To implement the prime_pi(n) function, the student used the Sieve of Eratosthenes, an efficient method for identifying all prime numbers up to n. The algorithm begins by creating a boolean list initialized to True, representing potential primality for each number from 0 to n. Starting from 2, the sieve iteratively marks all multiples of each prime as False, ensuring only primes remain marked True. Once the sieve completes, the function counts how many entries from 2 to n are still marked True and returns that count as the approximation of π(n). This approach efficiently computes prime counts in O(n log log n) time.

BRIEF DESCRIPTION: The function prime_pi(n) returns the number of prime numbers less than or equal to n. Instead of checking each number individually for primality, the function uses the Sieve of Eratosthenes to eliminate composite numbers in bulk. After the sieve finishes, a simple count of remaining prime entries gives the value of π(n). This method is widely used because it balances accuracy with excellent computational efficiency, making prime-counting practical even for large values of n.

RESULTS ACHIEVED:

```
=================================================================
Enter a positive integer: 6
Number of prime numbers <= 6 are 3

--- Performance Metrics ---
Time taken (excluding input): 0.012268 seconds
Memory utilized (Final RSS): 26501120 bytes
>>>
```

DIFFICULTY FACED BY STUDENT: The student initially struggled with understanding how the sieve logically eliminates composite numbers without missing any primes. Implementing the sieve efficiently—such as starting the marking process at p^2 and avoiding unnecessary iterations—also required careful attention. Managing memory for large n values and ensuring boundary conditions were correct posed additional challenges. Despite these difficulties, the student gradually understood how the sieve's structure ensures both speed and accuracy.

CONCLUSION: By building the prime_pi(n) function, the student gained strong insight into prime-generation algorithms and the elegance of the Sieve of Eratosthenes. The task demonstrated how mathematical patterns can be converted into highly efficient computational methods. Although the student faced initial confusion regarding sieve mechanics and optimization, they ultimately produced a correct and fast implementation, deepening their understanding of primes and algorithmic number theory.

Practical No: 5

Date: ___________

TITLE: Legendre function

AIM/OBJECTIVE(s): To create the legendre function.

METHODOLOGY & TOOL USED: tool used: VS CODE

The method computes a^((p−1)/2)mod p using Python's built-in modular exponentiation (pow(a, (p-1)//2, p)), which efficiently handles large numbers. If the result is 1, the function returns 1 (quadratic residue); if it is p−1p-1p−1, the function returns −1 (non-residue); and if a % p == 0, it returns 0. This approach is mathematically sound and computationally efficient.

BRIEF DESCRIPTION: The Legendre function calculates the Legendre symbol (a/p)(a/p)(a/p), which determines whether a number a is a quadratic residue modulo an odd prime p. The output is 1 if a has a square root modulo p, −1 if it does not, and 0 if a is divisible by p. This concept is important in number theory, especially in quadratic reciprocity, cryptographic algorithms, and modular arithmetic. The function provides a quick way to classify numbers using modular exponentiation.

RESULTS ACHIEVED:

```
Enter the value for 'a': 1
Enter the value for 'p': 5
a/p=1

--- Performance Metrics ---
Time taken (excluding input): 0.008842 seconds
Memory utilized (Final RSS): 26906624 bytes
>>>
```

DIFFICULTY FACED BY STUDENT: The student initially struggled to understand what a “quadratic residue” means and how Euler’s Criterion converts a theoretical definition into a usable algorithm. Another challenge was interpreting the modular exponentiation result: realizing that a value of p−1 corresponds to −1 was not immediately intuitive. The student also had to ensure that the input correctly handles cases where a is a multiple of p. Despite these conceptual hurdles, careful reading of number-theory principles allowed the student to implement the function accurately.

CONCLUSION: Writing the Legendre function helped the student connect theoretical number theory with practical programming. Although the underlying mathematics—quadratic residues, Euler’s Criterion, and modular exponentiation—seemed complex at first, the student ultimately gained confidence in converting these ideas into efficient code. By overcoming initial confusion and learning to interpret modular results correctly, the student successfully implemented a reliable Legendre symbol function, deepening their understanding of modular arithmetic and computational number theory.

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE10884
Name of Student	: Deeptasundar Mohanty
Course Name	:Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Factorial calculator	5-10-2025	
2	Palindrome calculator	5-10-2025	
3	Mean Calculator	5-10-2025	
4	Sum of digits calculator	5-10-2025	
5	Abundant Number Checker	5-10-2025	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: _5-10-2025__________

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

AIM/OBJECTIVE(s): To calculate the factorial of a non-negative integer.

METHODOLOGY & TOOL USED: Tool used: VS CODE

Language used: Python

I first checked if the number entered by the user is positive or not. Then, I ran a loop from 1 to the number and made a variable to store the product of all these numbers.

BRIEF DESCRIPTION: n this task, I wrote a Python program to calculate the factorial of a given number. The program takes an integer input from the user and computes its factorial using iterative or recursive methods. This exercise allowed me to translate a mathematical concept into a working program, reinforcing the connection between theory and practical implementation.

RESULTS ACHIEVED:

```
enter a non-negative integer: 5
FACTORIAL: 120
PS D:\chapter 4 ps>
```

DIFFICULTY FACED BY STUDENT: I faced difficulty in working with VS Code. I initially committed a lot of errors in the indentation.

SKILLS ACHIEVED:

By completing this program, I enhanced my problem-solving and logical thinking abilities. I gained hands-on experience with Python programming concepts such as loops, conditional statements, functions, and recursion. Additionally, I improved my understanding of handling user input and output, structuring code efficiently, and applying algorithmic reasoning to achieve accurate results.

The Program:

x=int(input('Enter a non-negative ihteger')

def factorial(x):

```
    f = 1
    if x < 0:
        print('wrong input')
    else:
        for i in range(1, x + 1):
            f *= i
        print( 'FACTORIAL:' , f)
factorial(x)
```

VIT BHOPAL

Practical No: 2

Date: ____5-10-25______

TITLE: Write a function to calculate the palindrome of a number.

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

METHODOLOGY & TOOL USED: Tool: VS CODE

Language: Python

I first took input from the user and stored it in another variable. Then, I separated the digits of the number using while loop. I used a variable to reverse the number and then put an 'if' statement to check if the reversed number is same as the original number or not.

BRIEF DESCRIPTION: In this task, I wrote a Python program to check whether a given number or string is a palindrome. The program accepts input from the user and determines if it reads the same forwards and backwards. This exercise helped me translate a logical condition into a functional program, reinforcing the connection between problem-solving and coding.

RESULTS ACHIEVED:

```
Enter an integer121
palindrome
```

```
Enter an integer: 21
not palindrome
```

DIFFICULTY FACED BY STUDENT: I forgot to call the function and this silly error wasted a lot of my time.

SKILLS ACHIEVED: By completing this program, I enhanced my analytical and logical thinking skills. I gained practical experience with Python concepts such as loops, conditional statements, string manipulation, and functions. Additionally, I improved my ability to handle user input and output, write structured and readable code, and implement algorithms to solve real-world problems.

Program:

```
n = int(input('Enter an integer: '))
```

```
def palindrome(n):
    k = n
    l = 0
    while k != 0:
        j = k % 10
        l = l * 10 + j
        k = k // 10

    if l == n:
        print("palindrome")
    else:
        print("not palindrome")

palindrome(n)
```

Practical No: 3

Date: __5-10-2025________

TITLE: To create a function mean_of_digits(n).

AIM/OBJECTIVE(s): To calculate the average of all the digits in a number.

METHODOLOGY & TOOL USED: Tool: VS CODE

Language: Python

I first created a loop to separate the digits of the number entered by the user. Then I used two variables, one for adding the digits and the other for counting the number of digits. By dividing the two, the mean value was obtained.

BRIEF DESCRIPTION: In this task, I wrote a Python program to calculate the average of all the digits in a given number. The program takes an integer input from the user, extracts each digit, sums them, and then divides by the total number of digits to find the average. This exercise helped me understand how to manipulate numbers at the digit level and translate mathematical operations into a working program.

RESULTS ACHIEVED:

```
enter an integer 1111
1.0
```

DIFFICULTY FACED BY STUDENT: I faced a difficulty in the integer division function and in making the program without using the count() function.

SKILLS ACHIEVED: By completing this program, I enhanced my problem-solving and logical thinking abilities. I gained hands-on experience with Python concepts such as loops, arithmetic operations, type conversion, and functions. Additionally, I improved my ability to handle user input and output, structure code efficiently, and apply algorithmic reasoning to solve numerical problems.

Program:

```
n= int(input("enter an integer "))
def mean_of_digits(n):
    c = 0
    m = 0
    k = n
    while k != 0:
        j = k % 10
        m += j
        c += 1
```

```
        k //= 10  # integer division

    avg = m / c
    print(avg)

mean_of_digits(n)
```

Practical No: 4

Date: _5-10-2025_________

TITLE:Write a program to make a function digital_root(n).

