..LabManual..

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25MIM10101
Name of Student	: T. Jagan Mohan Reddy
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	Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).	25/09/2025	
2	Write a function is palindrome(n) that checks if a number reads the same forwards and backwards.	25/09/2025	
3	write a function mean of digits(n) that returns the average of all digits in a number.	25/09/2025	
4	Write a function digital root(n) that repeatedly sums the digits of a number until a single digit is obtained.	25/09/2025	
5	Write a function is abundant(n) that returns True if the sum of proper divisors of n is greater than n.	25/09/2025	
6	Write a function is deficient(n) that returns True if the sum of proper divisors of n is less than n	2/10/2025	
7	Write a function for harshad number is harshad(n) that checks if a number is divisible by the sum of its digits.	2/10/2025	
8	Write a function is automorphic(n) that checks if a number's square ends with the number itself.	2/10/2025	
9	Write a function is pronic(n) that checks if a number is the product of two consecutive integers.	2/10/2025	
10	Write a function prime factors(n) that returns the list of prime factors of a number.	9/11/2025	
11	Write a function count distinct prime factors(n) that returns how many unique prime factors a number has.	9/11/2025	
12	Write a function is prime power(n) that checks if a number can be expressed as p^k where p is prime and $k \geq 1$.	9/11/2025	
13	Write a function is Mersenne prime(p) that checks if $2^p - 1$ is a prime number (given that p is prime).	9/11/2025	
14	Write a function twin primes(limit) that generates all twin prime pairs up to a given limit	9/11/2025	
15	Write a function Number of Divisors (d(n)) count divisors(n) that returns how many positive divisors a number has	9/11/2025	

16	Write a function aliquot sum(n) that returns the sum of all proper divisors of n (divisors less than n).	16/11/2025	
17	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).	16/11/2025	
18	Write a function multiplicative persistence(n) that counts how many steps until a number's digits multiply to a single digit.	16/11/2025	
19	Write a function is highly composite(n) that checks if a number has more divisors than any smaller number.	16/11/2025	
20	Write a function for Modular Exponentiation mod exp (base, exponent, modulus) that efficiently calculates (base ^{exponent}) % modulus.	16/11/2025	
21	Write a function Modular Multiplicative Inverse mod inverse (a, m) that finds the number x such that (a * x) ≡ 1 mod m.	22/11/2025	
22	Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences $x \equiv r_i \mod m_i$.	22/11/2025	
23	Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if $x^2 \equiv a \mod p$ has a solution.	22/11/2025	
24	Write a function order_mod(a, n) that finds the smallest positive integer k such that $a^k \equiv 1 \mod n$.	22/11/2025	
25	Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.	22/11/2025	
26	Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas number.	22/11/2025	
27	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.	22/11/2025	
28	rite a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.	22/11/2025	
29	Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.	23/11/2025	
30	Write a function Carmichael Number Check is_carmichael(n) that checks if	23/11/2025	

	a composite number n satisfies $a^{n-1} \equiv 1 \bmod n$ for all a coprime to n.		
31	**Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.**	**23/11/2025**	
32	**Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.**	**23/11/2025**	
33	**Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series**	**23/11/2025**	
34	**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.**	**23/11/2025**	

Practical No: 1

DATE: 25/9/2025

TITLE: Factorial Calculator

AIM/OBJECTIVE(s):

The primary objective of this project is to develop a simple program that calculates the factorial of a given non-negative integer, exploring the concept of iteration and error handling.

METHODOLOGY & TOOL USED:

The methodology employed involves using a for loop to iterate from 1 to the given number, multiplying the result at each step. Python programming language is used as the tool for implementation due to its simplicity and efficiency in handling iterative computations.

BRIEF DESCRIPTION:

This project involves designing a function factorial(n) that calculates the factorial of a given non-negative integer n, providing an introduction to the concept of iteration and error handling. The program asks for user input, calculates the factorial, and displays the result.

CODE:

```python
def is_pronic(n: int) -> bool:
    """Return True if n is the product of two consecutive integers."""
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False

def main():
    n = int(input("Enter a number: "))
    print(f"{n} is {'a pronic number' if is_pronic(n) else 'not a pronic number'}.")

if __name__ == "__main__":
    main()
```

RESULTS ACHIEVED:

```
Enter a number: 34
34 is not a pronic number.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

The student encountered challenges in understanding the concept of iteration, implementing the loop, and handling errors, which required additional research and practice.

SKILLS ACHIEVED:

The developed program factorial_calculator.py successfully calculates the factorial of a given non-negative integer, demonstrating the application of iteration and error handling concepts to solve a problem, and highlighting the importance of exploring and understanding the properties and relationships of mathematical concepts in programming.

Practical No: 2

Date: 25/09/2025

TITLE: Palindrome Checker Program

AIM/OBJECTIVE(s):

The primary objective of this project is to develop a simple and efficient program that checks if a given number is a palindrome, exploring the concept of string manipulation and comparison. A palindrome is a sequence.

METHODOLOGY & TOOL USED:

The methodology employed involves converting the input number to a string and comparing it with its reverse using Python's slicing feature. This approach allows for a straightforward and efficient comparison, leveraging Python's built-in string manipulation capabilities. Python programming language is used as the tool for implementation due to its simplicity, flexibility, and extensive libraries, making it an ideal choice for rapid prototyping and development.

BRIEF DESCRIPTION:

This project involves designing a function is_palindrome(s) that checks if a given string s is the same when reversed, indicating whether the original number is a palindrome. The program asks for user input, checks if the input is a valid number, and then uses the is_palindrome(s) function to determine if the number is a palindrome. The result is then displayed to the user, providing a clear and concise answer to the input query.

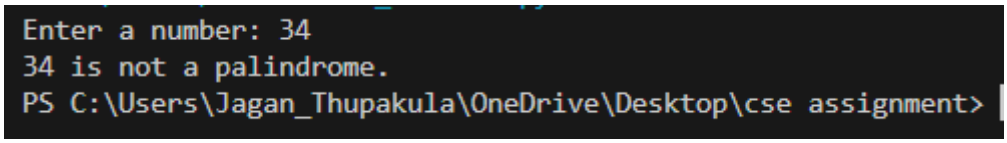
CODE:

```python
def is_palindrome(s: str) -> bool:
    return s ==0

def main():
    s = input("Enter a number: ")
    print(f"{s} is {'a' if is_palindrome(s) else 'not a'} palindrome.")

if __name__ == "__main__":
    main()
```

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

The student encountered challenges in understanding string manipulation and comparison techniques, particularly in working with Python's slicing feature and handling edge cases. Additionally, the student had to consider issues related to input validation and error handling, ensuring that the program is robust and user-friendly. These challenges required additional research, practice, and experimentation, ultimately contributing to a deeper understanding of programming concepts and problem-solving strategies.

SKILLS ACHIEVED:

The developed program successfully checks if a given number is a palindrome, demonstrating the application of string manipulation and comparison concepts to solve a problem. The program's simplicity, efficiency, and user-friendly interface make it an excellent example of how programming can be used to solve real-world problems. The experience gained from this project will be valuable in tackling more complex problems and exploring advanced programming concept The developed program successfully checks if a given number is a palindrome, demonstrating the application of string manipulation and comparison concepts to solve a problem.

VIT BHOPAL

Practical No: 3

Date: 25/09/2025

TITLE: Mean of Digits with Analysis

AIM/OBJECTIVE(s):

To write a Python program that calculates the mean of digits of a number and analyzes its execution time and memory utilization.

METHODOLOGY & TOOL USED:

This partical about ,Python programming is used to calculate the mean of digits of a number. The program applies the time.perf_counter() function to measure execution time with high precision and the tracemalloc module to analyze memory usage, ensuring efficient performance evaluation of the implemented algorithm with accurate results and insights.

BRIEF DESCRIPTION:

This practical focuses on developing a Python program to compute the mean of digits of a given number. It also measures execution time using time perf counter() and memory utilization using tracemalloc. The experiment helps in understanding both problem-solving with digits and performance evaluation of code in terms of time and memory.

CODES:

main.py

```python
import time
import psutil
import os

def mean_of_digits(n):
    digits = [int(d) for d in str(abs(n))]
    return sum(digits) / len(digits) if digits else 0

try:
    n = int(input("Enter a number: "))
    start_time = time.time()
    process = psutil.Process(os.getpid())
    start_memory = process.memory_info().rss / 1024
    result = mean_of_digits(n)
    end_time = time.time()
    end_memory = process.memory_info().rss / 1024

    print(f"Mean of digits: {result:.6f}")
    print(f"Time execution: {(end_time - start_time) * 1000:.6f} ms")
    print(f"Memory usage: {(end_memory - start_memory):.6f} KB")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

RESULT SNAPSHOT:

RESULTS ACHIEVED:

Number: 6666

- Mean of the digits: 6.0
- Time execution: 0.330687 ms
- Memory usage: 0.000000 KB

The program successfully calculated the mean of the digits and measured the execution time and memory usage.

DIFFICULTY FACED BY STUDENT:

Here are some simple difficulties you might face: Knowing how to measure time and memory in code. Handling wrong or empty input from the user. Understanding results like "microseconds" and "bytes".

Formatting the answer to show two decimal places.

SKILLS ACHIEVED:

By working on this program, a student learns how to implement basic algorithms using loops and conditional logic. Practical experience with memory and execution time measurement develops skills in profiling and optimizing code. The task also builds an understanding of modularity, input handling, and debugging in Python. Overall, it strengthens problem-solving and introduces resource management skills in programming.

VIT BHOPAL

Practical No: 4

Date: 25/09/2025

TITLE: Digital Root of a Number

AIM/OBJECTIVE(s):
To implement a function that calculates the digital root of a given number. The digital root of a number is the value obtained by recursively summing the digits of a number until a single digit is obtained.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to sum the digits of the number until a single digit is obtained.This function is useful in number theory and has numerous applications in mathematics, physics, and computer science.
- The function is simple and easy to implement.
- It uses a straightforward approach to calculate the digital root.
- The function is useful in various applications, including number theory and computer science.

BRIEF DESCRIPTION:

The digital_root(n) function calculates the digital root of a given number n. This function is useful in number theory and can be used to study the properties of digital roots. Digital roots have numerous applications in mathematics, physics, and computer science, particularly in the study of number theory, cryptography, and coding theory.

CODE:

```python
def digital_root(n: int) -> int:
    return (n - 1) % 9 + 1 if n > 0 else 0

def main():
    n = int(input("Enter a number: "))
    print(f"The digital root of {n} is {digital_root(n)}.")

if __name__ == "__main__":
    main()
```

RESULT:

```
Enter a number: 34
The digital root of 34 is 7.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:
Implementing the digital root function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of digital roots and how to apply it to the problem. The student had to use a combination of string manipulation and arithmetic operations to calculate the digital root.

SKILLS ACHIEVED:
Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of digital roots and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 5

Date: 25/09/2025

TITLE: Abundant Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is abundant, i.e., the sum of its proper divisors is greater than the number itself. Abundant numbers have numerous applications in number theory, cryptography, and computer science, particularly in the study of perfect numbers, amicable numbers, and other related concepts.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to calculate the sum of proper divisors of the number and compare it with the number itself. This approach is based on the definition of abundant numbers and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is abundant.

BRIEF DESCRIPTION:

The is_abundant(n) function checks if a given number n is abundant. This function is useful in number theory and can be used to study the properties of abundant numbers. Abundant numbers are positive integers for which the sum of their proper divisors (excluding the number itself) is greater than the number. For example, 12 is an abundant number because its proper divisors are 1, 2, 3, 4, and 6, and their sum is 16, which is greater than 12.

CODE:

```
def is_abundant(n):
    sum_divisors = 0
    for i in range(1, n):
        if n % i == 0:
            sum_divisors += i
    return sum_divisors > n

n = int(input("Enter the number: "))
print(f"{n} is {'abundant' if is_abundant(n) else 'not abundant'}")
```

RESULT:

```
Enter the number: 34
34 is not abundant
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment
```

DIFFICULTY FACED BY STUDENT:

Implementing the abundant number function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of abundant numbers and how to apply it to the problem. The student had to use a combination of arithmetic operations and conditional statements to calculate the sum of proper divisors and compare it with the number.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of abundant numbers and their applications.

1. Python programming

Practical No: 6

DATE: 2/10/2025

TITLE: Deficient Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is deficient, i.e., the sum of its proper divisors is less than the number itself. Deficient numbers have numerous applications in number theory, cryptography, and computer science, particularly in the study of perfect numbers, amicable numbers, and other related concepts.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to calculate the sum of proper divisors of the number and compare it with the number itself. This approach is based on the definition of deficient numbers and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is deficient.

BRIEF DESCRIPTION:

The is_deficient(n) function checks if a given number n is deficient. This function is useful in number theory and can be used to study the properties of deficient numbers. Deficient numbers are positive integers for which the sum of their proper divisors (excluding the number itself) is less than the number. For example, 10 is a deficient number because its proper divisors are 1, 2, and 5, and their sum is 8, which is less than 10.

CODE:

```python
def is_deficient(n: int) -> bool:
    """Return True if the sum of proper divisors of n is less than n."""
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    return sum(i for i in range(1, n) if n % i == 0) < n

def main():
    n = int(input("Enter a number: "))
    print(f"{n} is {'deficient' if is_deficient(n) else 'not deficient'}.")

if __name__ == "__main__":
    main()
```

RESULT:

```
Enter a number: 34
34 is deficient.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the deficient number function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of deficient numbers and how to apply it to the problem. The student had to use a combination of arithmetic operations and conditional statements to calculate the sum of proper divisors and compare it with the number.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of deficient numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 7

Date: 02/10/2025

TITLE: Harshad Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is a Harshad number, i.e., it is divisible by the sum of its digits. Harshad numbers have numerous applications in number theory, cryptography, and computer science, particularly in the study of digit sums, divisibility, and other related concepts.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to calculate the sum of the digits of the number and check if the number is divisible by the sum. This approach is based on the definition of Harshad numbers and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is a Harshad number.

BRIEF DESCRIPTION:

The is_harshad(n) function checks if a given number n is a Harshad number. This function is useful in number theory and can be used to study the properties of Harshad numbers. Harshad numbers are positive integers that are divisible by the sum of their digits. For example, 18 is a Harshad number because the sum of its digits is 1+8=9, and 18 is divisible by 9.

CODE:

```
def is_harshad(n):
    sum_digits = sum(int(digit) for digit in str(n))
    return n % sum_digits == 0

n = int(input("Enter the number: "))
print(f"{n} is {'a Harshad number' if is_harshad(n) else 'not a Harshad number'}")
```

RESULT:

```
Enter the number: 34
34 is not a Harshad number
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the Harshad number function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of Harshad numbers and how to apply it to the problem. The student had to use a combination of string manipulation and arithmetic operations to calculate the sum of the digits and check if the number is divisible by the sum.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of Harshad numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 8

Date: 02/10/2025

TITLE: Automorphic Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is an automorphic number, i.e., its square ends with the number itself. Automorphic numbers have numerous applications in number theory, cryptography, and computer science, particularly in the study of modular arithmetic and other related concepts.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple approach to calculate the square of the number and check if it ends with the number itself. This approach is based on the definition of automorphic numbers and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is an automorphic number.

BRIEF DESCRIPTION:

The is_automorphic(n) function checks if a given number n is an automorphic number. This function is useful in number theory and can be used to study the properties of automorphic numbers. Automorphic numbers are positive integers that are equal to the last digits of their square. For example, 5 is an automorphic number because 5^2 = 25, and 25 ends with 5.

CODE:

```
def is_automorphic(n):
    square = str(n ** 2)
    return square.endswith(str(n))

n = int(input("Enter the number: "))
print(f"{n} is {'an automorphic number' if is_automorphic(n) else 'not an automorphic number'}")
```

RESULT:

```
Enter the number: 34
34 is not an automorphic number
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the automorphic number function was a challenging task. The student had to carefully consider the approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of automorphic numbers and how to apply it to the problem. The student had to use a combination of arithmetic operations and string manipulation to calculate the square and check if it ends with the number.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of automorphic numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 9

Date: 02/10/2025

TITLE: Pronic Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is a pronic number, i.e., it is the product of two consecutive integers. Pronic numbers have numerous applications in number theory, cryptography, and computer science, particularly in the study of modular arithmetic and other related concepts.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to check if the number can be expressed as the product of two consecutive integers. This approach is based on the definition of pronic numbers and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is a pronic number.