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

METHODOLOGY & TOOL USED: Tool: VS CODE

Language: Python

A nested while loop is used in the program. The first one to check if the sum is a single digit number and the second one to calculate the sum of the digits. The function will print the single- digit sum when both the loops are terminated.

BRIEF DESCRIPTION: In this task, I wrote a Python program to compute the digital root of a number, which involves repeatedly summing its digits until a single-digit result is obtained. The program accepts an integer input from the user, extracts and sums its digits iteratively or recursively, and continues this process until only one digit remains. This exercise helped me understand iterative and recursive logic, as well as how to apply mathematical concepts in programming.

RESULTS ACHIEVED:

```
Enter an integer: 34
7
```

DIFFICULTY FACED BY STUDENT: I faced a difficulty in the nested loops and conditional statements.

SKILLS ACHIEVED: By completing this program, I improved my problem-solving and logical thinking skills. I gained practical experience with Python concepts such as loops, recursion, functions, and arithmetic operations. Additionally, I enhanced my ability to handle user input and output, structure code effectively, and implement algorithms that solve repeated computational tasks efficiently.

Program:

```
n=int(input("enter an integer "))

def digital_root(n):

    while n >= 10:  # keep going until n is a single digit

        s = 0

        while n > 0:

            s += n % 10

            n //= 10

        n = s

    return n


digital_root(n)
```

Practical No: 5

Date: _5-10-2025_________

TITLE: Write function is_abundant(n).

AIM/OBJECTIVE(s): To write a function that returns true if the sum of proper divisors of n is greater than n.

METHODOLOGY & TOOL USED: Tool: VS CODE

Language: Python

In this question, I first took input from the user and stored it in another variable. Then, I used for loop to calculate the proper divisors of the number and add them. At the end, I used the if-else statements to check if the sum is greater than the number.

BRIEF DESCRIPTION: In this task, I wrote a Python function that determines whether a number is abundant, meaning the sum of its proper divisors (all divisors excluding the number itself) is greater than the number. The program takes an integer input from the user, finds all its proper divisors, sums them, and compares the sum with the original number. This exercise helped me understand divisor-based properties of numbers and how to translate mathematical logic into a functional program.

RESULTS ACHIEVED:

```
enter an integer 12
true
```

```
enter an integer 10
false
```

DIFFICULTY FACED BY STUDENT: I faced difficulty in understanding the question and implementing it.

SKILLS ACHIEVED: By completing this program, I enhanced my problem-solving and logical reasoning skills. I gained practical experience with Python concepts such as loops, conditional statements, functions, and arithmetic operations. Additionally, I improved my ability to handle user input and output, write structured and efficient code, and implement algorithms for number-theoretic computations.

Program:

```
n=int(input("enter an integer "))
def is_abundant(n):
    k = n
    s = 0
    for i in range(1, k):
        if k % i == 0:
            s += i

    if s > n:
        print("true")
    else:
        print("false")

is_abundant(n)
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BCE10884

Name of Student : Deeptasundar Mohanty

Course Name :Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name : SCAI

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Factorial Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.	02-11-2025	
2	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	
3	Write a function is_automorphic(n) that checks if a number's square ends with the number itself.	02-11-2025	
4	Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.	02-11-2025	
5	Write a function prime_factors(n) that returns the list of prime factors of a number.	02-11-2025	
6			
7			
8			
9			
10			
11			
12			
13			

14			
15			

Practical No: 1

Date: _02-11-2025__________

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): To check if the sum of proper divisors of a number is less than the number .

METHODOLOGY & TOOL USED: Tool used: IDLE

Language used: Python

First I define a function to check if a number is deficient. I calculate the sum of all proper divisors by only checking up to the number's square to save time. I add both the divisor and its paired counterpart. If the total sum is less than the original number, It returns True else false.

BRIEF DESCRIPTION: I wrote this code to check to see if a number is "deficient." It quickly finds all the numbers that divide the input (excluding the input itself) by only looking up to its square root. It adds those divisors up, and if that total is less than the original number, the function returns True else false.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/]
ion.py
Enter a positive integer: 8
True
```

DIFFICULTY FACED BY STUDENT: I faced difficulty in understanding the concept o deficient function.

SKILLS ACHIEVED:

By completing this program, I enhanced my problem-solving and logical thinking abilities. My biggest skill here is speed; I created a clever shortcut by only checking numbers up to the square root, which saves massive amounts of time when the input is large . I also figured out the trick to finding all divisor pairs correctly without mistakes, like avoiding double-counting on perfect squares.

The Program:

```
n = int(input("Enter a positive integer: "))

def is_deficient(n):

    s = 1

    i = 2

    while i * i <= n:

        if n % i == 0:

            s += i

            if i * i != n:

                s += n // i

        i += 1

    return s < n

result = is_deficient(n)

print(result)
```

Practical No: 2

Date: ____02-12-25______

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): To write a function that checks if a number is divisible by the sum of its digits.

METHODOLOGY & TOOL USED: Tool: IDLE

Language: Python

I first take input for a number. Then, I define a function to check if that number is a Harshad number. Inside the function, I save the original value, and then I systematically add up all the digits of the number by using the modulo operator to grab the last digit and division to remove it . Finally, It check if the original number is perfectly divisible by that total sum and prints the result.

BRIEF DESCRIPTION: In this task, I wrote a Python program to check whether a given number is divisible by the sum of its digits. The code saves the input number, then calculates the sum of its digits by iteratively using modulo 10 and floor division . Finally, it checks if the original number is perfectly divisible by the digit sum (a Harshad check) and prints a descriptive outcome.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python,
n.py
Enter a number: 4
4 is divisible by the sum of its digits
```

DIFFICULTY FACED BY STUDENT: Took me lot of time to correctly define this function

SKILLS ACHIEVED: I learned digit manipulation, specifically using the modulo operator to isolate digits and division to strip them from a number . I got better at iterative processing with a while loop, calculating a running sum. I also achieved skills in basic input handling and implementing the Harshad number mathematical rule for divisibility checking.

Program:

def is_harshad(n):

```
  n = int(n)
  original_n = n
  sum_of_digits = 0
  while n > 0:
    digit = n % 10
    sum_of_digits += digit
    n //= 10

  if (original_n % sum_of_digits == 0):
    print( original_n ,"is divisible by the sum of its digits")
  else:
     print( original_n ,"is not divisible by the sum of its digits")

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

is_harshad(num)
```

Practical No: 3

Date: __2-11-2025________

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

AIM/OBJECTIVE(s): To check if a number's square ends with the number itself.

METHODOLOGY & TOOL USED: Tool: IDLE

Language: Python

I ask the user for a positive number and check for errors. I then find its square. My key step is counting the digits to set the correct power-of-ten modulo base . I use this base to precisely extract the trailing digits from the square. Finally, It compares the extracted digits to your original number and print whether they match.

BRIEF DESCRIPTION: The function first calculates the square of the input number. It determines the number of digits to set the correct power-of-ten modulo base . It then uses this base to extract the exact number of trailing digits from the square. Finally, it compares these extracted digits to the original number to check if they match, defining it as automorphic.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/I
ction.py
enter a number 7
7 is not an automorphic number
```

DIFFICULTY FACED BY STUDENT: I faced a difficulty in understanding the automorphic function.

SKILLS ACHIEVED: By completing this program, I enhanced my problem-solving and logical thinking abilities. I gained hands-on experience with Python concepts such as loops, arithmetic operations, type conversion, and functions. Additionally, I improved my ability to handle user input and output, structure code efficiently, and apply algorithmic reasoning to solve numerical problems.

Program:

```
def is_automorphic(n):
    sq= n**2
    n2=n
    a=0
    while (n!=0):
        a=a+1
        n=n/10

    last=sq%a
```

```
    if( last==n2):
        print( n2, "is an automorphic number")
    else:
        print( n2, "is not an automorphic number")

num=int(input("enter a number "))

is_automorphic(num)
```

Practical No: 4

Date: _2-11-2025_________

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

AIM/OBJECTIVE(s): To create a function that checks if a number is the product of two consecutive integers.

METHODOLOGY & TOOL USED: Tool: IDLE

Language: Python

I start by asking for the input and then ensure it's a valid positive number using error checks. I then perform an optimized loop that only checks numbers up to the square root of the input. Inside the loop, It tests if the current integer times the next integer equals the number . If It finds a match, It immediately return True.