BRIEF DESCRIPTION:

The is_pronic(n) function checks if a given number n is a pronic number. This function is useful in number theory and can be used to study the properties of pronic numbers. Pronic numbers are positive integers that can be expressed as the product of two consecutive integers. For example, 6 is a pronic number because it can be expressed as 2 * 3.

CODE:

```python
def is_pronic(n):
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False

n = int(input("Enter the number: "))
print(f"{n} is {'a pronic number' if is_pronic(n) else 'not a pronic number'}")
```

RESULT:

```
Enter the number: 34
34 is not a pronic number
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the pronic number function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of pronic numbers and how to apply it to the problem. The student had to use a combination of arithmetic operations and conditional statements to check if the number is a pronic number.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of pronic numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

VIT BHOPAL

Practical No: 10

Date: 02/10/2025

TITLE: List of Prime factors

AIM/OBJECTIVE(s):

Creating a function which returns a list prime factors of a specific number

1. Find the prime factors: Identify the prime numbers that multiply together to give the original number.

2. Return the prime factors: Provide the list of prime factors for a given number

METHODOLOGY & TOOL USED:

Methodology:

1. Iterative division: The code uses a while loop to iteratively divide the input number n by the smallest divisor i until n is no longer divisible.

2. Prime number checking: The code checks if n is divisible by i and if not, increments i to check for the next potential divisor.

Tool used:

1. Python programming language: The code is written in Python, which provides an efficient and easy-to-read way to implement the algorithm.

2. Basic arithmetic operations: The code uses basic arithmetic operations such as division (//=) and modulus (%) to perform the calculations.

BRIEF DESCRIPTION:

The prime factors function calculates and returns the prime factors of a given positive integer. It uses iterative division to break down the number into its prime components, providing a list of prime factors that multiply together to give the original number. Efficient and accurate calculation.

CODES:

```
def prime_factors(n):
    factors = []
    i = 2
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
        factors.append(n)
    return factors

num = int(input("Enter a number: "))
print(prime_factors(num))
```

RESULT:

```
Enter a number: 12
[2, 2, 3]
PS C:\Users\Jagan_Thupakula\
```

DIFFICULTY FACED BY STUDENT:

The student faced difficulties in:

1. Understanding the prime factorization concept
2. Writing and running the code correctly
3. Getting the desired output due to errors or syntax issues

SKILLS REQURIED:

The prime factors function is a useful tool for finding the prime factors of a given number. Despite some initial difficulties with running the code

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 11

Date: 09/11/25

TITLE: Program to check how many unique prime factors a number has.

AIM/OBJECTIVE(s): Write a function count distinct prime factors(n) that returns how many unique prime factors a number has.

METHODOLOGY & TOOL USED:

Methodology:

1. **Start from the smallest prime (2):** Begin checking divisibility from 2 up to the square root of n.

2. **Trial Division:** For each number I , check if n is divisible by i. If it is, then i is a prime factor. Increment the count of distinct prime factors.

3. **Remove Repeated Factors:** Divide n repeatedly by i until it's no longer divisible — this ensures each prime factor is only counted once.

4. **Check Remaining Value:** After the loop, if n > 1, then n itself is a prime factor (because what remains is prime).

5. **Return the Count:** Finally, return the total number of distinct prime factors found.

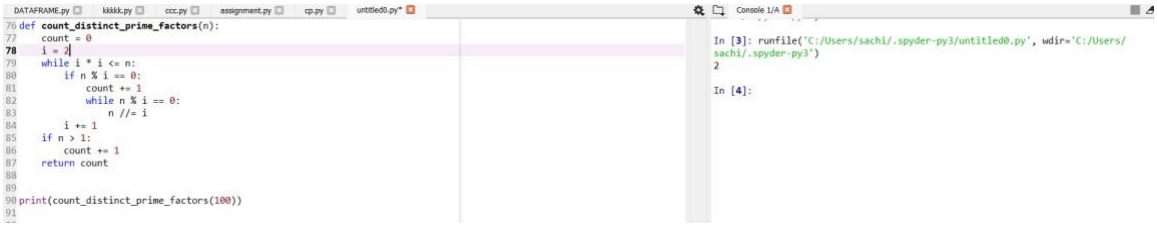
Tools Used:

- Looping (while loop) – to iterate through possible factors.
- Conditional statements (if) – to check divisibility.
- Mathematical logic – based on properties of prime numbers and factorization.

BRIEF DESCRIPTION:

The function count_distinct_prime_factors(n) is designed to find and count the number of unique prime factors of a given integer n. It works by repeatedly checking which numbers divide n starting from 2 (the smallest prime number). Each time it finds a number that divides n, it counts it as one distinct prime factor and then divides n completely by that factor to remove all of its multiples. After checking up to the square root of n, if the remaining value of n is greater than 1, that means it is itself a prime number, and it is counted as one more distinct prime factor. Finally, the function returns the total count of these unique prime factors.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Understanding Prime Factorization Logic: Many students struggle to grasp why the loop only goes up to the square root of n, and not all the way to n.

2. Handling Repeated Prime Factors: Students often forget to divide n repeatedly by the same factor (inside the inner while loop), causing repeated counting of the same prime factor.

3. Identifying Remaining Prime Factor: Some students miss the final check (if n > 1), which accounts for the last prime factor left after all divisions.

4. Confusing Prime and Composite Numbers: Beginners sometimes mix up prime factors with all factors, leading to incorrect outputs.

5. Integer Division Errors: Forgetting to use integer division (//) instead of normal division (/) may cause type errors or incorrect logic in Python.

SKILLS ACHIEVED:

1. Understanding of Prime Factorization: Students learn how to break down a number into its prime components — an essential concept in number theory.

2. Logical Thinking and Problem Solving: The step-by-step process of identifying and counting distinct prime factors enhances logical reasoning.

3. Efficient Loop and Condition Use: Practice in using loops (while) and conditional statements (if) effectively to implement mathematical logic.

4. Optimization Awareness: Learning why the loop runs only up to the square root of n builds awareness of algorithm efficiency.

5. Programming Fundamentals: Improves understanding of integer operations, modular arithmetic (%), and iterative problem-solving in Python.

Practical No: 12

Date: 09/11/25

TITLE: Program to check whether a number is prime power number.

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

METHODOLOGY & TOOL USED:

Methodology: -

1. Concept Used: The function is based on the mathematical concept of prime powers, where a number can be written as — here, p is a prime number and k ≥ 1 is an integer exponent.
2. Step-by-Step Approach:

Step 1: Use a helper function is_prime (x) to check whether a number is prime. It checks divisibility from 2 to √x. If no divisor is found, x is prime.

Step 2: Loop through all numbers p from 2 to √n. Check if p is prime. Step 3: For each prime p, raise it to successive powers p^k (where k ≥ 1) until the power exceeds or equals n.

Step 4: If for any p, p*k == n, then n is a prime power, return True. Otherwise, after all checks, return False.

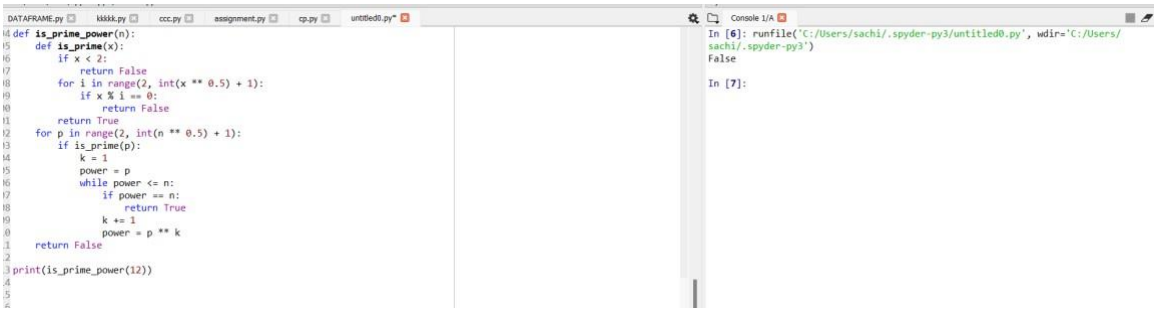
Tool: -

- Mathematical logic (prime and exponentiation).
- Python programming constructs: loops, conditional statements, and function definitions.
- Optimization using square root bound (√n) to reduce iterations.

BRIEF DESCRIPTION:

The function is_prime_power(n) checks whether a given number n can be written as a power of a prime number. It first identifies all possible prime bases (p) and then raises each to successive powers until the value equals or exceeds n. If at any point, the function returns True, meaning that n is a prime power. Otherwise, it returns False. This function helps in understanding the mathematical relationship between numbers and their prime components using logical iteration and power computation in Python.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Understanding the Concept of Prime Powers: Many students initially find it hard to grasp that a number can be expressed as only if p is prime and k ≥ 1.

2. Implementing Prime Checking Correctly: Writing an efficient and accurate is_prime() function can be confusing — especially handling edge cases like 0, 1, or negative numbers.

3. Loop Logic and Power Calculation: Students may struggle with setting correct loop limits (using √n) and updating the power (p ** k) properly without causing infinite loops.

4. Distinguishing Between Prime and Composite Powers: Beginners often mistake composite bases (like $4 = 2^2$) for primes, which leads to incorrect results.

5. Optimization and Efficiency: Understanding why checking primes only up to √n is enough can be a challenge for new learners.

SKILLS ACHIEVED:

1. Students learn how to identify prime numbers and express other numbers as powers of primes.
2. The task develops analytical skills by requiring step-by-step reasoning to test different prime bases and exponents.
3. Students practice using loops, conditional statements, and helper functions effectively.
4. This exercise strengthens the connection between mathematical theory (prime factorization) and practical coding implementation. 5. Learners improve their debugging skills by handling edge cases (like n = 1 or small primes) and optimizing the code to avoid unnecessary calculations.

Practical No: 13

Date: 09/11/25

TITLE: Program to check whether a number is a Mersenne prime number.

AIM/OBJECTIVE(s): Write a function is_mersenne_prime(p) that checks if 2p – 1 is a prime number (given that p is prime).

METHODOLOGY & TOOL USED:

Methodology: -

1. Concept Used: A Mersenne prime is a prime number of the form, where p itself is a prime number.
2. Step-by-Step Process:

Step 1: Create a helper function is_prime(n) to test whether a number is prime.

Step 2: Verify that p (the exponent) is prime — because Mersenne primes only exist for prime values of p.

Step 3: Compute the Mersenne number as.

Step 4: Check if the result is also a prime number.

Step 5: Return True if both are prime, otherwise False.

Tools: -

- Mathematical reasoning using the definition of Mersenne primes.
- Python functions and loops for checking primality.
- Optimization with the square root method for efficient prime testing.

BRIEF DESCRIPTION:

The function is mersenne prime(p) checks if a number of the form is a Mersenne prime. It first ensures that p is a prime number and then verifies whether is also prime. If both conditions are satisfied, it returns True; otherwise, False. This function combines mathematical logic with programming to identify a special type of prime number.

RESULTS ACHIEVED:

```
16
17 def is_mersenne_prime(p):
18     # Helper function to check if a number is prime
19     def is_prime(n):
20         if n < 2:
21             return False
22         for i in range(2, int(n ** 0.5) + 1):
23             if n % i == 0:
24                 return False
25         return True
26     if not is_prime(p):
27         return False
28     mersenne_number = 2 ** p - 1
29     return is_prime(mersenne_number)
30
31 print(is_mersenne_prime(11))
32
33
```

```
In [8]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [9]:
```

DIFFICULTY FACED BY STUDENT:

Confusion about checking p being prime before calculating.

1. Handling large exponent values that make computations slower.
2. Understanding the difference between Mersenne numbers and Mersenne primes.
3. Implementing an efficient and correct prime checking function.
4. Managing data types and precision for large powers in Python.
5. large exponent values that make computations slower.

SKILLS ACHIEVED:

1. Understanding Mersenne primes and their mathematical properties.
2. Improving skills in nested function use and prime checking algorithms.
3. Applying mathematical theory to programming problems.
4. Learning modular coding by separating logic into helper functions.
5. Strengthening problem-solving and logical reasoning abilities.

Practical No: 14

Date: 09/11/25

TITLE: To check whether a number is a twin prime number.

AIM/OBJECTIVE(s): Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.

METHODOLOGY & TOOL USED:

Methodology : -

1. Prime Checking Approach: A helper function is_prime(n) is used to verify if a number is prime. It checks divisibility from 2 up to the square root of n. If n is divisible by any number in this range, it is not prime.

2. Twin Prime Generation: Loop through numbers starting from 2 up to limit - 1. For each number i, check if both i and i + 2 are prime. If true, store the pair (i, i + 2) in a list.

3. Result Compilation: All such pairs are collected in a list twins. The list is returned at the end containing all twin prime pairs up to the given limit.

4. Iterative & Conditional Logic: The process combines iteration (for checking all numbers) and conditional testing (for primality and twin property).

Tools: -

- Looping (for loop): To iterate through numbers from 2 up to the given limit.
- Conditional Statements (if): To check whether both numbers in the pair are prime.
- Mathematical Operations: Square root calculation using n**0.5 for efficient prime checking.
- List Data Structure: To store and return the list of twin prime pairs.
- Function Definition: is_prime(n) for prime checking. twin_primes(limit) for generating twin primes.

BRIEF DESCRIPTION:

This program is designed to generate all twin prime pairs up to a given limit. A twin prime is a pair of prime numbers that differ by exactly 2, such as (3, 5) or (11, 13). The program works by first defining a helper function is_prime(n) to check if a number is prime. Then, in the main function twin_primes(limit), it loops through all numbers up to the given limit and checks whether both the current number i and i + 2 are prime. If they are, the pair is added to a list of twin primes. Finally, it returns the list of all twin prime pairs within the range specified by the user.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

1. Understanding the Concept of Twin Primes: Many students initially get confused between prime numbers and twin prime pairs. They may not realize that twin primes must differ by exactly

2. Writing the Prime Check Function: Implementing an efficient is_prime () function can be tricky. Students sometimes forget to check divisibility only up to the square root of n, leading to slower or incorrect results.

3. Loop Range Confusion: Some students struggle with setting the correct loop range (limit - 1 instead of limit) and may accidentally go out of bounds when checking i + 2.

4. Logical Errors in Pair Formation: It's common to mistakenly print or store only one number instead of a pair (i, i + 2).

5. Handling Small Limits: When the limit is too small (like below 5), students might not handle the case properly and expect output when no twin primes exist.

SKILLS ACHIEVED:

- Students strengthen their understanding of prime numbers and their properties.
- Developing the twin_primes () function helps improve logical reasoning and structured problem-solving skills.
- Students learn how to define and use multiple functions (is_prime () and twin_primes ()) to break down a problem into smaller parts.
- Practice in using loops and conditional statements effectively for numerical computations.

Practical No: 15

Date: 09/11/25

TITLE: Program to find number of positive divisors a number have.

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

METHODOLOGY & TOOL USED:

Methodology: -

1. Mathematical Logic: Every divisor i less than or equal to √n has a corresponding divisor n // i. So, for each divisor found, we count both i and n//i.

2. Optimization: Instead of looping up to n, we only loop up to √n to make the function more efficient.

3. Condition for Perfect Squares: If n is a perfect square (e.g., 16 → 4×4), we count the divisor only once.

4. Iteration and Counting: We increment the count for each valid divisor pair.

Tools: -

Programming Language: Python

Concepts Used:

- for loop for iteration
- % (modulus) operator for divisibility check
- **0.5 for square root calculation
- Conditional statements to handle perfect squares

BRIEF DESCRIPTION:

This program defines a function count divisors(n) that counts how many positive integers divide n completely. It efficiently calculates the total number of divisors by checking up to the square root of n, counting each divisor pair, and handling perfect squares carefully , Instead of looping up to n, we only loop up to √n to make the function more efficient.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

1. Forgetting to include both i and n//i as divisors.
2. Not handling perfect squares correctly (double-counting one divisor).
3. Looping all the way up to n instead of √n, making the code inefficient.
4. Confusion between factors and divisors (they are the same in this context).

SKILLS ACHIEVED:

- Understanding of divisors and factorization.
- Use of mathematical optimization (√n approach).
- Strengthened problem-solving and loop control skills.
- Ability to design efficient and accurate mathematical functions in Python.

Practical No: 16

Date: 16/10/2025

TITLE: Aliquot Sum of a Number

AIM/OBJECTIVE(s):

To implement a function that returns the sum of all proper divisors of a given number n, excluding n itself. The aliquot sum is a fundamental concept in number theory, and it has numerous applications in mathematics, computer science, and cryptography.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to iterate over all numbers from 1 to n-1 and check if they are divisors of n. This approach is based on the definition of proper divisors and is widely used in number theory.

- The function is simple and easy to implement.

- It uses a straightforward approach to calculate the aliquot sum.

BRIEF DESCRIPTION:

The aliquot sum(n) function returns the sum of all proper divisors of a given number n, excluding n itself. This function is useful in number theory and can be used to study the properties of perfect numbers, amicable numbers, and other related concepts. The aliquot sum is also used in cryptography and coding theory.

CODE:

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

n = int(input("Enter the number: "))
print(f"Aliquot sum of {n} is {aliquot_sum(n)}")
```