BRIEF DESCRIPTION: I first check the user's input for validity to prevent crashes. Then, I optimize the check by only looping up to the square root of the number. Inside the loop, I test if the current number i multiplied by the next consecutive number i + 1 equals the input number . If it matches, I immediately return True, proving it is pronic.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Pr
.py
enter a number 5
5 is not a pronic number
```

DIFFICULTY FACED BY STUDENT: I faced a difficulty in the conditional statements.

SKILLS ACHIEVED: I developed skills in creating efficient code by implementing an optimized loop that only checks up to the square root of the number, saving computational time . I also learned robust input handling by using try/except to ensure the program accepts only valid positive integers and prevents unexpected crashes. Finally, I translated the pronic number mathematical rule into clean, functional code.

Program:

```
def is_pronic(n):
    a=0
    for i in range (1,n):
        if( i*(i+1)==n):
            a=1
            break
        else:
            a=0

    return a


num= int(input("enter a number "))
if (is_pronic(num)==1):
  print( num , "is a pronic number")
elif (is_pronic(num)==0):
  print( num , "is not a pronic number")
```

Practical No: 5

Date: _2-11-2025__________

TITLE: Write a function prime_factors(n) that returns the list of

prime factors of a number.

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

METHODOLOGY & TOOL USED: Tool: IDLE

Language: Python

I ask the user for a positive number and handle any errors. I then optimize the check by only looping up to the number's square root. Inside the loop, I test if the current integer times the next integer equals your number . If I find a match, I immediately confirm it is pronic by returning True.

BRIEF DESCRIPTION: This program efficiently finds all prime factors of a user-inputted number. It first isolates all factors of 2. Then, it iteratively checks odd numbers, starting from 3, only up to the square root of the number . If a factor is found, it is repeatedly divided out. The last remaining number, if greater than two, is added as the final prime factor, and the complete list is printed.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local
.py
Enter a positive integer: 9
[3, 3]
```

DIFFICULTY FACED BY STUDENT: I faced difficulty in understanding the question and implementing it.

SKILLS ACHIEVED: I got better at building super-fast math tools. My key skill is speed optimization by first finding all factors of 2 and then only checking odd numbers up to the square root . This saves massive computation time. I also learned to correctly handle the remaining number if it's a large prime itself, always returning a complete and accurate list of factors.

Program:

```
n = int(input("Enter a positive integer: "))

def prime_factors(n):

    factors = []


    while n % 2 == 0:

        factors += [2]

        n //= 2


    i = 3

    while i * i <= n:

        while n % i == 0:

            factors += [i]

            n //= i

        i += 2


    if n > 2:

        factors += [n]


    return factors


if n <= 1:
```

```
    print([])
else:
    print(prime_factors(n))
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BCE10884
Name of Student	: Deeptasundar Mohanty
Course Name	:Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	:SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Distinct prime numbers	9-11-2025	
2	Power of a number	9-11-2025	
3	Mersenne prime number	9-11-2025	
4	Twin prime numbers	9-11-2025	
5	Count divisors	9-11-2025	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: __9/11/25______

TITLE: Counting unique prime factors

AIM/OBJECTIVE(s): To create a function that returns how many unique prime factors a number has.

METHODOLOGY & TOOL USED: VS Code

We first divide the number by 2 repeatedly, then check for divisibility by all odd numbers up to its square root. Every time we find a divisor, we add it to a set to ensure uniqueness. Finally, we return the length of this set as the count of unique prime factors.

BRIEF DESCRIPTION: This Python program calculates how many unique prime factors a given integer has. It applies number theory concepts and efficient iteration techniques to ensure accurate results. By using a set, it avoids duplicate factor counting and simplifies the process. The algorithm is optimized for both small and large inputs and provides insight into the number's prime composition. It serves as a clear example of combining mathematical logic with programming efficiency.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313/
1.py
3
```

DIFFICULTY FACED BY STUDENT: Students often struggle to distinguish between regular factors and prime factors. Handling repeated factors and ensuring they aren't counted multiple times can be confusing. They may also find it challenging to determine when to stop checking for divisibility using the square root approach. Edge cases like n = 1 or negative numbers often lead to logical errors. Debugging and optimizing the loop structure for efficiency can also pose difficulties during implementation.

SKILLS ACHIEVED: By completing this program, students strengthen their understanding of prime numbers and factorization. They gain experience in translating mathematical logic into Python code. The use of sets enhances their knowledge of data structures and uniqueness handling. They also improve their debugging and problem-solving abilities while learning to write clean, efficient code. This activity helps build logical thinking and confidence in algorithmic design.

Practical No: 2

Date: ___9/11/25____

TITLE: square of a prime number

AIM/OBJECTIVE(s): To create a function that checks if a number can be expressed as the square of a prime number or not.

METHODOLOGY & TOOL USED: VS CODE

The program first checks if the given number is a perfect square by finding its square root. If the square root is an integer, the program then verifies whether this root is a prime number. It uses a simple prime- checking loop or function to test divisibility. If both conditions are true, the number can be expressed as the square of a prime number.

BRIEF DESCRIPTION: This program determines whether a given number can be expressed as the square of a prime. It combines two key mathematical checks — verifying perfect squares and testing primality. The algorithm ensures accurate results with minimal computation. The output clearly indicates whether the number meets the given condition or not.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313/
1.py
Enter a number to check if it's a prime power: 6
6 is not a prime power.
memory usage :26775552 bytes

Execution Time: 2.458290 seconds
```

DIFFICULTY FACED BY STUDENT: Students may find it tricky to check both conditions correctly — especially identifying

integer square roots.
Implementing an efficient prime-checking method without errors can

also be challenging. Handling edge cases like small numbers or non- perfect squares often causes confusion. Ensuring logical accuracy between the two checks may require careful debugging.

SKILLS ACHIEVED: Students learn to integrate multiple mathematical concepts into a single algorithm. They strengthen their understanding of perfect squares, square roots, and prime numbers. This exercise improves their problem-solving and logical reasoning abilities. It also helps them gain confidence in writing efficient and structured Python code.

Practical No: 3

Date: ___9/11/25____

TITLE: Mersenne prime number

AIM/OBJECTIVE(s): To check if the 2r-1 of a number is prime or not.

METHODOLOGY & TOOL USED: VS CODE

The program first identifies whether a number can be written in the form 2p−12^p - 12p−1, where ppp is a prime number. It loops through possible values of ppp and checks if 2p−12^p - 12p−1 equals the given number. Once a match is found, the program verifies if the number itself is prime. If both conditions hold true, the number is classified as a Mersenne prime.

BRIEF DESCRIPTION: This Python program checks if a number is a Mersenne prime — a special kind of prime that fits the form 2p−12^p - 12p−1. It combines exponentiation, logical checks, and prime verification in one function. The output confirms whether the given number satisfies both mathematical conditions. The algorithm demonstrates an elegant link between exponents and prime numbers.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313/disti
1.py
Enter a prime number to check if it's a Mersenne prime: 8
2^8 - 1 is not a Mersenne prime.
```

DIFFICULTY FACED BY STUDENT:

Students may find it challenging to correctly express the condition 2p−1=n2^p - 1 = n2p−1=n. They often struggle with efficiently checking for primality, especially for large numbers. Understanding the mathematical relationship between ppp and 2p−12^p - 12p−1 can also be confusing at first. Logical errors may arise while combining both conditions into a single function.

SKILLS ACHIEVED: Students develop a deeper understanding of number theory and prime patterns. They strengthen their ability to translate mathematical formulas into working code. This task enhances their problem-solving, logical thinking, and debugging skills. It also helps them gain experience in handling mathematical algorithms efficiently in Python.

Practical No: 4

Date: ___9/11/25____

TITLE: Twin prime numbers

AIM/OBJECTIVE(s): To generate twin prime numbers upto a given number.

METHODOLOGY & TOOL USED: VS CODE

To find twin prime numbers till a given number, we use the Python programming language and its simple control structures like loops and conditionals. The methodology involves creating a function to check if a number is prime, then iterating through numbers up to the limit to find pairs of primes that differ by 2. Python's efficiency and readability make it ideal for implementing mathematical logic clearly and effectively.

BRIEF DESCRIPTION: The program defines a function that first checks primality by testing divisibility. Then, it identifies and prints all pairs of primes (p, p+2) up to the given number. Twin primes like (3, 5), (5, 7), and (11, 13) are displayed as output. This helps in understanding prime relationships and logical iteration in number theory problems.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313/distinct prime fac
1.py
The twin pairs are:[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61
1, 73), (101, 103), (107, 109)]
time taken to run this code: 2.1457672119140625e-06 seconds
memory usage: 184 KB
```

DIFFICULTY FACED BY STUDENT:

Students often face difficulty in writing an efficient prime-checking function and managing nested loops correctly. They might also struggle with optimizing the algorithm to avoid unnecessary calculations and ensuring they correctly handle edge cases like small numbers or upper limits.

SKILLS ACHIEVED: By completing this task, students improve their logical thinking, problem-solving, and understanding of loops and functions in Python. They also gain confidence in applying mathematical concepts programmatically and writing clean, structured, and optimized code.

Practical No: 5

Date: ____9/11/25_____

TITLE: Counting the number of divisors

AIM/OBJECTIVE(s): To count the number of positive divisors of a number.

METHODOLOGY & TOOL USED: VS CODE

To find the number of divisors of a given number, Python is used for its simplicity and efficiency in performing mathematical operations. The methodology involves iterating from 1 to the number itself (or up to its square root for optimization) and counting how many numbers divide it evenly using the modulus operator %. Python's looping and conditional constructs make this logic easy to implement.

BRIEF DESCRIPTION: The function takes an integer as input and checks each number to see if it divides the given number without leaving a remainder. Every time a divisor is found, the count increases. Finally, it returns the total number of positive divisors. This helps students understand factors and the use of iteration in real applications.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs
1.py
Enter a number to count its divisors: 6
Number of divisors of 6: 4
memory usage :26873856 bytes

Execution Time: 2.599454 seconds
```

DIFFICULTY FACED BY STUDENT: Students often struggle with optimizing the loop to run efficiently and handling perfect squares correctly to avoid double-counting divisors. They may also find it tricky to differentiate between divisors and prime factors in similar problems.