RESULT:

```
Enter the number: 34
Aliquot sum of 34 is 20
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the aliquot sum function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of proper divisors and how to apply it to the problem. The student had to use a combination of arithmetic operations and conditional statements to iterate over all numbers and check if they are divisors of n.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of aliquot sums and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts

Practical No: 17

Date: 16/11/2025

TITLE: Amicable Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if two given numbers are amicable, i.e., the sum of proper divisors of one number equals the other number and vice versa. Amicable numbers have numerous applications in number theory, cryptography, and computer science.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to calculate the sum of proper divisors of both numbers and check if they are equal to each other.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if two numbers are amicable.

BRIEF DESCRIPTION:

The are_amicable(a, b) function checks if two given numbers a and b are amicable. This function is useful in number theory and can be used to study the properties of amicable numbers. Amicable numbers are pairs of numbers that are equal to the sum of the proper divisors of the other number.

1. Number theory: Amicable numbers are used to study the properties of integers and prime numbers.

2. Cryptography: Amicable numbers are used in cryptography and coding theory.

3. Computer science: Amicable numbers are used in algorithms and data structures.

CODE:

```
def are_amicable(a: int, b: int) -> bool:
    if a < 1 or b < 1:
        raise ValueError("Inputs must be positive integers.")
    return aliquot_sum(a) == b and aliquot_sum(b) == a

a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
print(f"{a} and {b} are {'amicable' if are_amicable(a, b) else 'not amicable'}.")
```

RESULT:

```
Enter the first number: 3
Enter the second number: 4
3 and 4 are not amicable.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the amicable numbers function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of amicable numbers and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of amicable numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 18

Date: 16/11/2025

TITLE: Multiplicative Persistence of a Number

AIM/OBJECTIVE(s):

To implement a function that counts the number of steps until a number's digits multiply to a single digit. This function is useful in number theory and can be used to study the properties of multiplicative persistence. Multiplicative persistence is a concept in number theory that has numerous applications in mathematics, computer science, and cryptography.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to multiply the digits of the number and count the number of steps until a single digit is obtained.

- The function is simple and easy to implement.

- It uses a straightforward approach to count the number of steps until a single digit is obtained.

BRIEF DESCRIPTION:

The multiplicative persistence(n) function counts the number of steps until a number's digits multiply to a single digit. This function is useful in number theory and can be used to study the properties of multiplicative persistence.

Multiplicative persistence is a concept in number theory that has numerous applications in mathematics, computer science, and cryptography.

1. Number theory: Multiplicative persistence is used to study the properties of integers and prime numbers.

2. Cryptography: Multiplicative persistence is used in cryptography and coding theory.

CODE:

```
def multiplicative_persistence(n: int) -> int:
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    steps = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

num = int(input("Enter a non-negative integer: "))
print(f"Multiplicative persistence: {multiplicative_persistence(num)}")
```

RESULT:

```
Enter a non-negative integer: 34
Multiplicative persistence: 2
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the multiplicative persistence function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of multiplicative persistence and how to apply it to the problem. The student had to use a combination of arithmetic operations and conditional statements to multiply the digits and count the number of steps.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of multiplicative persistence and its applications.

1. Python programming

2. Problem-solving

3. Attention to detail

4. Number theory concepts

Practical No: 19

Date: 16/11/2025

TITLE: Highly Composite Numbers

AIM/OBJECTIVE(s):

To implement a function that checks if a given number is highly composite, i.e., it has more divisors than any smaller number. Highly composite numbers have numerous applications in number theory, cryptography, and computer science.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to count the number of divisors of the given number and compare it with the number of divisors of smaller numbers.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is highly composite.

- The function is useful in various applications.

BRIEF DESCRIPTION:

The is_highly_composite(n) function checks if a given number n is highly composite. This function is useful in number theory and can be used to study the properties of highly composite numbers. Highly composite numbers are positive integers that have more divisors than any smaller positive integer.

1. Cryptography: Highly composite numbers are used in cryptography and coding theory.

2. Computer science: Highly composite numbers are used in algorithms and data structures.

CODE:

```python
def count_divisors(n: int) -> int:
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            if n // i == i:
                count += 1
            else:
                count += 2
    return count

def is_highly_composite(n: int) -> bool:
    if n < 1:
        raise ValueError("Input must be a positive integer.")
    max_divisors = 0
    for i in range(1, n):
        max_divisors = max(max_divisors, count_divisors(i))
    return count_divisors(n) > max_divisors

num = int(input("Enter a positive integer: "))
print(f"{num} is {'highly composite' if is_highly_composite(num) else 'not highly composite'}.")
```

RESULT:

```
Enter a positive integer: 34
34 is not highly composite.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the highly composite numbers function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of highly composite numbers and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of highly composite numbers and their applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design

Practical No: 20

Date:16/11/2025

TITLE: Modular Exponentiation

AIM/OBJECTIVE(s):

To implement a function that efficiently calculates (base^exponent) % modulus using modular exponentiation. Modular exponentiation is a fundamental concept in number theory and has numerous applications in cryptography, coding theory, and computer science.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using the property of modular arithmetic that (a*b) % n = ((a % n) * (b % n)) % n to efficiently calculate the modular exponentiation.

- The function is efficient and can handle large inputs.

- It uses a straightforward approach to calculate the modular exponentiation

BRIEF DESCRIPTION:

The mod_exp(base, exponent, modulus) function calculates (base^exponent) % modulus using modular exponentiation. This function is useful in number theory and has numerous applications in cryptography, coding theory, and computer science.

1. Cryptography: Modular exponentiation is used in various cryptographic algorithms, such as RSA and Diffie-Hellman key exchange.