SKILLS ACHIEVED: Through this exercise, students develop logical reasoning, mathematical problem-solving, and coding proficiency in Python. They gain experience in using loops, conditionals, and return statements effectively while improving algorithmic thinking.

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BCE10884
Name of Student	: Deeptasundar Mohanty
Course Name	:Introduction to Problem Solving and Programming

Course Code	: CSE1021
School Name	:SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Sum of all proper divisors	16-1125	
2	Amicable number checker	16-1125	
3	Steps until a number's digits multiply to a single digit	16-1125	
4	number has more divisors than any smaller number checker	16-1125	
5	(base^exponent) % modulus calculator	16-1125	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: 16-11-25

TITLE: sum of all proper divisors

AIM/OBJECTIVE(s): To write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

METHODOLOGY & TOOL USED: IDLE

We first start by requesting a positive integer input, n. Next, it systematically checks every whole number, i, from 1 up to n-1. If i divides n without leaving a remainder (meaning i is a proper divisor), that number is added to a running total. Once all smaller numbers have been checked, the final accumulated sum of these divisors is returned and displayed as the aliquot sum of n.

BRIEF DESCRIPTION: This Python program defines the function aliquot_sum(n), which calculates the sum of all proper divisors of a positive integer n. It iterates from 1 up to n-1, checking if each number divides n exactly. All divisors found are added together. The program then asks the user for a number, calls the function, and prints the resulting aliquot sum, which is used to classify numbers as perfect, abundant, or deficient

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python
gnment.py
Enter a number: 6
The aliquot sum of 6 is: 6
```

DIFFICULTY FACED BY STUDENT: Starting the loop at zero caused ZeroDivisionError. I struggled to adhere strictly to the definition of proper divisors (excluding n). Finally, ignoring efficiency leads to slow execution for very large numbers n.

SKILLS ACHIEVED: By completing this program, demonstrating ability to structure reusable code. It requires strong conditional logic to correctly identify divisors using the modulo operator (%) and the ability to manage accumulator variables to calculate the running sum. Crucially, it shows an understanding of number theory by translating the definition of proper divisors into an effective algorithm.

PROGRAM:

```
def aliquot_sum(n):
    s = 0
    for i in range(1, n):
        if (n % i == 0):
            s = s + i
    return s

n = int(input("Enter a number: "))
print(f"The aliquot sum of {n} is: {aliquot_sum(n)}")
```

Practical No: 2

Date: 16-11-25

TITLE: amicable number checker

AIM/OBJECTIVE(s): To 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).

METHODOLOGY & TOOL USED: IDLE

This program determines if two input integers, a and b, are an amicable pair. The methodology involves two identical calculation steps: first, it iterates through all numbers less than a to find and sum its proper divisors (storing the result in s1); second, it does the same for b, storing the sum in s2. Finally, it checks the amicability condition: if s1 equals b and s2 equals a, the numbers are declared amicable.

BRIEF DESCRIPTION: This program defines the function are_amicable(a, b) to check if two positive integers, a and b, form an **amicable pair**. It separately calculates the **aliquot sum** (sum of proper divisors) for a (as s1) and for b (as s2). The numbers are deemed amicable if and only if the sum of a's proper divisors equals b (s1 = b) and the sum of b's proper divisors equals a (s2 = a). The script then prompts the user for two numbers and reports the result.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313
gnment.py
Enter a number: 8
Enter another number: 5
numbers are not amicable
```

DIFFICULTY FACED BY STUDENT: The main difficulties involve duplicating code for calculating both s1 and s2, which is inefficient and error-prone. I also incorrectly set the loop ranges, possibly including a or b and thus violating the proper divisors rule, andforget to enforce the crucial two-way check (s1=b AND s2=a) for true amicability

SKILLS ACHIEVED: The skills achieved include function definition with multiple arguments and algorithmic design to implement complex number theory concepts. It requires effective use of loops and the modulo operator to calculate the aliquot sum twice, demonstrating strong conditional logic to perform the strict, two-way comparison required to determine if the input numbers are a true amicable pair.

PROGRAM:

```
def are_amicable(a,b):
    s1 = 0
    for i in range(1, a):
        if (a % i == 0):
            s1 += i

    s2 = 0
    for j in range(1, b):
        if (b % j == 0):
            s2 += j

    if (s1 == b and s2 == a):
        print("numbers are amicable")
    else:
        print("numbers are not amicable")

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))
are_amicable(a, b)
```

Practical No: 3

Date: 16-11-25

TITLE: Steps until a number's digits multiply to a single digit

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

METHODOLOGY & TOOL USED: IDLE

The code calculates multiplicative persistence by using a while loop that continues as long as the number n is 10 or greater, incrementing a persistence counter (c) each time. Inside this loop, a nested while loop extracts the digits of n one by one using the modulo operator (% 10) and multiplies them together to find a new product (p). This product replaces the original n, and the process repeats until n is a single-digit number, at which point the final count (c) is returned.

BRIEF DESCRIPTION: This code calculates the multiplicative persistence of an input integer n. The main while loop runs as long as n is two or more digits, tracking the step count (c). Inside, a nested loop extracts and multiplies n's digits to get a new product, which replaces n. This process repeats until n is a single digit (less than 10), and the final count c is returned.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313
gnment.py
Enter a number: 4
0
```

DIFFICULTY FACED BY STUDENT:

Primarily struggle is with nested loop logic, specifically coordinating the inner loop (digit extraction) with the outer loop (persistence count). We often make errors using the modulo operator (% 10) and integer division (// 10) to correctly manipulate digits. Additionally, managing the variables n and temp_n without confusing the outer loop's termination condition is a frequent challenge

SKILLS ACHIEVED: The skills achieved include strong command of nested loop structures for iteration and control, demonstrating complex algorithmic design. It shows mastery of digital manipulation using the modulo operator (% 10) for digit extraction and integer division (// 10) for number reduction. Finally, it proves the ability to translate the abstract mathematical concept of multiplicative persistence into a working, iterative program.

PROGRAM:

```
def multiplicative_persistence(n):
    c = 0
    while n >= 10:
        p = 1
        temp_n = n
        while temp_n > 0:
            digit = temp_n % 10
            p *= digit
            temp_n //= 10
        n = p
        c += 1
    return c

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

Practical No: 4

Date: 16-11-25

TITLE: number has more divisors than any smaller number checker

AIM/OBJECTIVE(s):To Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.

METHODOLOGY & TOOL USED: IDLE

The code determines if a number n is highly composite by using a two-stage approach. First, the count_divisors function efficiently calculates the total number of divisors for any given number by checking only up to its square root. Second, the main function iterates through all integers i smaller than n and uses the helper function to compare their divisor counts. If n does not have strictly more divisors than every smaller number, it returns False.

BRIEF DESCRIPTION: The program checks if a number n is Highly Composite (HCN). It uses an optimized helper function, count_divisors, which efficiently finds the total number of divisors for any integer by checking only up to its square root. The main function, is_highly_composite, then iterates through all smaller numbers to confirm that n has strictly more divisors than every one of them.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/Python/Python313/\
gnment.py
Enter a positive integer to check for Highly Composite status: 3

3 is NOT a Highly Composite Number.
   (Divisor count: 2)
```

DIFFICULTY FACED BY STUDENT:

The primary difficulty lies in the divisor counting optimization, specifically implementing the complex logic of checking only up to the square root of n while correctly handling the divisor pairs and the perfect square case to avoid double counting. Furthermore, managing the computational expense is challenging, as the is_highly_composite function repeatedly calls the count_divisors function for every number up to n, leading to very slow execution for large inputs.

SKILLS ACHIEVED: The skills achieved include algorithmic optimization by implementing the efficient technique in count_divisors to calculate divisor counts quickly. This also showcases the ability to design modular code using helper functions. Furthermore, it involves complex iterative comparison logic and the accurate mathematical translation of the strict definition of a Highly Composite Number for verification.

PROGRAM:

```
def count_divisors(num):
    if num <= 0:
        return 0
    if num == 1:
        return 1

    count = 0
    i = 1
    while i * i <= num:
        if num % i == 0:
            if i * i == num:
                count += 1
            else:
                count += 2
        i += 1
    return count

def is_highly_composite(n):
    if n <= 0:
        return False
    if n == 1:
        return True

    divisors_of_n = count_divisors(n)

    for i in range(1, n):
        divisors_of_i = count_divisors(i)
        if divisors_of_i >= divisors_of_n:
            return False

    return True

number = int(input("Enter a positive integer: "))
```

```
if is_highly_composite(number):
    print(f"\n{number} IS highly composite.")
else:
    print(f"\n{number} is NOT highly composite.")

if number > 0:
    print(f"Divisors: {count_divisors(number)}")
```

Practical No: 5

Date: <u>16-11-25</u>

TITLE: Counting the number of divisors

AIM/OBJECTIVE(s): To Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (base^exponent) % modulus.

METHODOLOGY & TOOL USED: IDLE

The code calculates the modular exponentiation (base^{exponent}) \pmod{modulus} using the efficient Square-and-Multiply algorithm. It initially reduces the base modulo the modulus. It then employs a while loop that iterates through the exponent's binary bits. If the current bit is 1 (exponent is odd), it multiplies the result by the current base modulo the modulus. In every iteration, the base is squared modulo the modulus, and the exponent is halved, ensuring intermediate numbers remain small and preventing overflow.

BRIEF DESCRIPTION: This code implements the Modular Exponentiation algorithm, also known as Square-and-Multiply. It efficiently calculates the result of (base^{exponent}) \pmod{modulus}. It uses a while loop to process the exponent's binary form. By repeatedly squaring the base and multiplying it into the result only when the exponent is odd, it prevents intermediate numbers from becoming excessively large, which is vital for cryptographic applications.

RESULTS ACHIEVED:

```
= RESTART: C:/Users/CAAS/AppData/Local/Programs/
gnment.py
Enter the base: 4
Enter the exponent: 5
Enter the modulus: 2

Result: (4^5) mod 2 = 0
```

DIFFICULTY FACED BY STUDENT: The core challenge I see in this code is the Square-and-Multiply logic. I find it difficult to grasp how the algorithm uses the binary form of the exponent to efficiently control the multiplication. I have to be very careful to correctly apply the modulo operator in *both* the squaring of the base and the multiplication into the result, which I know is vital for me to prevent integer overflow and keep the modular arithmetic correct throughout the computation.

SKILLS ACHIEVED: Writing this code demonstrates advanced proficiency in algorithmic efficiency by correctly implementing the Square-and-Multiply method. This requires understanding bitwise logic and the ability to prevent integer overflow by strategically applying the modulo operator after every multiplication. This highly optimized approach is essential for large-number calculations

PROGRAM:

```
def mod_exp(base, exponent, modulus):

    if modulus == 1:
        return 0

    result = 1
    base %= modulus

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

        base = (base * base) % modulus
        exponent //= 2

    return result
 b = int(input("Enter the base: "))
e = int(input("Enter the exponent: "))
m = int(input("Enter the modulus: "))

if e < 0:
    print("This function does not handle negative exponents.")
else:
```

```
final_result = mod_exp(b, e, m)
print(f"\nResult: ({b}^{e}) mod {m} = {final_result}")
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BCE10884
Name of Student	:Deeptasundar Mohanty
Course Name	:Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	:SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Modular Multiplicative inverse	25-11-25	
2	Chinese remainder theorem	25-11-25	
3	Quadratic Residue	25-11-25	
4	Smallest positive integer	25-11-25	
5	Fibonacci prime check	25-11-25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: __25-11-25________

TITLE: Modular Multiplicative inverse

AIM/OBJECTIVE(s): to write a function Modular Multiplicative

Inverse mod_inverse(a, m) that finds the number x such

that (a * x) ≡ 1 mod m.

METHODOLOGY & TOOL USED: Tool used: VS CODE

To find the modular multiplicative inverse, the project used the **Extended Euclidean Algorithm**, a mathematical method that computes integers x and y such that ax+my=gcd(a,m). Since the inverse of a mod m exists only when a and m are coprime (i.e., gcd(a,m)=1), this algorithm helps determine both the gcd and the required coefficient x, which becomes the modular inverse after applying modulus m. The methodology involved taking user input, applying the extended algorithm step-by-step, extracting the coefficient corresponding to x, and adjusting it to a positive value within the range 0 to m−1. This approach ensured accuracy, efficiency, and compatibility with any valid pair of integers.

BRIEF DESCRIPTION:

This project implements a Python function **mod_inverse(a, m)** that returns a number x such that (a×x)≡1mod m. The program checks whether the inverse exists by calculating the greatest common divisor of a and m. If they are coprime, the Extended Euclidean Algorithm is used to compute the inverse; otherwise, the function returns that no modular inverse exists. The project demonstrates how mathematical theory can be translated into a practical algorithm, making it useful in cryptography, number theory problems, hashing, and secure communication protocols.

RESULTS ACHIEVED:

```python
# Test
start_time = time.time()
result = mod_inverse(3, 26)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024  # bytes
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 9, Runtime: 0.000059s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty faced during the project was understanding how the Extended Euclidean Algorithm works behind the scenes, especially the back-substitution step where coefficients are traced to extract the value of x. Initially, it was confusing to see how a series of recursive gcd calculations eventually leads to the inverse. Another challenge was ensuring that the final answer is always positive because the algorithm sometimes returns negative values. Handling edge cases—such as when a and m are not coprime—also required careful checks to prevent incorrect outputs. Debugging these issues helped strengthen both algorithmic understanding and Python implementation skills.

CONCLUSION: The project successfully demonstrates how the modular multiplicative inverse can be computed using an efficient and reliable method. By implementing the Extended Euclidean Algorithm in Python, it becomes clear how powerful mathematical concepts can be applied directly in programming. The challenges faced during the project improved problem-solving skills, logical thinking, and understanding of number theory. Overall, the work highlights the importance of modular arithmetic in real-world applications like cryptography and shows how a simple function can contribute to more complex and secure systems.

Practical No: 2

Date: ____25-11-25______

TITLE: Chinese remainder theorem

AIM/OBJECTIVE(s): to write a function chinese Remainder Theorem

Solver crt(remainders, moduli) that solves a system of

congruences $x \equiv ri \bmod mi$.

METHODOLOGY & TOOL USED: tool used: VS CODE

To solve a system of congruences using the Chinese Remainder Theorem (CRT), the project used a step-by-step constructive approach based on modular arithmetic. First, all moduli were checked to ensure they are pairwise coprime, which is required for a unique solution. The algorithm then computed the product $M=m1\times m2\times\cdots\times mn$ and, for each congruence, calculated $Mi=M/mi$. Using a modular multiplicative inverse function, the inverse of Mi modulo mi was found, and the partial solution $ri\times Mi\times Mi-1$ was accumulated. The final result was obtained by taking this sum modulo M, giving the smallest non-negative solution to the entire system. This structured methodology ensured correctness, efficiency, and scalability.

BRIEF DESCRIPTION: This project implements a Python function **crt(remainders, moduli)** that solves simultaneous congruences of the form $x\equiv ri \bmod mi$. The goal of the function is to compute a single integer xxx that satisfies every congruence in the system. The program first validates the inputs and then applies the Chinese Remainder Theorem to combine all congruences into one final solution. The function is useful in number theory, cryptography (like RSA), computing with large integers, and problems requiring modular synchronization. It demonstrates how theoretical mathematical principles can be translated into a practical and reusable computational tool.

RESULTS ACHIEVED:

```
# Test
start_time = time.time()
result = crt([2, 3], [3, 5])
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")

Result: 11, Runtime: 0.000077s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The major difficulty faced while working on this project was understanding the logic behind breaking down the large modulus into smaller components and reconstructing the final solution. It was initially challenging to grasp why the product M, the partial products Mi, and the modular inverses together guarantee that each term contributes correctly to the final answer. Debugging errors due to incorrect inverses or mismatched list lengths was also tricky. Ensuring that all moduli were pairwise coprime and handling the situation when they were not added additional complexity. Overall, the process required careful attention to both mathematical reasoning and implementation details.

CONCLUSION: The project successfully demonstrates the practical implementation of the Chinese Remainder Theorem and shows how multiple modular equations can be solved through a single systematic algorithm. By coding the CRT solver, the student gained a deeper understanding of modular arithmetic, inverse calculation, and the structure of number-theoretic algorithms. Although the problem initially seemed complex, breaking it into smaller mathematical steps made it manageable and strengthened both logic and programming skills. The final function is efficient, accurate, and highlights the power of classical mathematics in solving modern computational problems.

Practical No: 3

Date: ____25-11-25_______

TITLE: Quadratic Residue

AIM/OBJECTIVE(s): To write a function Quadratic Residue

Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p

has a solution.

METHODOLOGY & TOOL USED: tool used: VS CODE

To determine whether the congruence $x^2 \equiv a \bmod p$ has a solution, the project used **Euler's Criterion**, which provides a fast and reliable method for checking quadratic residues when p is an odd prime. According to Euler's Criterion, a is a quadratic residue modulo p if and only if a^(p−1)/2≡1 mod . The methodology involved validating that p is prime, taking input a, computing the exponentiation efficiently using Python's modular power function, and checking the result against 1 or -1 mod p. This approach ensured correctness, minimized computational time, and avoided brute-force checking of all possible values of x.