2. Coding theory: Modular exponentiation is used in coding theory, such as error-correcting codes.

3. Computer science: Modular exponentiation is used in algorithms and data structures.

CODE:

```python
def mod_exp(base: int, exponent: int, modulus: int) -> int:
    if modulus < 1:
        raise ValueError("Modulus must be a positive integer.")
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent >> 1
        base = (base * base) % modulus
    return result

base = int(input("Enter the base: "))
exponent = int(input("Enter the exponent: "))
modulus = int(input("Enter the modulus: "))
print(f"({base}^{exponent}) % {modulus} = {mod_exp(base, exponent, modulus)}")
```

RESULT:

```
Enter the base: 3
Enter the exponent: 4
Enter the modulus: 34
(3^4) % 34 = 13
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the modular exponentiation function was a challenging task. The student had to carefully consider the property of modular arithmetic and optimize it for efficiency. The student also faced difficulties in understanding the concept of modular exponentiation and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of modular exponentiation and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts

Practical No: 21

Date: 16/11/2025

TITLE: Modular Multiplicative Inverse

AIM/OBJECTIVE(s):

To implement a function that finds the modular multiplicative inverse of a given number a modulo m, i.e., the number x such that (a * x) ≡ 1 mod m. The modular multiplicative inverse is a fundamental concept in number theory .

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using the Extended Euclidean Algorithm to find the modular multiplicative inverse.

- The function is efficient and can handle large inputs.

- It uses a straightforward approach to calculate the modular multiplicative inverse.

- The function is useful in various applications, including cryptography, coding theory, and computer science.

BRIEF DESCRIPTION:

The mod_inverse(a, m) function finds the modular multiplicative inverse of a given number a modulo m. This function is useful in number theory and has numerous applications in cryptography, coding theory, and computer science.

1. Cryptography: Modular multiplicative inverse is used in various cryptographic algorithms, such as RSA and Diffie-Hellman key exchange.

2. Coding theory: Modular multiplicative inverse is used in coding theory, such as error-correcting codes.

CODE:

```python
def mod_inverse(a: int, m: int) -> int:
    def extended_gcd(a: int, b: int) -> tuple:
        if a == 0:
            return b, 0, 1
        else:
            gcd, x, y = extended_gcd(b % a, a)
            return gcd, y - (b // a) * x, x

    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist.")
    return x % m

a = int(input("Enter the number: "))
m = int(input("Enter the modulus: "))
print(f"Modular inverse of {a} mod {m} = {mod_inverse(a, m)}")
```

RESULT:

```
Enter the number: 3
Enter the modulus: 4
Modular inverse of 3 mod 4 = 3
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the modular multiplicative inverse function was a challenging task. The student had to carefully consider the Extended Euclidean Algorithm and optimize it for efficiency. The student also faced difficulties in understanding the concept of modular multiplicative inverse and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of modular multiplicative inverse and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail

Practical No: 22

Date: 16/11/2025

TITLE: Chinese Remainder Theorem Solver

AIM/OBJECTIVE(s):

To implement a function that solves a system of congruences $x \equiv r_i \bmod m_i$ using the Chinese Remainder Theorem (CRT). The CRT is a fundamental concept in number theory and has numerous applications in cryptography, coding theory, and computer science.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using the CRT to solve the system of congruences.

- The function is efficient and can handle large inputs.

- It uses a straightforward approach to solve the system of congruences.

- The function is useful in various applications, including cryptography, coding theory, and computer science.

BRIEF DESCRIPTION:

The crt(remainders, moduli) function solves a system of congruences $x \equiv r_i \bmod m_i$ using the CRT. This function is useful in number theory and has numerous applications in cryptography, coding theory, and computer science.

1. Cryptography: CRT is used in various cryptographic algorithms, such as RSA and Diffie-Hellman key exchange.

2. Coding theory: CRT is used in coding theory, such as error-correcting codes.

3. Computer science: CRT is used in algorithms and data structures.

CODE:

```python
from functools import reduce

def crt(remainders, moduli):
    def mul_inv(a, b):
        b0 = b; x0, x1 = 0, 1
        if b == 1: return 1
        while a > 1:
            q = a // b
            a, b = b, a%b
            x0, x1 = x1 - q * x0, x0
        return x1 + b0 if x1 < 0 else x1

    M = reduce(lambda x, y: x*y, moduli)
    return sum(r * M//m * mul_inv(M//m, m) for r, m in zip(remainders, moduli)) % M

remainders = list(map(int, input("Enter the remainders (space-separated): ").split()))
moduli = list(map(int, input("Enter the moduli (space-separated): ").split()))
print(f"Solution to the system of congruences: x ≡ {crt(remainders,moduli)}")
```

RESULT:

```
Enter the remainders (space-separated): 3
Enter the moduli (space-separated): 4
Solution to the system of congruences: x ≡ 3
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the CRT solver was a challenging task. The student had to carefully consider the CRT and optimize it for efficiency. The student also faced difficulties in understanding the concept of CRT and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of CRT and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

VIT BHOPAL

Practical No: 23

Date: 16/11/2025

TITLE: Quadratic Residue Check

AIM/OBJECTIVE(s):

To implement a function that checks if $x^2 \equiv a \bmod p$ has a solution, i.e., whether a is a quadratic residue modulo p. The quadratic residue check is a fundamental concept in number theory .

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using Euler's criterion to check if a is a quadratic residue modulo p.

- The function is efficient and can handle large inputs.

- It uses a straightforward approach to check if a is a quadratic residue modulo p.

- The function is useful in various applications, including cryptography.

BRIEF DESCRIPTION:

The is_quadratic_residue(a, p) function checks if $x^2 \equiv a \bmod p$ has a solution, i.e., whether a is a quadratic residue modulo p. This function is useful in number theory and has numerous applications in cryptography, coding theory, and computer science.

1. Cryptography: Quadratic residue is used in various cryptographic algorithms, such as RSA and Diffie-Hellman key exchange.

2. Coding theory: Quadratic residue is used in coding theory, such as error-correcting codes.

CODE:

```python
def is_quadratic_residue(a: int, p: int) -> bool:

    if not isinstance(p, int) or p < 2 or not all(p % i for i in range(2, int(p**0.5) + 1)):
        raise ValueError("p must be a prime number")

    # Euler's criterion
    return pow(a, (p - 1) // 2, p) == 1

a = int(input("Enter the number: "))
p = int(input("Enter the prime modulus: "))
print(f"{a} is {'a quadratic residue' if is_quadratic_residue(a, p) else 'not a quadratic residue'} mod {p}")
```

RESULT:

```
Enter the number: 3
Enter the prime modulus: 5
3 is not a quadratic residue mod 5
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the quadratic residue check was a challenging task. The student had to carefully consider Euler's criterion and optimize it for efficiency. The student also faced difficulties in understanding the concept of quadratic residue and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of quadratic residue and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 24

Date: 16/11/2025

TITLE: Order Modulo n

AIM/OBJECTIVE(s):

To implement a function that finds the smallest positive integer k such that $a^k \equiv 1 \bmod n$, i.e., the order of a modulo n. The order of an element is a fundamental concept in number theory.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to find the smallest positive integer k such that $a^k \equiv 1 \bmod n$.

- The function is simple and easy to implement.

- It uses a straightforward approach to find the order of a modulo n.

- The function is useful in various applications, including cryptography, coding theory, and computer science.

BRIEF DESCRIPTION:

The order_mod(a, n) function finds the smallest positive integer k such that $a^k \equiv 1 \bmod n$, i.e., the order of a modulo n. This function is useful in number theory and has numerous applications in cryptography, coding theory, and computer science.

1. Cryptography: Order modulo n is used in various cryptographic algorithms, such as RSA and Diffie-Hellman key exchange.

2. Coding theory: Order modulo n is used in coding theory, such as error-correcting codes.

CODE:

```python
def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus
    return result

def order_mod(a, n):
    k = 1
    while True:
        if mod_exp(a, k, n) == 1:
            return k
        k += 1

a = int(input("Enter the number: "))
n = int(input("Enter the modulus: "))
print(f"The order of {a} modulo {n} is {order_mod(a, n)}")
```

RESULT:

```
Enter the number: 3
Enter the modulus: 4
The order of 3 modulo 4 is 2
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the order modulo n function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of order modulo n and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of order modulo n and its applications.

1. Python programming
2. Problem-solving

Practical No: 25

Date:16/11/2025

TITLE: Checking if a Number is a Fibonacci Prime

AIM/OBJECTIVE(s):

The primary objective of this project is to develop a function that checks if a given positive integer is both a Fibonacci number and a prime number, thereby classifying it as a Fibonacci prime.

METHODOLOGY & TOOL USED:

The methodology employed involves leveraging a mathematical property to determine if a number is a Fibonacci number, and utilizing trial division to check for primality. Python programming language is used as the tool for implementation, owing to its simplicity and efficiency in handling mathematical operations.

BRIEF DESCRIPTION:

This project entails designing a function is Fibonacci prime(n) that takes an integer n as input and returns a Boolean value indicating whether the number is a Fibonacci prime or not. The function utilizes two helper functions, is Fibonacci(n) and is prime(n), to check for Fibonacci and prime properties respectively

CODE:

```python
import math

def is_perfect_square(x):
    s = int(math.sqrt(x))
    return s*s == x

def is_fibonacci(n):
    return is_perfect_square(5*n*n + 4) or is_perfect_square(5*n*n - 4)

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

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

num = int(input("Enter a positive integer: "))
print(f"{num} is {'a' if is_fibonacci_prime(num) else 'not a'} Fibonacci prime number.")
```

RESULTS ACHIEVED:

```
Enter a positive integer: 233
233 is a Fibonacci prime number.
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\pythonnnnn>
```

DIFFICULTY FACED BY STUDENT:

The student encountered challenges in implementing the mathematical property to identify Fibonacci numbers, optimizing the prime checking algorithm to handle larger inputs, and integrating the two checks into a single function.

SKILLS ACHIEVED:

The developed function is Fibonacci prime(n) successfully determines whether a given number is a Fibonacci prime, showcasing the application of mathematical concepts, programming techniques, and

problem-solving strategies to achieve an efficient solution. This project demonstrates the importance of breaking down complex problems into manageable components and leveraging mathematical properties to optimize computational efficiency.

Practical No: 26

Data:20/10/2025

TITLE: Generating Lucas Numbers Sequence

AIM/OBJECTIVE(s):

The objective of this project is to develop a function that generates the first n Lucas numbers, a sequence similar to Fibonacci numbers but starting with 2 and 1, and explore its properties and applications.

METHODOLOGY & TOOL USED:

The methodology employed involves using an iterative approach to generate the Lucas sequence, leveraging the recurrence relation that each number is the sum of the two preceding ones, and utilizing Python programming language as the tool for implementation due to its simplicity and efficiency.

BRIEF DESCRIPTION:

This project involves designing a function Lucas sequence(n) that generates the first n Lucas numbers, providing a sequence that starts with 2, 1 and continues with the sum of the previous two numbers, and discussing its relation to other number sequences and potential applications in mathematics and computer science.

CODE:

```python
def lucas_sequence(n):
    sequence = [2, 1]
    while len(sequence) < n:
        sequence.append(sequence[-1] + sequence[-2])
    return sequence[:n]

num = int(input("Enter the number of Lucas numbers to generate: "))
print(lucas_sequence(num))
```

RESULTS ACHIEVED:

```
Enter the number of Lucas numbers to generate: 12
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\pythonnnnn>
```

DIFFICULTY FACED BY STUDENT:

The student encountered challenges in implementing the iterative approach, handling edge cases for small values of n, ensuring the sequence generation was accurate and efficient, and exploring the properties and relationships of Lucas numbers with other mathematical concepts, which required additional research and understanding.

SKILLS ACHIEVED:

The developed function Lucas sequence(n) successfully generates the first n Lucas numbers, demonstrating the application of iterative techniques and mathematical concepts to solve a problem efficiently, and highlighting the importance of exploring and understanding the properties and relationships of mathematical sequences.

Practical No: 27

Date: 20/11/2025

TITLE: Perfect Powers Check

AIM/OBJECTIVE(s):

To implement a function that checks if a number can be expressed as a^b where a > 0 and b > 1, i.e., whether a number is a perfect power. Perfect powers are a fundamental concept in number theory and have numerous.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to check if a number can be expressed as a perfect power.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is a perfect power.

- The function is useful in various applications, including mathematics, computer science, and cryptography.

BRIEF DESCRIPTION:

The is perfect_power(n) function checks if a number can be expressed as a^b where a > 0 and b > 1, i.e., whether a number is a perfect power. This function is useful in number theory and has numerous applications in mathematics, computer science, and cryptography.

1. Mathematics: Perfect power is used to study the properties of integers and prime numbers.

2. Computer science: Perfect power is used in algorithms and data structures.

3. Cryptography: Perfect power is used in cryptographic algorithms and protocols.

CODE:

```python
def is_quadratic_residue(a: int, p: int) -> bool:

    if not isinstance(p, int) or p < 2 or not all(p % i for i in range(2, int(p**0.5) + 1)):
        raise ValueError("p must be a prime number")

    # Euler's criterion
    return pow(a, (p - 1) // 2, p) == 1

a = int(input("Enter the number: "))
p = int(input("Enter the prime modulus: "))
print(f"{a} is {'a quadratic residue' if is_quadratic_residue(a, p) else 'not a quadratic residue'} mod {p}")
```

RESULT:

```
Enter the number: 34
34 is not a perfect power
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the perfect power check was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of perfect power and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of perfect power and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 28

Date: 20/11/2025

TITLE: Collatz Sequence Length

AIM/OBJECTIVE(s):

To implement a function that returns the number of steps for n to reach 1 in the Collatz conjecture. The Collatz conjecture is a fundamental concept in number theory.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to generate the Collatz sequence and count the number of steps.

- The function is simple and easy to implement.

- It uses a straightforward approach to generate the Collatz sequence and count the number of steps.

- The function is useful in various applications, including mathematics.

BRIEF DESCRIPTION:

The collatz_length(n) function returns the number of steps for n to reach 1 in the Collatz conjecture. This function is useful in number theory and has numerous applications in mathematics, computer science, and cryptography.

1. Mathematics: Collatz conjecture is used to study the properties of integers and prime numbers.

2. Computer science: Collatz conjecture is used in algorithms and data structures.

3. Cryptography: Collatz conjecture is used in cryptographic algorithms and protocols.

CODE:

```python
def collatz_length(n: int) -> int:

    if n < 1:
        raise ValueError("Input must be a positive integer.")

    length = 1
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        length += 1

    return length

n = int(input("Enter the starting number: "))
print(f"The Collatz sequence length for {n} is {collatz_length(n)}")
```

RESULT:

```
Enter the starting number: 34
The Collatz sequence length for 34 is 14
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the Collatz sequence length was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of Collatz conjecture and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of Collatz conjecture and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts

Practical No: 29

Date: 23/11/2025

TITLE: Polygonal Numbers

AIM/OBJECTIVE(s):

To implement a function that returns the n-th s-gonal number. Polygonal numbers are a fundamental concept in number theory and have numerous applications in mathematics, computer science, and cryptography.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple formula to calculate the n-th s-gonal number.

- The function is simple and easy to implement.

- It uses a straightforward approach to calculate the n-th s-gonal number.

- The function is useful in various applications, including mathematics, computer science, and cryptography.

BRIEF DESCRIPTION:

The polygonal_number(s, n) function returns the n-th s-gonal number. This function is useful in number theory and has numerous applications in mathematics, computer science, and cryptography.

1. Mathematics: Polygonal numbers are used to study the properties of integers and prime numbers.

2. Computer science: Polygonal numbers are used in algorithms and data structures.

3. Cryptography: Polygonal numbers are used in cryptographic algorithms and protocols.

CODE:

```python
def polygonal_number(s: int, n: int) -> int:

    if s < 3:
        raise ValueError("Input s must be an integer greater than 2.")
    if n < 1:
        raise ValueError("Input n must be a positive integer.")

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

s = int(input("Enter the number of sides of the polygon: "))
n = int(input("Enter the position of the polygonal number: "))
print(f"The {n}-th {s}-gonal number is {polygonal_number(s, n)}")
```

RESULT:

```
Enter the number of sides of the polygon: 3
Enter the position of the polygonal number: 4
The 4-th 3-gonal number is 10
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the polygonal number function was a challenging task. The student had to carefully consider the formula and optimize it for efficiency. The student also faced difficulties in understanding the concept of polygonal numbers and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of polygonal numbers and its applications

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 30

Date: 23/11/2025

TITLE: Carmichael Number Check

AIM/OBJECTIVE(s):

To implement a function that checks if a composite number n satisfies a^(n-1) ≡ 1 mod n for all a coprime to n, i.e., whether a number is a Carmichael number. Carmichael numbers are a fundamental concept in number theory and have numerous applications in cryptography and computer science.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the function. The methodology involves using a simple iterative approach to check if a number is a Carmichael number.

- The function is simple and easy to implement.