BRIEF DESCRIPTION: This project implements a Python function **is_quadratic_residue(a, p)** that checks whether a solution exists for the congruence x2≡a mod p. The function uses Euler's Criterion to determine if a is a quadratic residue modulo the prime number p. If the result of the criterion equals 1, the function returns true, indicating a solution exists; otherwise, it returns false. This concept is fundamental in number theory and has applications in cryptography, primality testing, and algorithms involving modular arithmetic. The project demonstrates how theoretical mathematical properties can be directly applied in computational methods.

RESULTS ACHIEVED:

```
# Test
start_time = time.time()
result = is_quadratic_residue(2, 7)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000056s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The primary difficulty faced during this project was understanding why Euler's Criterion works and how it connects to Fermat's Little Theorem and the structure of multiplicative groups modulo p. Initially, interpreting the meaning of quadratic residues and relating them to modular exponentiation was confusing. Another challenge was ensuring the program handled edge cases correctly, such as values of a greater than p or negative inputs, as well as confirming that p is indeed an odd prime. Implementing modular exponentiation correctly and verifying results without brute force required careful reasoning and testing.

CONCLUSION: The project effectively demonstrates how to determine quadratic residues using an elegant and efficient mathematical tool. Implementing Euler's Criterion in a Python function made the concept clearer and showed how number-theoretic ideas translate into algorithmic solutions. The challenges faced during the process improved the understanding of modular arithmetic, exponentiation, and theoretical concepts behind quadratic residues. Overall, the project strengthened problem-solving skills and highlighted the importance of number theory in modern computational and cryptographic applications.

Practical No: 4

Date: ___25-10-25________

TITLE: Smallest positive integer

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

METHODOLOGY & TOOL USED: tool used: VS CODE

To find the order of an integer a modulo n, the project followed a systematic method based on properties of modular arithmetic. First, the function checked whether gcd(a,n)=1, since the multiplicative order exists only when a is coprime with n. Once validated, the algorithm iteratively computed a^k mod n for increasing values of k, starting from 1, until the expression became congruent to 1. Efficient modular exponentiation was used to ensure the process remained fast even for larger values, and the loop stopped immediately upon finding the smallest such k. This approach ensured correctness by directly following the definition of multiplicative order while keeping the implementation simple and computationally manageable.

BRIEF DESCRIPTION: This project implements a Python function **order_mod(a, n)** that calculates the smallest positive integer k for which $a^k \equiv 1 \bmod n$. The function serves as a tool to study the structure of modular arithmetic, especially in multiplicative groups, and is useful in applications such as cryptography, cyclic groups, primitive roots, and number-theoretic algorithms. By repeatedly evaluating powers of a modulo n until the congruence equals 1, the function determines the exact order of a modulo n, helping to understand how elements behave within modular systems.

RESULTS ACHIEVED:

```
start_time = time.time()
result = order_mod(2, 7)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 3, Runtime: 0.000056s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: One of the main difficulties encountered during the project was understanding why the multiplicative order only exists when a and n are coprime, and how that connects to the underlying group structure. Another challenge was ensuring that the loop terminates correctly for all valid inputs while remaining efficient, especially when dealing with large exponents or numbers with many possible divisors. Debugging errors caused by incorrect modular exponentiation or not handling non-coprime inputs properly also required careful attention. These challenges helped improve understanding of modular arithmetic, exponentiation, and algorithmic optimization.

CONCLUSION: The project successfully demonstrates how to compute the multiplicative order of a number modulo n using a direct and intuitive method rooted in fundamental number theory. Implementing the function deepened understanding of cyclic behaviour in modular systems and highlighted the importance of concepts such as gcd, modular exponentiation, and group theory. Overcoming the challenges strengthened both mathematical intuition and coding skills. Overall, the function is accurate, practical, and provides valuable insight into the behavior of numbers under modular arithmetic, making it useful in various theoretical and cryptographic applications.

Practical No: 5

Date: ___25-11-25_______

TITLE: Fibonacci prime check

AIM/OBJECTIVE(s): To write a function Fibonacci Prime

Check is_fibonacci_prime(n) that checks if a number is

both Fibonacci and prime.

METHODOLOGY & TOOL USED: tool used: VS CODE

To determine whether a number is a Fibonacci prime, the project used a two-step methodology combining Fibonacci detection and primality testing. First, the program checked if the number belongs to the Fibonacci sequence using the property that a number n is Fibonacci if and only if either 5n^2+4 or 5n^2−4 is a perfect square. After confirming this, the algorithm performed a primality test using an efficient method such as trial division up to square root of n, ensuring accuracy while keeping computations simple. Only when both conditions were satisfied did the function classify the number as a Fibonacci prime. This structured approach ensured reliability, avoided generating full Fibonacci sequences, and balanced mathematical precision with computational efficiency.

BRIEF DESCRIPTION: This project implements a Python function **is_fibonacci_prime(n)** that checks whether a given number is both a Fibonacci number and a prime number. The function uses a mathematical identity to verify Fibonacci membership and then applies a primality test to confirm whether the number is prime. Combining these two checks allows the program to identify rare special numbers known as Fibonacci primes, which hold significance in number theory, cryptography, and mathematical research. The function is efficient, easy to use, and demonstrates how theoretical concepts can be converted into practical computational tools.

RESULTS ACHIEVED:

```
start_time = time.time()
result = is_fibonacci_prime(13)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")

Result: True, Runtime: 0.000065s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty faced during the project was understanding how to check Fibonacci membership without generating the entire sequence and grasping why the expressions 5n^2+4 and 5n^2−4 indicate Fibonacci numbers. Another challenge was implementing an efficient primality test that avoids unnecessary computations for large numbers. Handling edge cases such as n=0, n=1, and negative inputs also required careful attention. Putting together two different mathematical tests in one function required clear logic and systematic debugging, which ultimately strengthened problem-solving and implementation skills.

CONCLUSION: The project successfully demonstrates how to identify Fibonacci primes by integrating Fibonacci detection with primality checking in a single function. Through implementing **is_fibonacci_prime(n)**, the student gained deeper insight into number-theoretic properties, efficient computational techniques, and the rarity of such special numbers. Despite initial challenges, the final function works accurately and efficiently, reinforcing concepts like perfect squares, modular arithmetic, and algorithm design. Overall, the project highlights how elegant mathematical identities and careful programming can be combined to solve interesting real-world computational problems.

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BCE10884

Name of Student : Deeptasundar Mohanty

Course Name :Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name : SCAI

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Lucas numbers generator	25-11-25	
2	Perfect powers check	25-11-25	
3	Collatz sequence	25-11-25	
4	Polygonal numbers	25-11-25	
5	Carmichael number check	25-11-25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: ___25-11-25_______

TITLE: Lucas numbers generator

AIM/OBJECTIVE(s): To write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1).

METHODOLOGY & TOOL USED: tool used: VS CODE

To generate the Lucas numbers, the project followed an iterative methodology based on the recurrence relation $Ln=Ln-1+Ln-2$, which is similar to the Fibonacci sequence but starts with the initial values 2 and 1. The function initialized a list with these two base values and then used a loop to compute subsequent terms until the first n Lucas numbers were produced. Each new term was calculated using only the previous two terms, ensuring efficiency without unnecessary memory use. Input validation was also included to handle cases where n is 1 or 2. This methodology ensured that the generator was simple, efficient, and directly aligned with the mathematical definition of the Lucas sequence.

BRIEF DESCRIPTION: This project implements a Python function **lucas_sequence(n)** that generates and returns the first n Lucas numbers. Lucas numbers follow a pattern similar to the Fibonacci sequence but begin with 2 and 1, giving rise to a unique yet related number series used in number theory, combinatorics, and recursive algorithms. The function constructs the sequence iteratively and stores each value in a list, which is returned once the required number of terms is generated. This tool helps students understand recursive sequences and how simple recurrence relations can be converted into efficient computational algorithms.

RESULTS ACHIEVED:

```
start_time = time.time()
result = lucas_sequence(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76], Runtime: 0.000056s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty faced during the project was understanding the difference between the Fibonacci and Lucas sequences, especially how the change in starting values affects the entire progression. Another challenge was managing edge cases such as when the user requests only one or two terms, which required special handling to prevent errors or incorrect outputs. Ensuring that the loop generated exactly n values without off-by-one mistakes also required careful debugging. These challenges improved the student's understanding of recursive sequences, indexing, and iterative algorithm design.

CONCLUSION: The project successfully demonstrates how to generate Lucas numbers using a simple iterative algorithm rooted in a clear mathematical recurrence relation. Implementing **lucas_sequence(n)** helped reinforce the concepts of recursion, sequence generation, and efficient looping structures. Overcoming challenges related to initialization, edge cases, and logical flow enhanced both mathematical intuition and programming confidence. Overall, the project shows how classical number sequences like the Lucas series can be easily computed and applied in various mathematical and computational contexts.

Practical No: 2

Date: ___25-11-25________

TITLE: **Perfect powers check**

AIM/OBJECTIVE(s): Write a function for Perfect Powers

Check is_perfect_power(n) that checks if a number can be

expressed as a^b where a > 0 and b > 1.

METHODOLOGY & TOOL USED: tool used: VS CODE

To determine whether a number n is a perfect power, the project used a systematic approach based on checking all possible exponent values. Since any perfect power can be expressed as n=a^b with b>1, the algorithm iterated over possible values of b from 2 up to $\log_2(n)$, because any higher exponent would exceed the range. For each exponent, the base a was estimated using integer rounding of n^{1/b} and then verified by checking whether a^b = n. This avoided brute-force base iteration and ensured efficient computation even for large numbers. Input validation and mathematical bounds were used to keep the search space optimized and accurate.

BRIEF DESCRIPTION: This project implements a Python function **is_perfect_power(n)** that checks whether a given integer can be written as a^b with a>0 and b>1. Perfect powers include values like 4 (2^2), 8 (2^3), 27 (3^3), and 81 (3^4). The function systematically tests possible exponent values and verifies whether any valid base exists that satisfies the equation. Perfect-power checks are useful in number theory, cryptography, factorization algorithms, and simplifying mathematical expressions. The function provides a practical way to classify numbers based on their exponential structure.

RESULTS ACHIEVED:

```
start_time = time.time()
result = is_perfect_power(16)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000087s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: One of the main difficulties faced during this project was understanding how to efficiently limit the range of exponent values, since testing all bases and exponents would be computationally expensive. Computing integer roots accurately for each exponent also posed a challenge due to rounding issues, which required careful verification to avoid false positives. Handling edge cases such as n=1, negative numbers, or very large values required additional checks and input validation. Debugging the exponent–base calculations helped strengthen understanding of logarithms, integer powers, and computational number theory.

CONCLUSION: The project successfully demonstrates how to identify perfect powers using an efficient exponent-based method rather than brute force. Implementing **is_perfect_power(n)** enhanced understanding of mathematical properties related to exponential forms, integer roots, and logarithmic bounds. Despite initial challenges, the final function performs accurately and efficiently across a wide range of inputs. Overall, the project strengthened problem-solving skills and highlighted how mathematical reasoning and programming techniques combine to analyze numerical patterns in number theory.

Practical No: 3

Date: ___25-11-25_______

TITLE: **Collatz sequence**

AIM/OBJECTIVE(s): Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps

for n to reach 1 in the Collatz conjecture.

METHODOLOGY & TOOL USED: tool used: VS CODE

To calculate the number of steps required for a number n to reach 1 under the Collatz conjecture, the project used an iterative methodology directly based on the Collatz rules. Starting with the input value, the function repeatedly applied the transformation: if n is even, replace it with n/2; if n is odd, replace it with 3n+1. A counter was maintained to track the number of iterations taken until the value reached 1. The method included input validation to ensure n was positive, and the loop continued until the stopping condition was met. This approach is simple, efficient, and accurately follows the structure of the Collatz sequence while ensuring correct step-counting.

BRIEF DESCRIPTION: This project implements a Python function **collatz_length(n)** that returns the number of steps the number n takes to reach 1 according to the Collatz conjecture. The conjecture describes a sequence in which each integer eventually falls to 1 through repeated application of specific transformation rules, creating surprisingly complex behaviour despite its simple definition. The function tracks each transformation and counts the total steps, providing insight into how long different starting numbers take to collapse. This project helps illustrate how mathematical conjectures can be explored computationally using straightforward code.

RESULTS ACHIEVED:

```
start_time = time.time()
result = collatz_length(27)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 112, Runtime: 0.000068s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: One of the main difficulties faced during the project was managing very large intermediate values resulting from the "3n + 1" rule, which can make the sequence grow unexpectedly before collapsing. Understanding why the sequence eventually reaches 1 for all tested values, despite no formal proof existing, was also conceptually challenging. Ensuring the correctness of the loop and avoiding off-by-one errors in the step count required careful debugging. Handling invalid inputs, such as zero or negative integers, also needed proper checks to prevent runtime errors. These challenges helped improve skills in algorithm design, iteration control, and logical reasoning.

CONCLUSION: The project successfully demonstrates how to compute the step count for the Collatz sequence using a simple iterative function based on the conjecture's rules. Implementing **collatz_length(n)** helped deepen understanding of iterative processes, mathematical patterns, and the behavior of sequences generated by recursive rules. Although the conjecture remains unproven, writing code to explore it provides valuable insights into algorithmic thinking and computational experimentation. Overall, the project strengthened problem-solving skills and showcased how a famous open mathematical problem can be studied through programming.

Practical No: 4

Date: ___25-11-25________

TITLE: **Polygonal numbers**

AIM/OBJECTIVE(s): To write a function Polygonal Numbers polygonal_number(s,n) that returns the n-th s-gonal number.

METHODOLOGY & TOOL USED: tool used: VS CODE

The methodology involved taking the inputs s (number of sides) and n (term number), validating that both were positive integers, and then applying the formula directly to compute the value. This avoided iterative or geometric constructions and provided an efficient, constant-time computation. The function was tested on known sequences like triangular, square, pentagonal, and hexagonal numbers to ensure correctness. This mathematical, formula-based approach ensured accuracy, speed, and simplicity.

BRIEF DESCRIPTION: This project implements a Python function **polygonal_number(s, n)** that returns 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), and pentagonal numbers (s=5). Using the standard formula from number theory, the function directly computes the term without generating the entire sequence. The project helps students understand how geometric patterns translate into algebraic formulas and how these formulas can be implemented efficiently in code.

RESULTS ACHIEVED:

```
start_time = time.time()
result = polygonal_number(5, 3)  # Pentagon
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 12, Runtime: 0.000055s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty encountered during the project was understanding and memorizing the general polygonal number formula, especially how the coefficients relate to the shape's number of sides. Another challenge was verifying that the formula produced correct values for different sequences, which required cross-checking with known triangular, square, and pentagonal numbers. Handling invalid inputs, such as non-integer or too-small values for s and n, also required careful validation. These challenges strengthened understanding of number patterns, formula derivation, and implementing mathematical expressions correctly in code.

CONCLUSION: The project successfully demonstrates how polygonal numbers can be computed using a direct mathematical formula rather than iterative or geometric methods. Implementing **polygonal_number(s, n)** provided valuable insights into generalizing number sequences and understanding how shapes relate to algebraic patterns. Overcoming formula interpretation and input-handling challenges improved both mathematical reasoning and programming skills. Overall, the project highlights the beauty of number theory and shows how a single function can generate a wide variety of well-known and specialized sequences efficiently.

Practical No: 5

Date: _25-11-25_________

TITLE: **Carmichael number check**

AIM/OBJECTIVE(s): To 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.

METHODOLOGY & TOOL USED: tool used: VS CODE

To determine whether a given number nnn is a Carmichael number, the project followed the formal Korselt's Criterion, which states that a composite number n is a Carmichael number if and only if it is square-free and for every prime divisor p of n, the condition p−1|n−1 holds. The methodology began by checking that n is composite, then verifying that it has no repeated prime factors by performing prime factorization. Using the list of distinct prime divisors, the function tested the divisibility condition for each prime. Only when all checks were satisfied did the function classify n as a Carmichael number. This mathematical approach avoids brute-force testing of all values of a, making the process efficient and theoretically sound.

BRIEF DESCRIPTION: This project implements a Python function **is_carmichael(n)** that determines whether a composite number n satisfies the condition a^n−1≡1mod n for all integers a that are coprime to n. Known as Carmichael numbers, these rare integers behave like primes under Fermat's Little Theorem, despite being composite. The function uses prime factorization and Korselt's Criterion to validate whether the number meets the mathematical conditions that define Carmichael numbers. This tool highlights connections between number theory, cryptography, and pseudoprime behavior.

RESULTS ACHIEVED:

```
start_time = time.time()
result = is_carmichael(561)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000333s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty encountered during the project was understanding why Carmichael numbers behave like primes in modular exponentiation and how Korselt's Criterion replaces direct testing of all coprime values. Implementing prime factorization correctly and ensuring that the number is square-free required careful coding, especially for larger inputs. Another challenge was handling edge cases such as prime numbers, 1, or negative inputs, which must all be excluded. Debugging logical errors in the divisibility conditions and ensuring the function correctly identifies known Carmichael numbers helped strengthen both mathematical insight and programming accuracy.

CONCLUSION: The project successfully demonstrates how to identify Carmichael numbers using Korselt's Criterion, combining mathematical theory with efficient algorithmic implementation. Writing **is_carmichael(n)** deepened understanding of pseudoprimes, modular arithmetic, and prime factorization while highlighting why such numbers are significant in cryptography. Despite initial challenges with factorization logic and theoretical conditions, the final function works accurately and efficiently. Overall, the project strengthened analytical thinking and showcased how advanced number-theoretic concepts can be implemented through clear and well-structured code.

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BCE10884
Name of Student	:DEEPTASUNDAR MOHANTY
Course Name	:Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	:SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Miller-rabin test	25-11-25	
2	Pollard's rho algorithm	25-11-25	
3	Riemann zeta function ζ(s)	25-11-25	
4	Partition Function	25-11-25	
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: _____25-11-25______

TITLE: **Miller-rabin test**

AIM/OBJECTIVE(s): To implement the probabilistic Miller-Rabin

test is_prime_miller_rabin(n, k) with k rounds.

METHODOLOGY & TOOL USED: tool used: VS CODE

To implement the Miller–Rabin probabilistic primality test, the project used the standard decomposition of n−1 into the form 2^r·d, where d is odd. The methodology involved selecting a random base a in each of the k rounds and computing a^d mod n using efficient modular exponentiation. If the result was 1 or n−1, the round passed; otherwise, the value was repeatedly squared up to r−1 times to check whether it ever reached n−1. If no such value appeared, the number was declared composite. After k successful rounds, the number was classified as "probably prime." This approach allowed fast and reliable primality checking, especially for large integers, by combining modular arithmetic with repeated randomized tests.

BRIEF DESCRIPTION: This project implements a Python function **is_prime_miller_rabin(n, k)** that determines whether a number is prime using the Miller–Rabin probabilistic test. Instead of checking all possible divisors, the algorithm uses modular exponentiation and randomness to test whether n behaves like a prime number under certain mathematical conditions. Although the test is probabilistic, performing multiple rounds greatly reduces the chance of error, making the method widely used in cryptography and large-number computations. The function returns "probably prime" or "composite," offering an efficient alternative to deterministic primality testing for large values.

RESULTS ACHIEVED:

```python
start_time = time.time()
result = is_prime_miller_rabin(13)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000155s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main difficulty in this project was understanding the mathematical decomposition of n−1 into 2^r·d and why this transformation is crucial for the correctness of the test. Implementing modular exponentiation correctly and ensuring that the repeated squaring process followed Miller–Rabin's logic required careful coding and debugging. Handling edge cases such as small numbers, even values, and invalid inputs also posed challenges. Additionally, it took time to understand the probabilistic nature of the algorithm and how multiple rounds reduce the error probability. These challenges improved the student's understanding of modular arithmetic, randomness, and algorithmic complexity.

CONCLUSION: The project successfully demonstrates how to use the Miller–Rabin test as an efficient and practical primality-checking algorithm. Implementing **is_prime_miller_rabin(n, k)** strengthened understanding of modular exponentiation, probabilistic algorithms, and number-theoretic reasoning. Despite initial challenges in decomposition logic and repeated squaring, the final function performs accurately and efficiently across a wide range of integers. Overall, the project showcases how advanced mathematical concepts can be translated into powerful computational tools, especially for applications involving cryptography and large prime numbers.

Practical No: 2

Date: ___25-11-25________

TITLE: **Pollard's rho algorithm**

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

METHODOLOGY & TOOL USED: tool used: VS CODE

To implement the pollard_rho(n) function for integer factorization, the approach begins by selecting a pseudo-random polynomial function, typically f(x)=x^2+c mod n, where c is a small constant. Two values, often referred to as the **tortoise** and **hare**, are generated using this function to simulate Floyd's cycle-finding algorithm. At each iteration, the algorithm computes the greatest common divisor (gcd) of the absolute difference between the tortoise and hare with respect to n. If this gcd becomes a non-trivial factor, it is returned as a divisor of n. If it equals 1, iteration continues; if it equals n, the function restarts with a different constant or starting value.

BRIEF DESCRIPTION: Pollard's Rho is a probabilistic factorization algorithm that efficiently finds non-trivial factors of large composite numbers using very little memory. It relies on the idea that iterating a polynomial modulo n eventually falls into a cycle, and by comparing two sequences moving at different speeds, a gcd-based collision is detected. When such a collision reveals a non-trivial divisor, the algorithm successfully factors the number. It is significantly faster than trial division and works especially well for numbers with small factors.

RESULTS ACHIEVED:

```
start_time = time.time()
result = pollard_rho(315)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 21, Runtime: 0.000071s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: While implementing Pollard's Rho, the student struggled primarily with understanding the interplay between **cycle detection** and the **gcd computation**, as both concepts seem unrelated at first glance. Debugging was challenging because the algorithm is probabilistic, meaning it may fail or require restarts without any obvious error. Handling edge cases, such as even numbers, trivial factors, or when the algorithm repeatedly returns the number itself, also caused confusion. Ensuring the polynomial iteration stayed within modular constraints required careful attention.

CONCLUSION: The implementation of pollard_rho(n) helped demonstrate the power of probabilistic algorithms in solving complex computational problems like factorization. Despite initial difficulties in understanding its logic flow and random nature, the student eventually achieved a functional and efficient solution. This algorithm reinforced the importance of number theory concepts such as modular arithmetic, gcd, and cycle detection. Overall, Pollard’s Rho proved to be both an insightful and practical exercise in modern algorithmic factorization.

Practical No: 3

Date: ___25-11-25________

TITLE: **Riemann zeta function ζ(s)**

AIM/OBJECTIVE(s): To write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of

the series.

METHODOLOGY & TOOL USED: tool used: VS CODE

The method involves iterating from 1 to the specified number of terms and adding each term's contribution 1/n^s to a running total. Since s can be any real number greater than 1, the implementation ensures proper handling of floating-point exponentiation and accumulation. The algorithm is straightforward: initialize a sum variable, loop, compute each term, add it, and return the final approximation.

BRIEF DESCRIPTION: The function zeta_approx(s, terms) provides a numerical approximation of the Riemann zeta function by summing the first few terms of its infinite series. Because the true zeta function involves an infinite sum, truncating it after a finite number of elements gives an approximate value that improves as the number of terms increases. This approach works well for s>1, where the series is convergent, and allows the user to balance accuracy with computational cost by adjusting the number of terms.

RESULTS ACHIEVED:

```
start_time = time.time()
result = zeta_approx(2, 1000)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024  # bytes
print(f"Result: {result:.6f}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")

... Result: 1.643935, Runtime: 0.000199s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The main challenges the student faced were understanding the convergence requirements of the series and handling floating-point precision errors. Initially, it was confusing why the function behaves poorly for values of s≤1, prompting research into divergence properties. Another issue was ensuring that the exponentiation n^s was computed accurately without slowing the program too much, especially for large values of terms. The student also had to manage the trade-off between computational efficiency and approximation accuracy.

CONCLUSION: Implementing the zeta_approx(s, terms) function helped the student understand how infinite mathematical series can be approximated computationally. Through the process, they gained insight into convergence behavior, numerical precision, and the limits of simple summation-based approximations. Despite the initial hurdles, the student successfully created a functional and flexible implementation, highlighting how theoretical mathematical concepts can be converted into practical algorithms.

Practical No: 4

Date: __25-11-25________

TITLE: **Partition Function**

AIM/OBJECTIVE(s): To 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.

METHODOLOGY & TOOL USED: tool used: VS CODE

To implement the partition_function(n) algorithm, the student used a dynamic programming approach based on Euler's recurrence generating-function identity. The core idea is to build an array dp where dp[k] represents the number of partitions of the integer k. Starting with the base case dp[0] = 1, the algorithm iteratively updates the table using all integers from 1 to n as potential contributors to the sum. For each integer i, the algorithm adds dp[j - i] to dp[j] for all j ≥ i. This ensures that every unique combination of integers contributing to a partition is counted exactly once, resulting in an efficient method compared to brute-force enumeration.

BRIEF DESCRIPTION: The function partition_function(n) computes how many distinct ways the number n can be written as a sum of positive integers, where order does not matter (e.g., 4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1). Since the number of partitions grows rapidly, direct enumeration is impractical for larger values. The dynamic programming technique incrementally builds the number of partitions for smaller numbers and uses them to determine the partitions of larger ones. This results in a fast and reliable method for computing the partition count.

RESULTS ACHIEVED:

```
start_time = time.time()
result = partition_function(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
... Result: 42, Runtime: 0.000070s, Memory: 148709376 bytes
```

DIFFICULTY FACED BY STUDENT: The student initially struggled with understanding the difference between *combinations* and *partitions*, specifically why order should not be counted separately. Translating the mathematical definition of partitions into an iterative dynamic programming structure was also challenging. Another difficulty was ensuring that the inner loop iterated in the correct direction to avoid overcounting partitions. Handling large values was intimidating as well, since partition numbers grow extremely fast and require careful thought about memory and computational efficiency.

CONCLUSION: By completing the partition_function(n) implementation, the student not only learned a practical use of dynamic programming but also developed a deeper understanding of combinatorial mathematics. The task illustrated how complex counting problems can become efficient through structured recurrence relations. Although the concept was initially confusing and the implementation required attention to detail, the student ultimately succeeded in producing a correct and efficient solution, reinforcing both mathematical insight and programming confidence.