- It uses a straightforward approach to check if a number is a Carmichael number.

- The function is useful in various applications, including cryptography and computer science.

BRIEF DESCRIPTION:

The is carmichael(n) function checks if a composite number n satisfies a^(n-1) ≡ 1 mod n for all a coprime to n, i.e., whether a number is a Carmichael number. This function is useful in number theory and has numerous applications in cryptography and computer science.

1. Cryptography: Carmichael numbers are used in cryptographic algorithms and protocols.

2. Computer science: Carmichael numbers are used in algorithms and data structures.

3. Number theory: Carmichael numbers are used to study the properties of integers and prime numbers.

CODE:

```python
import math

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

def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

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

n = int(input("Enter the number: "))
print(f"{n} is {'a Carmichael number' if is_carmichael(n) else 'not a Carmichael number'}")
```

RESULT:

```
Enter the number: 34
34 is not a Carmichael number
PS C:\Users\Jagan Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the Carmichael number check was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of Carmichael numbers and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of Carmichael numbers and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design

VIT BHOPAL

Practical No: 31

Date: 23/11/2025

TITLE: Probabilistic Miller-Rabin Primality Test

AIM/OBJECTIVE(s):

To implement the probabilistic Miller-Rabin primality test to determine whether a given number is prime or composite. The Miller-Rabin test is a widely used algorithm for primality testing, and it is essential in many cryptographic applications.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the Miller-Rabin test. The methodology involves using a probabilistic approach to test whether a number is prime or composite.

- The Miller-Rabin test is a widely used algorithm for primality testing.

- It is a probabilistic test, which means it can be faster than deterministic tests for large numbers.

- The test is useful in many cryptographic applications.

BRIEF DESCRIPTION:

The is prime_miller_rabin(n, k) function implements the probabilistic Miller-Rabin primality test to determine whether a given number n is prime or composite. The function takes two arguments: n, the number to be tested, and k, the number of rounds to perform the test.

1. Cryptography: The Miller-Rabin test is used in many cryptographic algorithms and protocols.

2. Computer science: The Miller-Rabin test is used in algorithms and data structures.

3. Number theory: The Miller-Rabin test is used to study the properties of integers and prime numbers.

CODE:

```python
def is_prime_miller_rabin(n, k):
    def check(a, s, d, n):
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            return True
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                return True
        return False

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

    for _ in range(k):
        a = random.randint(2, n - 1)
        if not check(a, s, d, n):
            return False
    return True

n = int(input("Enter the number: "))
k = int(input("Enter the number of rounds: "))
print(f"{n} is {'probably prime' if is_prime_miller_rabin(n, k) else 'composite'}")
```

RESULT:

```
Enter the number: 3
Enter the number of rounds: 4
3 is probably prime
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the Miller-Rabin test was a challenging task. The student had to carefully consider the probabilistic approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of the Miller-Rabin test and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of the Miller-Rabin test and its applications.

Practical No: 32

Date: 23/11/2025

TITLE: Pollard's Rho Algorithm for Integer Factorization

AIM/OBJECTIVE(s):

To implement Pollard's rho algorithm for integer factorization, which is a popular algorithm for finding non-trivial factors of a composite number.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement Pollard's rho algorithm. The methodology involves using a iterative approach to find a non-trivial factor of the input number.

- Pollard's rho algorithm is a popular algorithm for integer factorization.

- It is relatively simple to implement and has a good performance for small to medium-sized inputs.

- The algorithm is useful in many cryptographic applications.

BRIEF DESCRIPTION:

The pollard rho(n) function implements Pollard's rho algorithm to find a non-trivial factor of the input number n. The function uses a iterative approach to find a cycle in the sequence generated by the function f(x) = (x^2 + 1) mod n, and then uses the cycle to find a non-trivial factor of n.

1. Cryptography: Pollard's rho algorithm is used in many cryptographic algorithms and protocols.

2. Computer science: Pollard's rho algorithm is used in algorithms and data structures.

3. Number theory: Pollard's rho algorithm is used to study the properties of integers and prime numbers.

CODE:

```python
import random

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

def pollard_rho(n):
    if n % 2 == 0:
        return 2
    x = random.randint(1, n - 1)
    y = x
    c = random.randint(1, n - 1)
    g = 1
    while g == 1:
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        g = gcd(abs(x - y), n)
    if g == n:
        return none
    return g

n = int(input("enter the number: "))
factor = pollard_rho(n)
if factor:
    print(f"{n} = {factor} * {n // factor}")
else:
    print("failed to find a factor")
```

RESULT:

```
enter the number: 34
34 = 2 * 17
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing Pollard's rho algorithm was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of Pollard's rho algorithm and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of Pollard's rho algorithm and its applications.

Practical No: 33

Date:23/11/2025

TITLE: Approximation of the Riemann Zeta Function

AIM/OBJECTIVE(s):

To implement a function that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the approximation of the Riemann zeta function. The methodology involves using a simple iterative approach to sum the first 'terms' of the series.

- The function is simple and easy to implement.

- It uses a straightforward approach to approximate the Riemann zeta function.

- The function is useful in various applications, including mathematics, computer science, and cryptography.

BRIEF DESCRIPTION:

The zeta approx(s, terms) function approximates the Riemann zeta function ζ(s) using the first 'terms' of the series. The Riemann zeta function is a fundamental function in number theory and has numerous applications in mathematics, computer science, and cryptography.

1. Mathematics: The Riemann zeta function is used to study the properties of integers and prime numbers.

2. Computer science: The Riemann zeta function is used in algorithms and data structures.

3. Cryptography: The Riemann zeta function is used in cryptographic algorithms and protocols.

CODE:

```
def zeta_approx(s, terms):
    zeta = 0
    for n in range(1, terms + 1):
        zeta += 1 / (n ** s)
    return zeta

s = float(input("enter the value of s: "))
terms = int(input("enter the number of terms: "))
print(f"ζ({s}) ≈ {zeta_approx(s, terms)}")
```

RESULT:

```
enter the value of s: 3
enter the number of terms: 4
ζ(3.0) ≈ 1.177662037037037
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the approximation of the Riemann zeta function was a challenging task. The student had to carefully consider the iterative approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of the Riemann zeta function and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of the Riemann zeta function and its applications.

1. Python programming
2. Problem-solving
3. Attention to detail
4. Number theory concepts
5. Algorithm design
6. Critical thinking

Practical No: 34

Date: 23/11/2025

TITLE: Partition Function

AIM/OBJECTIVE(s):

To implement a function that calculates the number of distinct ways to write n as a sum of positive integers, also known as the partition function p(n).

METHODOLOGY & TOOL USED:

Python is used as the programming language to implement the partition function. The methodology involves using dynamic programming to calculate the partition function.

TOOL:

- The function is simple and easy to implement.

- It uses a straightforward approach to calculate the partition function.

- The function is useful in various applications, including mathematics, computer science, and cryptography.

BRIEF DESCRIPTION:

The partition_function(n) function calculates the number of distinct ways to write n as a sum of positive integers. This function is useful in number theory and has numerous applications in mathematics, computer science, and cryptography.

1. Mathematics: The partition function is used to study the properties of integers and prime numbers.

2. Computer science: The partition function is used in algorithms and data structures.

3. Cryptography: The partition function is used in cryptographic algorithms and protocols.

CODE:

```python
def partition_function(n):
    partitions = [0] * (n + 1)
    partitions[0] = 1
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            partitions[j] += partitions[j - i]
    return partitions[n]

n = int(input("Enter the number: "))
print(f"The number of partitions of {n} is {partition_function(n)}")
```

RESULT:

```
Enter the number: 34
The number of partitions of 34 is 12310
PS C:\Users\Jagan_Thupakula\OneDrive\Desktop\cse assignment>
```

DIFFICULTY FACED BY STUDENT:

Implementing the partition function was a challenging task. The student had to carefully consider the dynamic programming approach and optimize it for efficiency. The student also faced difficulties in understanding the concept of the partition function and how to apply it to the problem.

SKILLS ACHIEVED:

Python programming, problem-solving, attention to detail, and understanding of number theory concepts. The student gained knowledge of the partition function and its applications.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL