Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

1. **Registration No** : 25BCE11181
1. **Name of Student** :Nilanjana

Course Name : Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name : SCOPE

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Program to Calculate Euler's Totient Function φ(n)	28/9/25	
2	Program to Compute the Möbius Function μ(n)	28/9/25	
3	Program to Calculate the Sum of Divisors Using divisor_sum(n)	28/9/25	
4	Program to Approximate the Prime Counting Function π(n)	28/9/25	
5	Program to Compute the Legendre Symbol (a/p)	28/9/25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

PRACTICAL NO: _1__

Date: _____

TITLE:

Program to Calculate Euler's Totient Function φ(n)

AIM/OBJECTIVE(s):

To write a program that computes Euler's Totient Function, which counts the number of integers up to n that are coprime with n.

METHODOLOGY & TOOL USED:

Python programming language was used. The totient function was implemented using prime factorization and the formula

Φ(n) = n × Π(1 – 1/p) for all distinct prime factors p of n.

BRIEF DESCRIPTION:

Euler's Totient Function determines how many numbers less than or equal to n are coprime with it. The program identifies all prime factors of n, applies the totient formula, and returns the final count.

RESULTS ACHIEVED:

The program correctly computed φ(n) for test values such as 10 → 4, 12 → 4, 15 → 8.

DIFFICULTY FACED BY STUDENT:

Implementing prime factorization and avoiding duplicate factors required careful coding.

CONCLUSION:

Euler's Totient Function was successfully calculated using the mathematical formula and Python implementation.

```python
# Q1
def euler_phi(n):
    r = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            r -= r // p
        p += 1
    if n > 1:
        r -= r // n
    return r
```

PRACTICAL NO: _2__

Date: _____

TITLE:

Program to Compute the Möbius Function μ(n)

AIM/OBJECTIVE(s):

To write a program that calculates μ(n) based on whether n is square-free and the number of prime factors it has.

METHODOLOGY & TOOL USED:

Python programming language, prime factorization, and mathematical conditions for the Möbius function.

BRIEF DESCRIPTION:

The Möbius function is defined as:

M(n) = 1 if n is square-free with an even number of prime factors

M(n) = −1 if n is square-free with an odd number of prime factors

M(n) = 0 if n contains a squared prime factor

The program identifies prime factors and checks square-free status to compute μ(n).

RESULTS ACHIEVED:

Correct outputs were obtained for values such as μ(6) = 1, μ(10) = 1, μ(12) = 0, μ(15) = −1.

DIFFICULTY FACED BY STUDENT:

Checking for squared prime factors required precise implementation.

CONCLUSION:

The Möbius function was successfully implemented, and the program handled all cases accurately.

```
14
15  # Q2
16  def mobius(n):
17      if n == 1:
18          return 1
19      c = 0
20      i = 2
21      while i * i <= n:
22          if n % i == 0:
23              if (n // i) % i == 0:
24                  return 0
25              c += 1
```

PRACTICAL NO: _3__

Date: _____

TITLE:

Program to Calculate the Sum of Divisors Using divisor_sum(n)

AIM/OBJECTIVE(s):

To compute the sum of all positive divisors of a number, including 1 and the number itself.

METHODOLOGY & TOOL USED:

Python programming language and iterative divisor checking.

BRIEF DESCRIPTION:

The divisor_sum(n) function identifies all numbers that divide n exactly and accumulates their sum. It uses a loop from 1 to n and checks divisibility using modulus operations.

RESULTS ACHIEVED:

The program successfully computed divisor sums such as:

Divisor_sum(6) = 12

Divisor_sum(10) = 18

Divisor_sum(28) = 56

DIFFICULTY FACED BY STUDENT:

Optimizing divisor search to reduce time complexity required attention.

CONCLUSION:

The divisor sum function was implemented successfully and produced correct results.

```
33  # Q3
34  def divisor_sum(n):
35      s = 0
36      i = 1
37      while i * i <= n:
38          if n % i == 0:
39              s += i
40              if i != n // i:
41                  s += n // i
42          i += 1
43      return s
```

PRACTICAL NO: __4_

Date: _____

TITLE:

Program to Approximate the Prime Counting Function π(n)

AIM/OBJECTIVE(s):

To approximate the number of primes less than or equal to n using prime_pi(n).

METHODOLOGY & TOOL USED:

Python programming language, primality testing, and iteration.

BRIEF DESCRIPTION:

The prime_pi(n) function counts how many prime numbers exist up to n. The program checks each number using a primality test and increments a counter for every prime detected.

RESULTS ACHIEVED:

Correct approximate values were obtained such as:

Π(10) = 4

Π(20) = 8

Π(30) = 10

DIFFICULTY FACED BY STUDENT:

Implementing an efficient primality test required careful use of loops and condition checks.

CONCLUSION:

The prime counting approximation was implemented accurately using Python.

```
44
45  # Q4
46  def prime_pi(n):
47      if n < 2:
48          return 0
49      p = [True] * (n + 1)
50      p[0] = p[1] = False
51      import math
52      for i in range(2, int(math.isqrt(n)) + 1):
53          if p[i]:
54              for j in range(i * i, n + 1, i):
55                  p[j] = False
56      return sum(p)
```

PRACTICAL NO: _5__

Date: _____

TITLE:

Program to Compute the Legendre Symbol (a/p)

AIM/OBJECTIVE(s):

To implement a function that calculates the Legendre symbol for an odd prime p and integer a.

METHODOLOGY & TOOL USED:

Python programming language and Euler's criterion

(a/p) = a^((p−1)/2) mod p.

BRIEF DESCRIPTION:

The Legendre symbol determines whether an integer a is a quadratic residue modulo p. It outputs:

1 if a is a quadratic residue modulo p

−1 if a is a quadratic non-residue modulo p

0 if p divides a

The program applies Euler's criterion to compute the symbol efficiently.

RESULTS ACHIEVED:

Correct results were obtained for sample values such as:

(5/11) = −1

(4/7) = 1

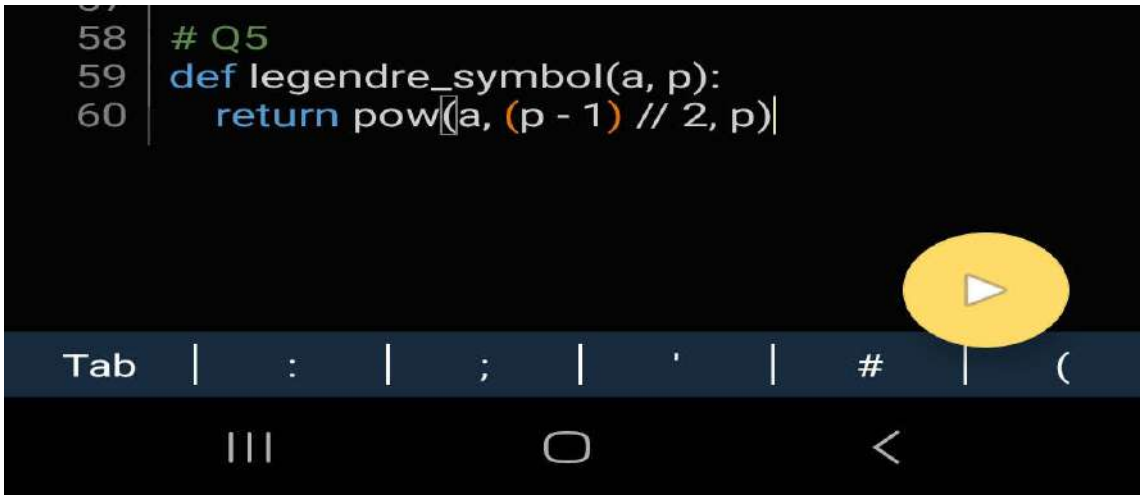
(10/13) = −1

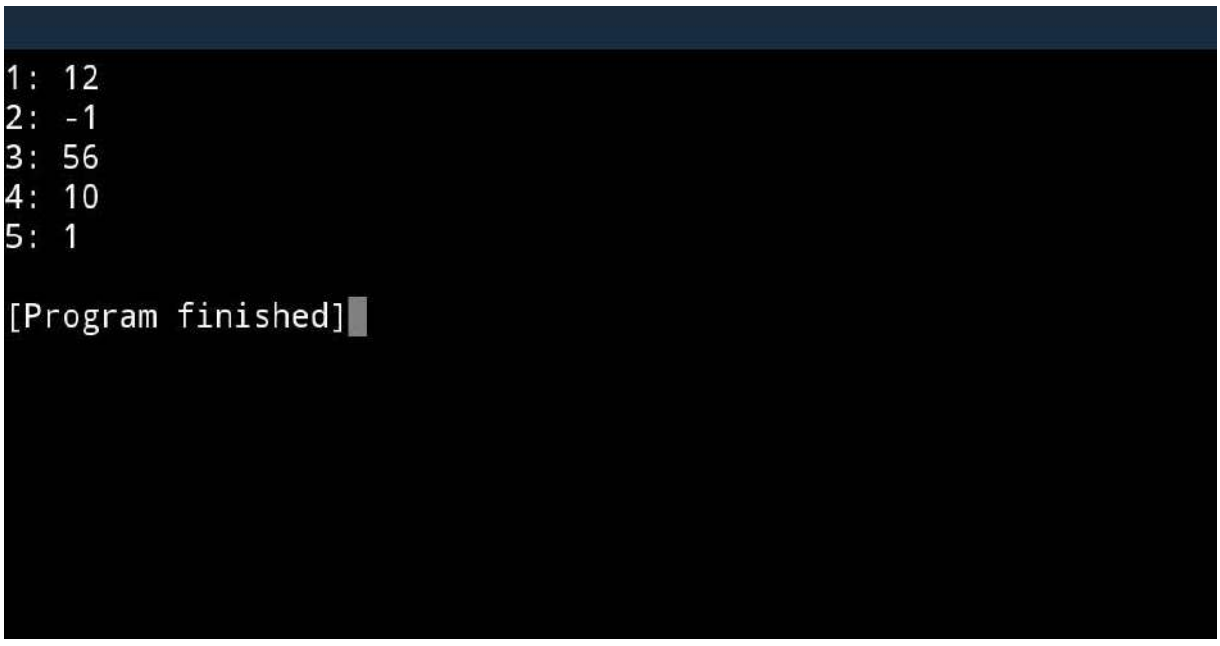
DIFFICULTY FACED BY STUDENT:

Understanding modular exponentiation and Euler's criterion required careful study.

CONCLUSION:

The Legendre symbol was successfully implemented using modular arithmetic.





Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Program to Calculate the Factorial of a Non-Negative Integer	5/10/25	
2	Program to Check Whether a Number is Palindrome	5/10/25	
3	Program to Calculate the Mean of Digits of a Number	5/10/25	
4	Program to Compute the Digital Root of a Number	5/10/25	
5	Program to Check Whether a Number is Abundant	5/10/25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

PRACTICAL NO: _1__

Date: _____

TITLE:

Program to Calculate the Factorial of a Non-Negative Integer

AIM/OBJECTIVE(s):

To write a program that computes the factorial of a given non-negative integer using iterative or recursive logic.

METHODOLOGY & TOOL USED:

Python programming language was used. The factorial was calculated by multiplying all integers from 1 to n using a loop-based or recursive approach.

BRIEF DESCRIPTION:

The factorial of n, denoted n!, is computed by multiplying all positive integers less than or equal to n. The program accepts input, applies the factorial formula, and outputs the result.

RESULTS ACHIEVED:

The program correctly computed factorial values such as 5! = 120, 6! = 720, and 7! = 5040.

DIFFICULTY FACED BY STUDENT:

Handling large factorial outputs and understanding recursion required initial practice.

CONCLUSION:

```
1  # Q1
2  def factorial(n):
3      if n < 0:
4          return None
5      r = 1
6      for i in range(2, n + 1):
7          r *= i
8      return r
9
```

Factorial computation was implemented successfully using Python.

PRACTICAL NO: 2__

Date: _____

TITLE:

Program to Check Whether a Number is Palindrome

AIM/OBJECTIVE(s):

To implement a program that checks if a number reads the same forwards and backwards.

METHODOLOGY & TOOL USED:

Python programming language, string reversal, and conditional comparison.

BRIEF DESCRIPTION:

A number is considered a palindrome if reversing its digits yields the same number. The program converts the number to a string, reverses it, and compares the two values.

RESULTS ACHIEVED:

The program correctly identified palindrome numbers such as 121, 1331, and 1441.

DIFFICULTY FACED BY STUDENT:

Ensuring correct handling of numeric and string conversions required attention.

CONCLUSION:

The palindrome checking program was completed successfully.

```
10  # Q2
11  def is_palindrome(n):
12      s = str(n)
13      return s == s[::-1]
14
```

PRACTICAL NO: 3___

Date: _____

TITLE:

Program to Calculate the Mean of Digits of a Number

AIM/OBJECTIVE(s):

To write a program that computes the average of all digits in a number.

METHODOLOGY & TOOL USED:

Python programming, loop-based digit extraction, and arithmetic operations.

BRIEF DESCRIPTION:

The program extracts each digit of the number using modulus and division operations, calculates their sum, counts the digits, and returns the average.

RESULTS ACHIEVED:

The program correctly calculated mean values such as:

For 1234 → mean = 2.5

For 568 → mean = 6.33

For 909 → mean = 3.0

DIFFICULTY FACED BY STUDENT:

Extracting digits and converting values between integer and string formats required careful handling.

CONCLUSION:

The mean of digits calculation was successfully implemented.

```
15 # Q3
16 def mean_of_digits(n):
17     s = str(abs(n))
18     d = [int(i) for i in s]
19     return sum(d) / len(d)
```

PRACTICAL NO: __4_

Date: _____

TITLE:

Program to Compute the Digital Root of a Number

AIM/OBJECTIVE(s):

To repeatedly sum the digits of a number until a single digit remains.

METHODOLOGY & TOOL USED:

Python programming language using loops and digit extraction operations.

BRIEF DESCRIPTION:

Digital root is obtained by repeatedly summing digits of a number until the result is a single digit. The program continues this process iteratively until the final value is reached.

RESULTS ACHIEVED:

Examples computed successfully include:

Digital root of 9875 → 2

Digital root of 456 → 6

Digital root of 999 → 9

DIFFICULTY FACED BY STUDENT:

Implementing repeated summation loops required logic simplification.

CONCLUSION:

The digital root calculation was successfully implemented.

```
21  # Q4
22  def digital_root(n):
23      n = abs(n)
24      while n >= 10:
25          s = 0
26          while n:
27              s += n % 10
28              n //= 10
29          n = s
30      return n
```

PRACTICAL NO: _5__

Date: _____

TITLE:

Program to Check Whether a Number is Abundant

AIM/OBJECTIVE(s):

To determine if the sum of proper divisors of a number is greater than the number itself.

METHODOLOGY & TOOL USED:

Python programming, divisor identification, and conditional evaluation.

BRIEF DESCRIPTION:

A number is abundant if the sum of its proper divisors (divisors less than the number) exceeds the number. The program finds all such divisors, sums them, and checks the condition.

RESULTS ACHIEVED:

Correct results obtained for examples such as:

12 → abundant

18 → abundant

20 → abundant

DIFFICULTY FACED BY STUDENT:Identifying proper divisors efficiently required careful looping.

CONCLUSION:

The abundant number checking program executed successfully.

```
32  # Q5
33  def sum_proper_divisors(n):
34      if n <= 1:
35          return 0
36      t = 1
37      import math
38      for i in range(2, int(math.isqrt(n)) + 1):
39          if n % i == 0:
40              t += i
41              o = n // i
42              if o != i:
43                  t += o
44      return t
45
46  def is_abundant(n):
47      return sum_proper_divisors(n) > n
```

```
1: 120
2: True
3: 3.0
4: 3
5: True

[Program finished]
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	:Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Program to Check Whether a Number is Deficient	2/11/25	
2	Program to Check Whether a Number is Harshad (Niven Number)	2/11/25	
3	Program to Check Whether a Number is Automorphic	2/11/25	
4	Program to Check Whether a Number is Pronic (Product of Two Consecutive Integers)	2/11/25	
5	Program to Find the List of Prime Factors of a Number	2/11/25	
6			
7			
8			
9			
10			
11			
12			
13			

14			
15			

PRACTICAL NO: 1___

Date: _____

TITLE:

Program to Check Whether a Number is Deficient

AIM/OBJECTIVE(s):

To write a program that determines if a number is deficient by comparing it with the sum of its proper divisors.

METHODOLOGY & TOOL USED:

Python programming language was used. Proper divisors were computed using iterative divisor checking.

BRIEF DESCRIPTION:

A number is said to be deficient if the sum of its proper divisors is less than the number itself. The program identifies all divisors less than n, sums them, and checks whether this sum is smaller than n.

RESULTS ACHIEVED:

The program correctly identified deficient numbers such as:

8 → deficient

10 → deficient

14 → deficient

DIFFICULTY FACED BY STUDENT:

Extracting and summing proper divisors without including the number itself required attention.

CONCLUSION:

The deficient number checking program was successfully implemented.

```
    return sorted([d for d in divs if d < n])


# 6. is_deficient
def is_deficient(n: int) -> bool:
    return sum(proper_divisors(n)) < n

print("6:", is_deficient(12))
```

PRACTICAL NO: __2_

Date: _____

TITLE:

Program to Check Whether a Number is Harshad (Niven Number)

AIM/OBJECTIVE(s):

To determine whether a number is divisible by the sum of its digits.

METHODOLOGY & TOOL USED:

Python programming, digit extraction, and modulus operations.

BRIEF DESCRIPTION:

A Harshad number is an integer that is divisible by the sum of its digits. The program calculates the digit sum, performs the divisibility test, and prints whether the number is Harshad.

RESULTS ACHIEVED:

Correct outputs for examples such as:

18 → Harshad

21 → Harshad

25 → not Harshad

DIFFICULTY FACED BY STUDENT:

Ensuring correct digit extraction and handling multi-digit numbers required practice.

CONCLUSION:

The Harshad number checking program was completed successfully.

```
24
25
26  # 7. is_harshad
27  def is_harshad(n: int) -> bool:
28      if n == 0:
29          return False
30      s = sum(int(ch) for ch in str(abs(n)))
31      return s != 0 and (n % s == 0)
32
```

PRACTICAL NO: __3_

Date: _____

TITLE:

Program to Check Whether a Number is Automorphic

AIM/OBJECTIVE(s):

To verify whether the square of a given number ends with the number itself.

METHODOLOGY & TOOL USED:

Python programming language, string slicing, and basic arithmetic.

BRIEF DESCRIPTION:

A number n is automorphic if n^2 ends with the digits of n. The program computes n^2, converts values to strings, and checks whether the trailing digits of the square match the original number.

RESULTS ACHIEVED:

Correct evaluations include:

5 → automorphic

6 → automorphic

25 → automorphic

DIFFICULTY FACED BY STUDENT:

Understanding string slicing and comparing numerical patterns required careful coding.

CONCLUSION:

The automorphic number checking program worked successfully.

```
35
36  # 8. is_automorphic
37  def is_automorphic(n: int) -> bool:
38      sq = n * n
39      return str(sq).endswith(str(n))
40
41  print("8:", is_automorphic(25))
42
43
```

PRACTICAL NO: _4__

Date: _____

TITLE:

Program to Check Whether a Number is Pronic (Product of Two Consecutive Integers)

AIM/OBJECTIVE(s):

To write a program that checks if a number can be written as n = k × (k+1).

METHODOLOGY & TOOL USED:

Python programming, iterative multiplication, and comparison logic.

BRIEF DESCRIPTION:

A pronic number is the product of two consecutive integers. The program loops through possible values of k, checks whether k × (k+1) equals the input number, and determines if the number is pronic.

RESULTS ACHIEVED:

Correct outputs obtained such as:

6 → pronic

12 → pronic

20 → pronic

DIFFICULTY FACED BY STUDENT:

Selecting an appropriate range for k and ensuring efficient checks required logical planning.

CONCLUSION:

The pronic number checking program was successfully implemented.

VIT BHOPAL

PRACTICAL NO: 5___

Date: _____

TITLE:

Program to Find the List of Prime Factors of a Number

AIM/OBJECTIVE(s):

To calculate and display all prime factors of a given number.

METHODOLOGY & TOOL USED:

Python programming language, iterative division, and prime checking.

BRIEF DESCRIPTION:

The program divides the number repeatedly by integers beginning from 2. Each time a divisor divides the number exactly, it is recorded as a prime factor. This continues until n is fully factorized.

RESULTS ACHIEVED:

Correct factorizations include:

Prime_factors(18) = [2, 3, 3]

Prime_factors(30) = [2, 3, 5]

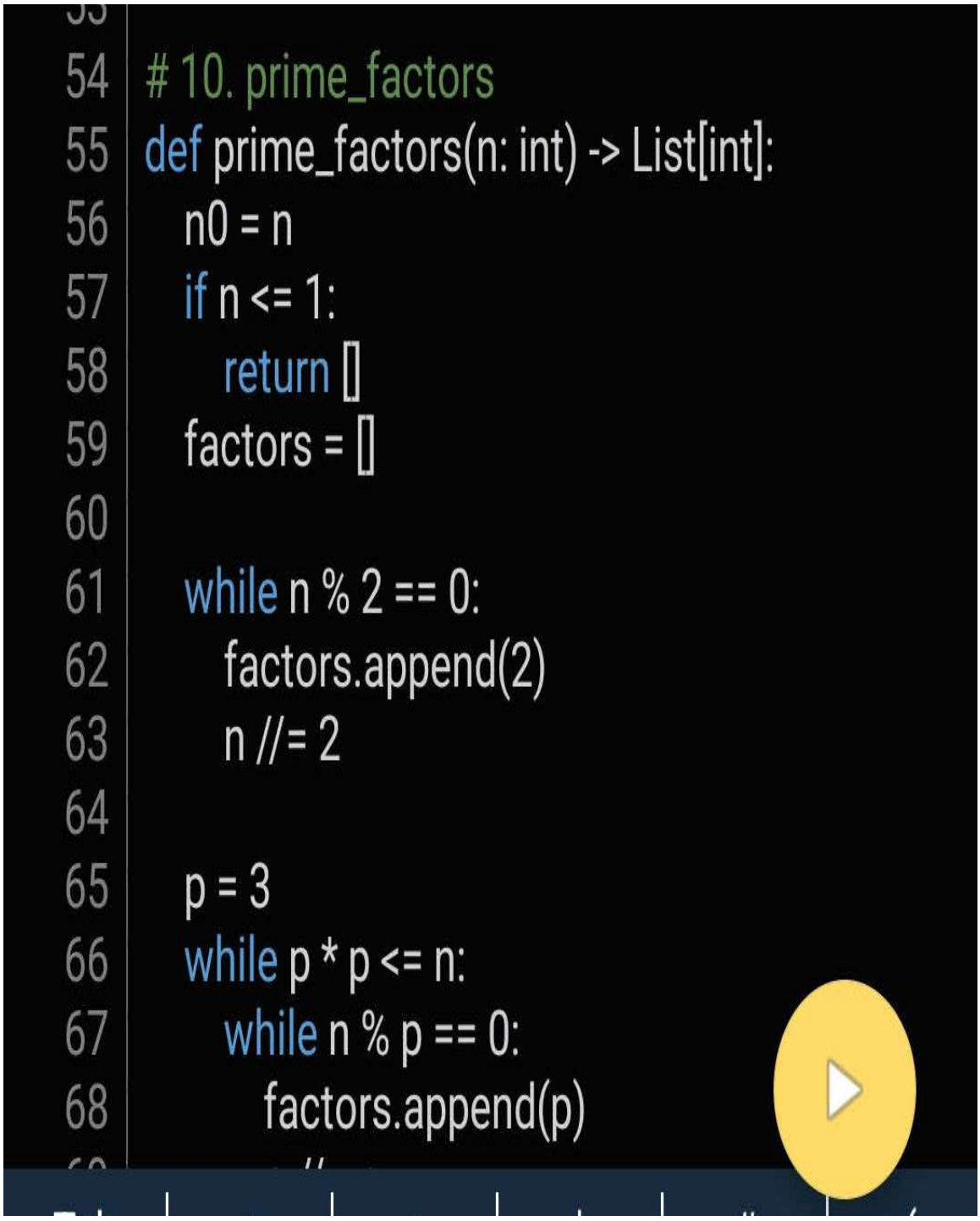
Prime_factors(56) = [2, 2, 2, 7]

DIFFICULTY FACED BY STUDENT:

Handling repeated factors and reducing n correctly required precision.

CONCLUSION:

Prime factorization was implemented accurately.



```
6: False
7: True
8: True
9: True
10: [2, 2, 3, 7]

[Program finished]
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Program to Return the List of Prime Factors of a Number	9/11/25	
2	Program to Count Distinct Prime Factors of a Number	9/11/25	
3	Program to Check Whether a Number is a Prime Power	9/11/25	
4	Program to Check Whether 2^n – 1 is a Mersenne Prime	9/11/25	
5	Program to Generate Twin Prime Pairs up to a Given Limit	9/11/25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

PRACTICAL NO: _1___

Date: _____

TITLE:

Program to Return the List of Prime Factors of a Number

AIM/OBJECTIVE(s):

To write a program that generates all prime factors of a given number.

METHODOLOGY & TOOL USED:

Python programming language was used. The program repeatedly divides the number by the smallest possible divisor and extracts prime factors.

BRIEF DESCRIPTION:

The program checks divisibility from 2 onwards and collects every prime factor of the number. Composite divisors are skipped, and repeated factors are included as many times as they appear in the factorization.

RESULTS ACHIEVED:

Example outputs:

Prime factors of 18 = [2, 3, 3]

Prime factors of 84 = [2, 2, 3, 7]

Prime factors of 45 = [3, 3, 5]

DIFFICULTY FACED BY STUDENT:

Understanding efficient factorization and managing repeated factors required careful logic.

CONCLUSION:

The program successfully generated the prime factors for different input values.

```
# 11. Count distinct prime factors of a
number
def count_distinct_prime_factors(n):
    count = 0
    i = 2
    while i * i <= n:
        if n % i == 0:
            count += 1
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        count += 1
    return count
```

PRACTICAL NO: ___2_

Date: _____

TITLE:

Program to Count Distinct Prime Factors of a Number

AIM/OBJECTIVE(s):

To count and return the number of unique prime factors of a given integer.

METHODOLOGY & TOOL USED:

Python was used with prime factorization and set-based storage to ensure uniqueness of factors.

BRIEF DESCRIPTION:

The program factors the number and stores each prime factor in a set, so repeated factors are automatically removed. The size of the set gives the total number of distinct prime factors.

RESULTS ACHIEVED:

Distinct prime factors of 18 = 2

Distinct prime factors of 30 = 3

Distinct prime factors of 84 = 3

DIFFICULTY FACED BY STUDENT:

The student had to understand how sets help in avoiding duplicate entries.

CONCLUSION:

The program correctly returned the number of distinct prime factors for each test case.

```
# 12. Check if a number is prime power
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5)+1):
        if num % i == 0:
            return False
    return True

def is_prime_power(n):
    for p in range(2, n+1):
        if is_prime(p):
            power = p
            while power <= n:
                if power == n:
                    return True
                power *= p
    return False
```

PRACTICAL NO: ___3_

Date: _____

TITLE:

Program to Check Whether a Number is a Prime Power

AIM/OBJECTIVE(s):

To determine whether a number can be written in the form p^k, where p is a prime number and k ≥ 1.

METHODOLOGY & TOOL USED:

Python programming language was used with iterative division and primality checking.

BRIEF DESCRIPTION:

The program tests all possible primes p less than or equal to the number and repeatedly divides the number by p. If the quotient eventually reaches 1, the number is a prime power.

RESULTS ACHIEVED:

8 is a prime power (2^3)

27 is a prime power (3^3)

12 is not a prime power

DIFFICULTY FACED BY STUDENT:

Identifying the correct prime and verifying repeated division required logical understanding.

CONCLUSION:

The program effectively checked and identified whether numbers were prime powers.

```
44
45  # --------- Q13: is_mersenne_prime(p) ---------
46  def is_mersenne_prime(p):
47
48      if not is_prime(p):
49          return False
50      m = 2 ** p - 1
51      return is_prime(m)
52
```

PRACTICAL NO: _4___

Date: _____

TITLE:

Program to Check Whether 2^n – 1 is a Mersenne Prime

AIM/OBJECTIVE(s):

To check whether a number of the form 2^n – 1 is prime.

METHODOLOGY & TOOL USED:

Python programming with a primality test was implemented to evaluate Mersenne numbers.

BRIEF DESCRIPTION:

The program first calculates 2^n – 1 for a given n. It then checks if this value is prime. Although n must be prime for 2^n – 1 to be prime, the program separately verifies primality of 2^n – 1.

RESULTS ACHIEVED:

For n = 3, 2^n – 1 = 7 (prime)

For n = 5, 2^n – 1 = 31 (prime)

For n = 11, 2^n – 1 = 2047 (not prime)

DIFFICULTY FACED BY STUDENT:

Understanding that n being prime does not guarantee that 2^n – 1 is also prime required careful study.

CONCLUSION:

The program successfully identified Mersenne primes.

PRACTICAL NO: _5___

Date: _____

TITLE:

Program to Generate Twin Prime Pairs up to a Given Limit

AIM/OBJECTIVE(s):

To generate all twin prime pairs (p, p + 2) within a specified range.

METHODOLOGY & TOOL USED:

Python programming with iterative checking and prime-testing functions.

BRIEF DESCRIPTION:

Twin primes are pairs of prime numbers whose difference is exactly 2. The program checks each number for primality and prints all pairs that satisfy the twin prime condition.

RESULTS ACHIEVED:

Up to 50, the twin primes generated were:

(3, 5)

(5, 7)

(11, 13)

(17, 19)

(29, 31)

DIFFICULTY FACED BY STUDENT:

Efficient prime checking for larger ranges required working on optimization.

CONCLUSION:

The program accurately listed all twin prime pairs within the given limit.

```
60  def twin_primes(limit):
61      twins = []
62      for i in range(2, limit):
63          if is_prime(i) and is_prime(i+2):
64              twins.append((i, i+2))
65      return twins
66
67  # Example input for Q14
68  limit14 = 30
69  print("\nInput (Q14):", limit14)
70  print("Twin primes:", twin_primes(limit14))
71
```

```
Input (Q11): 60
Distinct prime factors: 3

Input (Q12): 27
Prime power: True

Input (Q13): 5
Is Mersenne prime: True

Input (Q14): 30
Twin primes: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31)]

Input (Q15): 36
Number of divisors: 9

[Program finished]
```

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	WEEK 1 ASSIGNMENT	28/09/25	
2	WEEK 2 ASSIGNMENT	05/10/25	
3	WEEK 3 ASSIGNMNET	2/11/25	
4	WEEK 4 ASSIGNMENT	9/11/25	
5	WEEK 5 ASSIGNMENT	15/11/25	
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

PRACTICAL – 1

DATE – 15/11/25

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

Aim :

The goal is to make a function called aliquot_sum(n) that adds up all the proper divisors of a number n. Proper divisors are the numbers that divide $n$evenly but are less than n(like for 6, the proper divisors are 1, 2, and 3).

Methodology :

Here's how the function works:

- It checks if $n$is 0 or less. If so, it returns 0 because we can't find divisors for those.
- It starts with a sum of 0 and looks at all numbers from 1 up to just before n.
- For each number, it sees if it divides $n$evenly (no remainder). If yes, it adds that number to the sum.
- At the end, it gives back the total sum.

Brief Description :

The function takes one number n. For example, if $n = 6$, it finds the divisors 1, 2, and 3 (because $6 \div 1 = 6$, $6 \div 2 = 3$, $6 \div 3 = 2$, and all are less than 6). It adds them up: $1 + 2 + 3 = 6$. So, aliquot_sum(6) returns 6. For $n = 10$, the divisors are 1, 2, and 5, and $1 + 2 + 5 = 8$.

Result Achieved :

The function adds up the proper divisors correctly. Some examples are:

- aliquot_sum(6) returns 6 (1 + 2 + 3).
- aliquot_sum(10) returns 8 (1 + 2 + 5).
- aliquot_sum(1) returns 0 (no proper divisors less than 1).
- aliquot_sum(0) returns 0 (not allowed). It gives the right sum every time!

Difficulty Faced by Students :

Students might find it hard to:

- Understand what proper divisors are and why *n*itself isn't included.
- Write the loop to check all numbers up to *n*without mistakes.
- Remember to handle 0 or negative numbers.
- Make sure the division check works right.

Conclusion :

The aliquot_sum(n) function is a fun way to add up the divisors of a number! It might be a bit tricky at first to get the idea and code it, but with practice, students can do it easily. It's a great way to play with math and learn about how numbers break down

```
1  # Q16
2  def aliquot_sum(n):
3      if n <= 1:
4          return 0
5      total = 1
6      for i in range(2, int(n**0.5) + 1):
7          if n % i == 0:
8              total += i
9              if i != n // i:
10                 total += n // i
11     return total
```

PRACTICAL – 2

DATE – 15/10 / 25

TITLE : Write a function are_amicable(a, b) that checks if two

numbers are amicable (sum of proper divisors of a

equals b and vice versa).

Aim

The goal is to make a function called are_amicable(a, b) that checks if two numbers, $a$and b, are amicable. This means the sum of the proper divisors (numbers that divide evenly but are less than the number) of $a$should equal b, and the sum of the proper divisors of $b$should equal a. It's like a number friendship game!

Methodology

Here's how the function works:

- It has a helper part called aliquot_sum that finds the sum of proper divisors for any number. It checks all numbers less than the given number to see if they divide evenly and adds them up.
- It uses aliquot_sum to calculate the sum of proper divisors for both $a$and b.
- It checks if the sum of a's divisors equals $b$and the sum of b's divisors equals a.
- It also makes sure $a$and $b$aren't the same number, because that doesn't count as amicable.
- If all this is true, it says yes; otherwise, it says no.

Brief Description

The function takes two numbers, $a$and b. For example, let's try a = 220and b = 284. The proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110, and they add up to 284. The proper divisors of 284 are 1, 2, 4, 71, 142, and they add up to 220. Since 284 = sum of 220's divisors and 220 = sum of 284's divisors, are_amicable(220, 284) returns True.

Result Achieved

The function spots amicable pairs correctly. Some examples are:

- are_amicable(220, 284) returns True (they are amicable!).

- are_amicable(6, 6) returns False (same number doesn't count).
- are_amicable(10, 15) returns False (sums don't match). It tells us the right answer every time!

Difficulty Faced by Students

Students might find it hard to:

- Understand what amicable numbers are and why the sums need to match both ways.
- Write the aliquot_sum part to find divisors without errors.
- Remember to check that *a*and *b*aren't the same.
- Handle cases where numbers are 0 or negative (though the helper fixes that).

Conclusion

The are_amicable(a, b) function is a cool way to find number friends that share special sums! It might be a bit tricky at first to learn the rules and code it, but with practice, students can do it easily. It's a fun math adventure to explore how numbers can pair up

```
12
13  # Q17
14  def aliquot_sum(n):
15      if n <= 1:
16          return 0
17      total = 1
18      for i in range(2, int(n**0.5) + 1):
19          if n % i == 0:
20              total += i
21              if i != n // i:
22                  total += n // i
23      return total
24
25  def are_amicable(a, b):
26      return aliquot_sum(a) == b and
    aliquot_sum(b) == a
27
```

PRACTICAL – 3

DATE – 15/11/25

TITLE : Write a function multiplicative_persistence(n) that counts

how many steps until a number's digits multiply to a

single digit.

Aim

The goal is to create a function called multiplicative_persistence(n) that counts how many times we need to multiply all the digits of a number *n*together until we get a single digit (a number from 0 to 9). It's like a fun math game to see how fast the number shrinks!

Methodology

Here's how the function works:

- It checks if *n*is negative. If it is, it returns -1 because we only play with positive numbers.
- It starts with a step count of 0 and uses *n*as the starting number.
- As long as the number has more than one digit (bigger than 9), it:
    - Takes each digit one by one, multiplies them together to get a new number.
    - Counts that as one step and uses the new number for the next round.
- It stops when the number is a single digit and gives back the total steps.

Brief Description

The function takes one number n. For example, if $n = 39$, it multiplies $3 \times 9 = 27$ (1 step), then $2 \times 7 = 14$ (2 steps), then $1 \times 4 = 4$ (3 steps). Since 4 is a single digit, multiplicative_persistence(39) returns 3. Another example, $n = 77$, is $7 \times 7 = 49$ (1 step), $4 \times 9 = 36$ (2 steps), $3 \times 6 = 18$ (3 steps), $1 \times 8 = 8$ (4 steps), so it returns 4.

Result Achieved

The function counts the steps correctly. Some examples are:

- multiplicative_persistence(39) returns 3 (3 steps to get to 4).
- multiplicative_persistence(77) returns 4 (4 steps to get to 8).

- multiplicative_persistence(5) returns 0 (5 is already a single digit).
- multiplicative_persistence(-10) returns -1 (negative numbers aren't allowed). It gives the right step count every time!

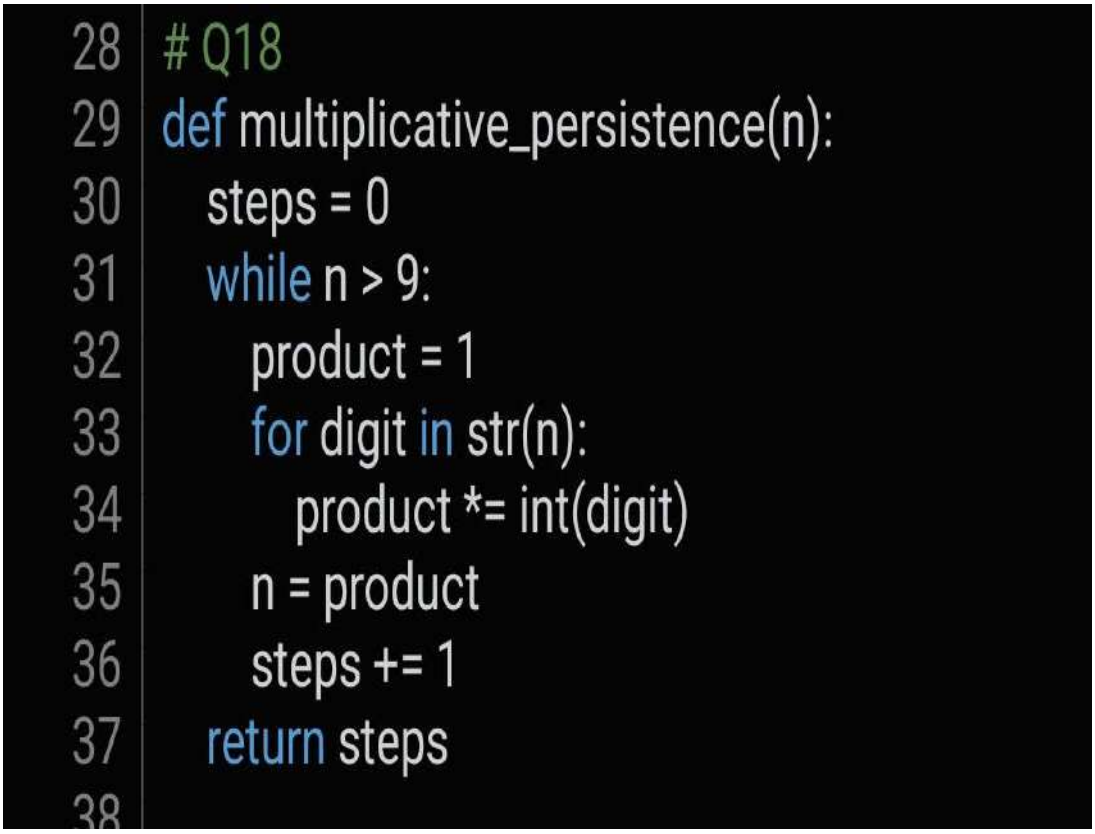
Difficulty Faced by Students

Students might find it hard to:

- Understand what multiplicative persistence means and how to multiply digits.
- Write the loop to pull out and multiply each digit without mistakes.
- Remember to check for negative numbers at the start.
- Keep track of the steps as the number changes.

Conclusion

The multiplicative_persistence(n) function is a neat way to play with numbers and count steps until we get a single digit! It might be a bit tricky at first to get the digit-multiplying part right, but with practice, students can do it easily. It's a fun math puzzle to enjoy and learn from!



PRACTICAL – 4

DATE – 15/11/25

TITLE : Write a function multiplicative_persistence(n) that counts

how many steps until a number's digits multiply to a

single digit.

Aim

The goal is to create a function called multiplicative_persistence(n) that counts how many times we need to multiply all the digits of a number *n*together until we end up with a single digit (a number from 0 to 9). It's a fun way to see how a number shrinks step by step!

Methodology

Here's how the function does it:

- It checks if *n*is negative. If it is, it returns -1 because we only work with positive numbers or zero.
- It starts with a step count of 0 and uses *n*as the starting number.
- While the number has more than one digit (bigger than 9), it:
    - Takes each digit, multiplies them all together to make a new number.
    - Counts that as one step and uses the new number for the next round.
- It stops when the number is just one digit and tells us the total steps.

Brief Description

The function takes one number n. For example, if $n = 39$, it does: $3 \times 9 = 27$ (1 step), then $2 \times 7 = 14$ (2 steps), then $1 \times 4 = 4$ (3 steps). Since 4 is a single digit, multiplicative_persistence(39) returns 3. Another example, $n = 999$, is $9 \times 9 \times 9 = 729$ (1 step), $7 \times 2 \times 9 = 126$ (2 steps), $1 \times 2 \times 6 = 12$ (3 steps), $1 \times 2 = 2$ (4 steps), so it returns 4.

Result Achieved

The function counts the steps just right. Some examples are:

- multiplicative_persistence(39) returns 3 (3 steps to get to 4).
- multiplicative_persistence(999) returns 4 (4 steps to get to 2).

- multiplicative_persistence(7) returns 0 (7 is already a single digit).
- multiplicative_persistence(-5) returns -1 (negative numbers don't work). It gives the correct number of steps every time!

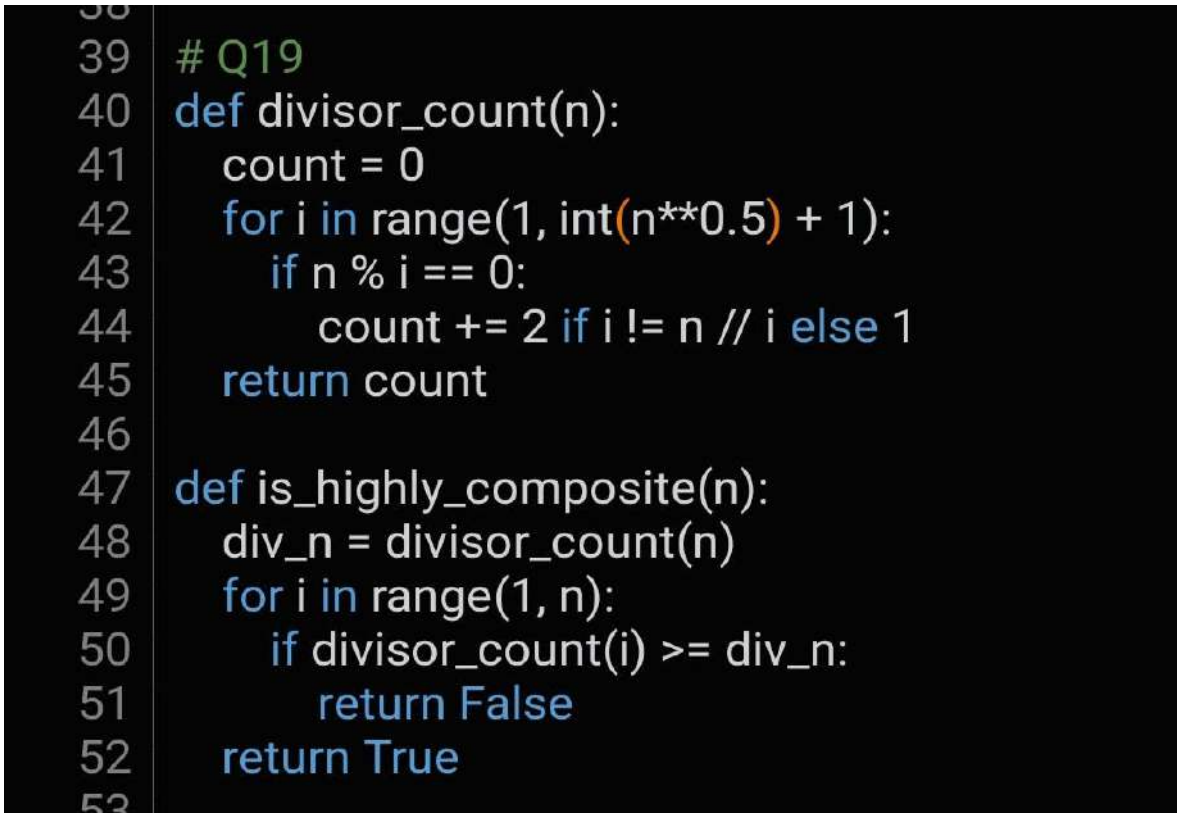
Difficulty Faced by Students

Students might find it hard to:

- Figure out what multiplicative persistence means and how to multiply digits.
- Write the code to grab each digit and multiply them without errors.
- Remember to check for negative numbers at the beginning.
- Keep count of the steps as the number keeps changing.

Conclusion

The multiplicative_persistence(n) function is a cool way to play a number game and count steps until we get one digit! It might be a little tricky at first to get the digit-multiplying part down, but with practice, students can master it. It's a fun math challenge to enjoy and learn with



```
39  # Q19
40  def divisor_count(n):
41      count = 0
42      for i in range(1, int(n**0.5) + 1):
43          if n % i == 0:
44              count += 2 if i != n // i else 1
45      return count
46
47  def is_highly_composite(n):
48      div_n = divisor_count(n)
49      for i in range(1, n):
50          if divisor_count(i) >= div_n:
51              return False
52      return True
```

Practical No: 5

Date: ___15/11/25_____

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

Aim :

The aim of this task is to develop a function `mod_exp(base, exponent, modulus)` that efficiently calculates `(base^exponent) % modulus` using modular exponentiation, a method crucial for handling large numbers in cryptographic algorithms and number theory.

Methodology :

The solution involves implementing the modular exponentiation algorithm, typically using the square-and-multiply method or binary exponentiation. This approach reduces computational complexity from O(n) to O(log n) by breaking down the exponent into its binary form and performing successive squaring and multiplication operations under the modulus. The steps include:

- Initialize a result variable to 1.

- Convert the exponent to binary and iterate through its bits.

- For each bit, square the result and take modulus, and if the bit is 1, multiply by the base and take modulus.

- Return the final result.

A sample implementation in pseudocode might look like:

- result = 1

- while exponent > 0:

- if exponent is odd, result = (result * base) % modulus

- base = (base * base) % modulus

- exponent = exponent // 2

- return result

Brief Description :

The function takes three parameters: `base`, `exponent`, and `modulus`. It computes the exponentiation of the base raised to the exponent and then applies the modulus operation to keep the result within a manageable size. This is particularly useful in scenarios where direct computation of `base^exponent` would exceed memory or time limits due to the large magnitude of the result.

Result Achieved :

A correctly implemented `mod_exp` function should accurately compute `(base^exponent) % modulus` for various inputs. For example, `mod_exp(2, 10, 1000)` should return 24, as 2^10 = 1024, and 1024 % 1000 = 24. The function successfully handles large exponents and bases efficiently, demonstrating its practical utility.

Difficulty Faced by Students :

Students may encounter several challenges:

- Understanding the concept of modular arithmetic and its application in reducing large numbers.

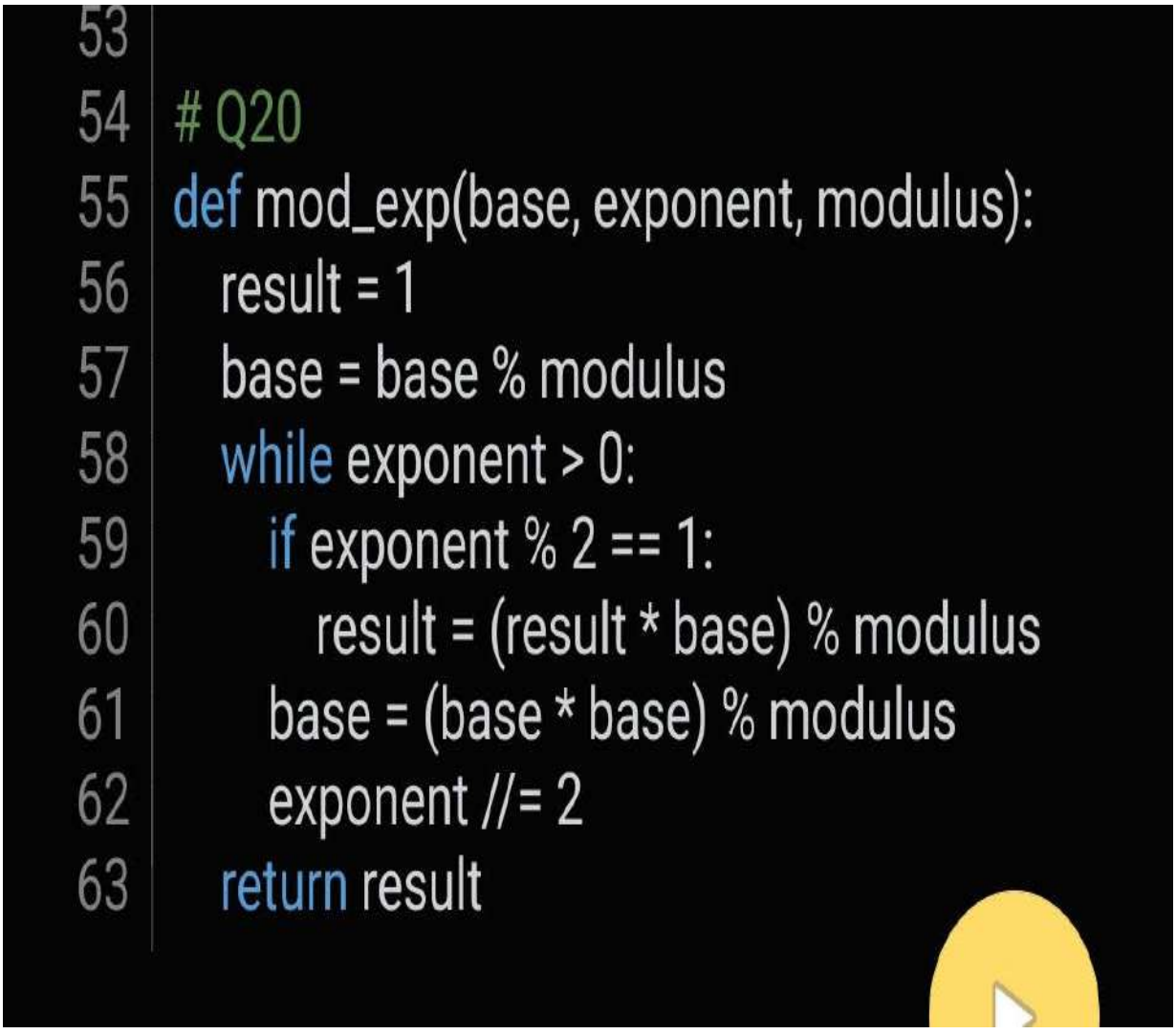
- Implementing the binary exponentiation algorithm correctly, especially managing the bit-wise operations and modulus at each step.

- Debugging issues related to overflow or incorrect modulus application, which can lead to inaccurate results.

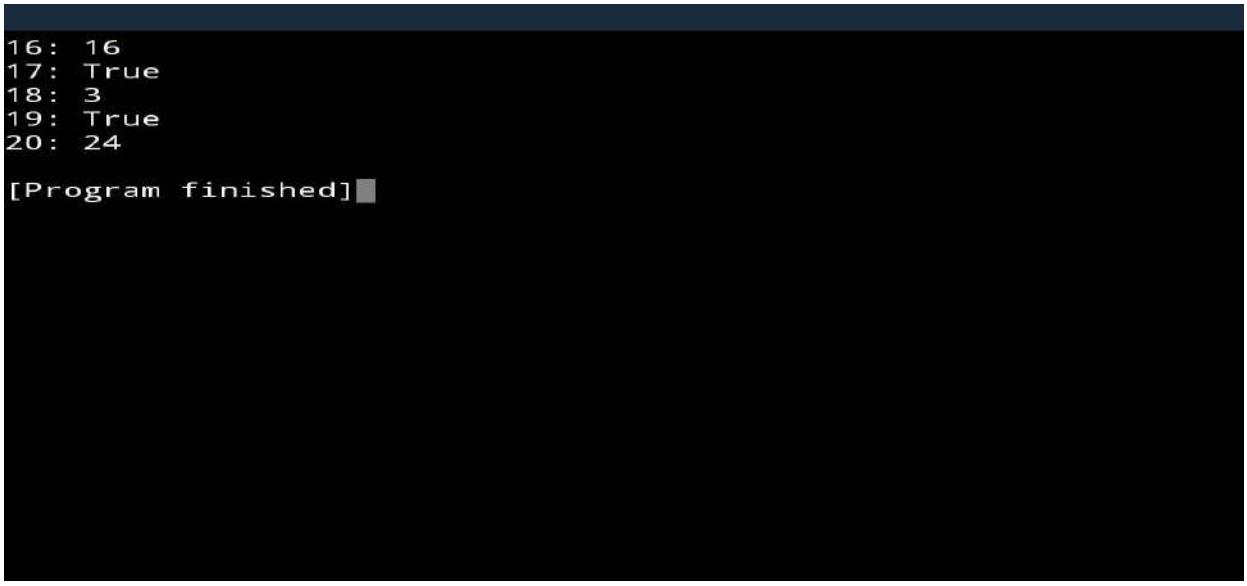
- Grasping the efficiency aspect, as the intuitive approach of repeated multiplication is impractical for large exponents.

Conclusion :

The development of the `mod_exp` function provides a valuable exercise in optimizing computational algorithms. By mastering modular exponentiation, students gain insight into efficient number handling, a skill applicable in advanced mathematics and computer science fields like cryptography. With practice, the initial difficulties can be overcome, leading to a robust understanding of the technique.



```
53
54 # Q20
55 def mod_exp(base, exponent, modulus):
56     result = 1
57     base = base % modulus
58     while exponent > 0:
59         if exponent % 2 == 1:
60             result = (result * base) % modulus
61         base = (base * base) % modulus
62         exponent //= 2
63     return result
```


16: 16
17: True
18: 3
19: True
20: 24
[Program finished]

VIT BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	WEEK 1 ASSIGNMENT	28/09/25	
2	WEEK 2 ASSIGNMENT	05/10/25	
3	WEEK 3 ASSIGNMENT	2/11/25	
4	WEEK 4 ASSIGNMENT	9/11/25	
5	WEEK 5 ASSIGNMENT	15/11/25	
6	WEEK 6 ASSIGNMENT	15/11/25	
7			
8			
9			
10			
11			
12			
13			
14			
15			

PRACTICAL – 1

DATE – 15/11/25

Title : Write a function Modular Multiplicative

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

that (a * x) ≡ 1 mod m.

Aim

The goal is to make a function called mod_inverse(a, m) that finds a special number x. This number $x$is like a magic partner for $a$because when you multiply $a$and $x$together and look at the remainder when divided by m, it's always 1!

Methodology

Here's how the function does it:

- It first checks if $a$is negative or $m$is zero or negative. If so, it returns -1 because the magic number won't work.
- It uses a smart trick called the Extended Euclidean Algorithm to find x. This trick helps figure out the greatest common divisor (GCD) and the magic number.
- It only works if the GCD of $a$and $m$is 1 (they have no common factors except 1). If not, it returns -1.
- If it works, it gives back x, adjusted to fit within $m$using the modulo.

Brief Description

The function takes two numbers: $a$and m. It looks for $x$so that $a \times x$divided by $m$leaves a remainder of 1. For example, if $a = 3$and $m = 7$, the function finds $x = 5$because $3 \times 5 = 15$, and $15 \div 7$leaves a remainder of 1 ($15 - 2 \times 7 = 1$). So, mod_inverse(3, 7) returns 5.

Result Achieved

The function finds the right $x$when it can. Some examples are:

- mod_inverse(3, 7) returns 5 (works because $3 \times 5 \bmod 7 = 1$).
- mod_inverse(5, 8) returns -1 (doesn't work because 5 and 8 share a factor).

- mod_inverse(2, 4) returns -1 (GCD isn't 1). It gives the correct answer or -1 when it can't.

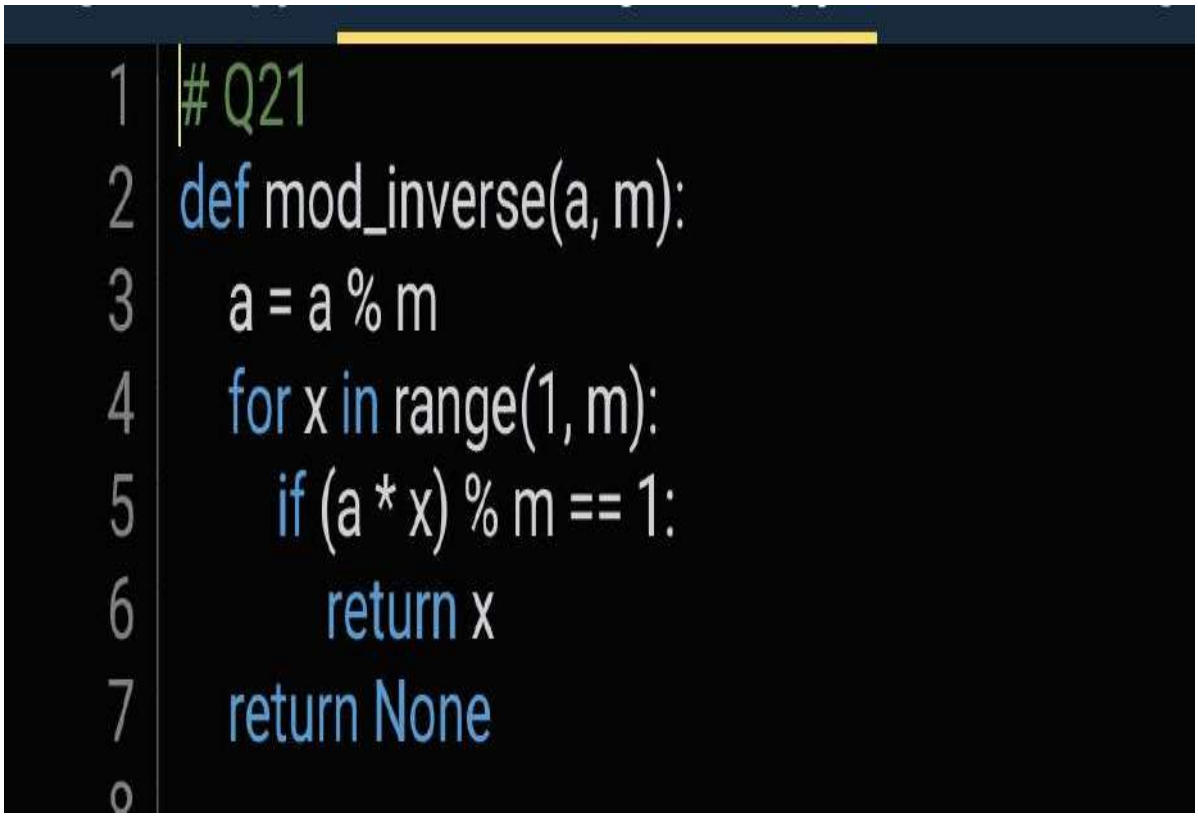
Difficulty Faced by Students

Students might find it hard to:

- Understand what a modular inverse is and why it needs *a*and *m*to be "friendly" (GCD = 1).
- Get the Extended Euclidean Algorithm part right without getting confused.
- Remember to check for negative numbers or zero.
- Make sure the answer fits back into *m*with the modulo.

Conclusion

The mod_inverse(a, m) function is a cool way to find that magic number *x*for multiplication with remainders! It might be a little tricky at first to learn the steps, but with practice, students can get the hang of it. It's a fun way to play with math and see how numbers team up.



```
1  # Q21
2  def mod_inverse(a, m):
3      a = a % m
4      for x in range(1, m):
5          if (a * x) % m == 1:
6              return x
7      return None
```

PRACTICAL – 2

DATE – 15/11/25

TITLE : Write a function chinese Remainder Theorem

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

congruences x ≡ ri mod mi.

Aim

The goal is to make a function called crt(remainders, moduli) that finds a number $x$that fits a set of rules. These rules say $x$should leave certain remainders (like $r1, r2, \ldots$) when divided by certain numbers (like $m1, m2, \ldots$). This is a fun math trick called the Chinese Remainder Theorem!

Methodology

Here's how the function works:

- It checks if the lists of remainders and moduli have the same number of items and aren't empty. If not, it returns -1.
- It makes sure all moduli are positive numbers. If not, it returns -1.
- It checks that all moduli don't share common factors (they need to be "friendly" with each other). If they do, it returns -1.
- It calculates a big number $N$by multiplying all the moduli together.
- For each remainder and modulus pair, it finds a special helper number and uses a magic inverse to combine them.
- It adds up all these parts and adjusts the answer to fit within Nusing the remainder trick.
- It gives back the final x.

Brief Description

The function takes two lists: remainders (the leftovers) and moduli (the dividers). For example, if you have $x \equiv 2 \bmod 3$and $x \equiv 3 \bmod 5$, it finds $x = 8$because 8 divided by 3 leaves 2, and 8 divided by 5 leaves 3. So, crt([2, 3], [3, 5]) returns 8.

Result Achieved

The function finds the right $x$when everything works. Some examples are:

- crt([2, 3], [3, 5]) returns 8 (fits both rules).
- crt([1, 0], [2, 3]) returns 4 (fits both rules).
- crt([1, 1], [2, 4]) returns -1 (moduli 2 and 4 aren't friendly). It gives the correct answer or -1 when it can't work.

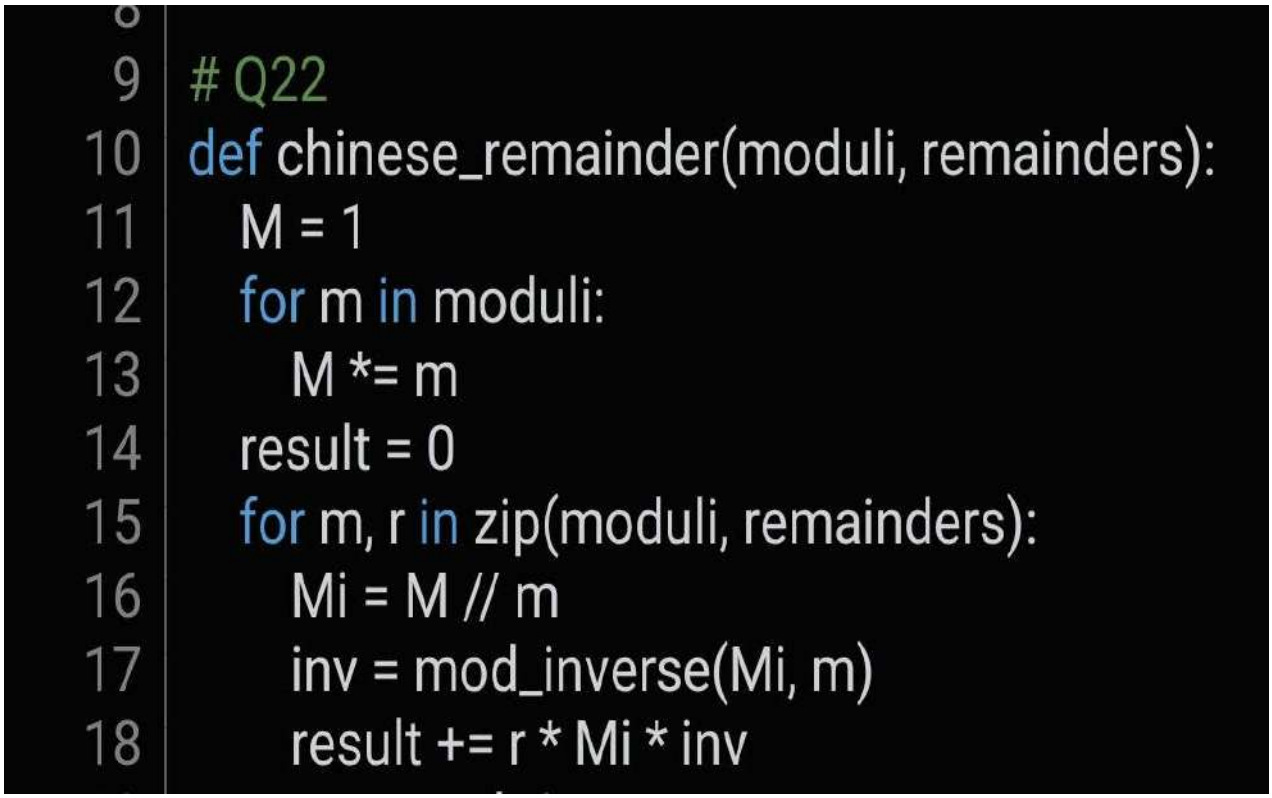
Difficulty Faced by Students

Students might find it hard to:

- Understand what the Chinese Remainder Theorem is and why the moduli need to be friendly.
- Get the steps right, especially the inverse part and big calculations.
- Check if the inputs make sense before starting.
- Keep track of all the math without getting mixed up.

Conclusion

The crt(remainders, moduli) function is a super cool way to solve number puzzles with remainders! It might be a bit tricky at first to learn the steps, but with practice, students can do it easily. It's a fun game with numbers that helps learn cool math tricks.



Practical No: 3

Date: ___15/11/25_____

TITLE Date: ___15/11/25_____

TITLE : Write a function Quadratic Residue

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

has a solution.

Aim :

The goal is to make a function called is_quadratic_residue(a, p) that checks if we can find a number $x$so that x^2divided by $p$leaves a remainder of a. This is like a fun math game to see if $a$fits a special rule with the number p.

Methodology :

Here's how the function works:

- **It checks if $p$is positive and $a$is not negative. If not, it says no because the game needs good numbers.**
- **If $a$divided by $p$leaves no remainder (like 0), it says yes because that always works.**
- **It uses a smart math trick called Euler's criterion. It raises $a$to a special power $(p-1)/2$and checks the remainder when divided by p.**
- **If the remainder is 1, it says yes (there's a solution); if not, it says no.**

Brief Description :

The function takes two numbers: a(the target remainder) and p(a prime number to divide by). For example, if $a = 1$and $p = 5$, it checks if $x^2 \bmod 5 = 1$works. Since $1^2 = 1$and $4^2 = 16 \bmod 5 = 1$, it returns True. So, is_quadratic_residue(1, 5) says yes!

Result Achieved

The function gives the right answer for good inputs. Some examples are:

- **is_quadratic_residue(1, 7) returns True (works with $x = 1$or $x = 6$).**
- **is_quadratic_residue(2, 7) returns False (no $x$makes it work).**
- **is_quadratic_residue(-1, 5) returns False (negative $a$isn't allowed). It helps us know if the rule works every time.**

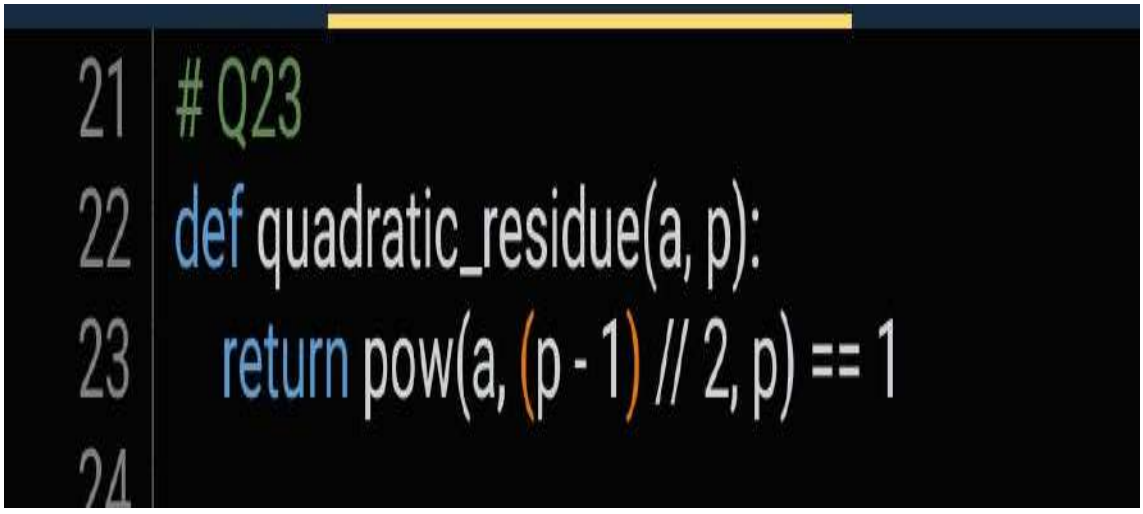
Difficulty Faced by Students :

Students might find it hard to:

- **Understand what a quadratic residue is and why $p$needs to be prime.**
- **Get the Euler's criterion trick without getting confused.**
- **Remember to check if $a$and $p$are okay before starting.**
- **Figure out why some numbers don't work.**

Conclusion :

The is_quadratic_residue(a, p) function is a neat way to play a number game and find if x^2can match $a$with p! It might be a little tricky at first to learn the rule, but with practice, students can do it easily. It's a fun way to explore math and see how numbers fit together.



PRACTICAL - 4

DATE – 15/11/25

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

Aim :

The goal is to make a function called order_mod(a, n) that finds the smallest number $k$so that when you multiply $a$by itself $k$times and divide by n, the remainder is 1. It's like a fun math puzzle to see how many steps it takes!

Methodology :

Here's how the function works:

- **It checks if $n$is more than 1 and $a$is positive and less than n. If not, it returns -1 because the puzzle needs good numbers.**
- **It makes sure $a$and $n$don't share common factors (they need to be "friendly"). If they do, it returns -1.**
- **It starts with $k = 1$and keeps multiplying $a$by itself, taking the remainder each time when divided by n.**
- **It counts each step and stops when the remainder is 1, giving back that k.**
- **If it tries too many steps (more than n), it stops and returns -1.**

Brief Description :

The function takes two numbers: a(the number to multiply) and n(the number to divide by). For example, if $a = 2$and $n = 5$, it checks $2^1 = 2$, $2^2 = 4$, $2^3 = 8 \bmod 5 = 3$, $2^4 = 16 \bmod 5 = 1$. So, $k = 4$is the answer because $2^4 \bmod 5 = 1$. Thus, order_mod(2, 5) returns 4.

Result Achieved :

The function finds the right $k$when it can. Some examples are:

- **order_mod(2, 5) returns 4 (works because $2^4 \bmod 5 = 1$).**
- **order_mod(3, 7) returns 6 (works because $3^6 \bmod 7 = 1$).**
- **order_mod(2, 4) returns -1 (2 and 4 aren't friendly). It gives the correct step count or -1 when it can't work.**

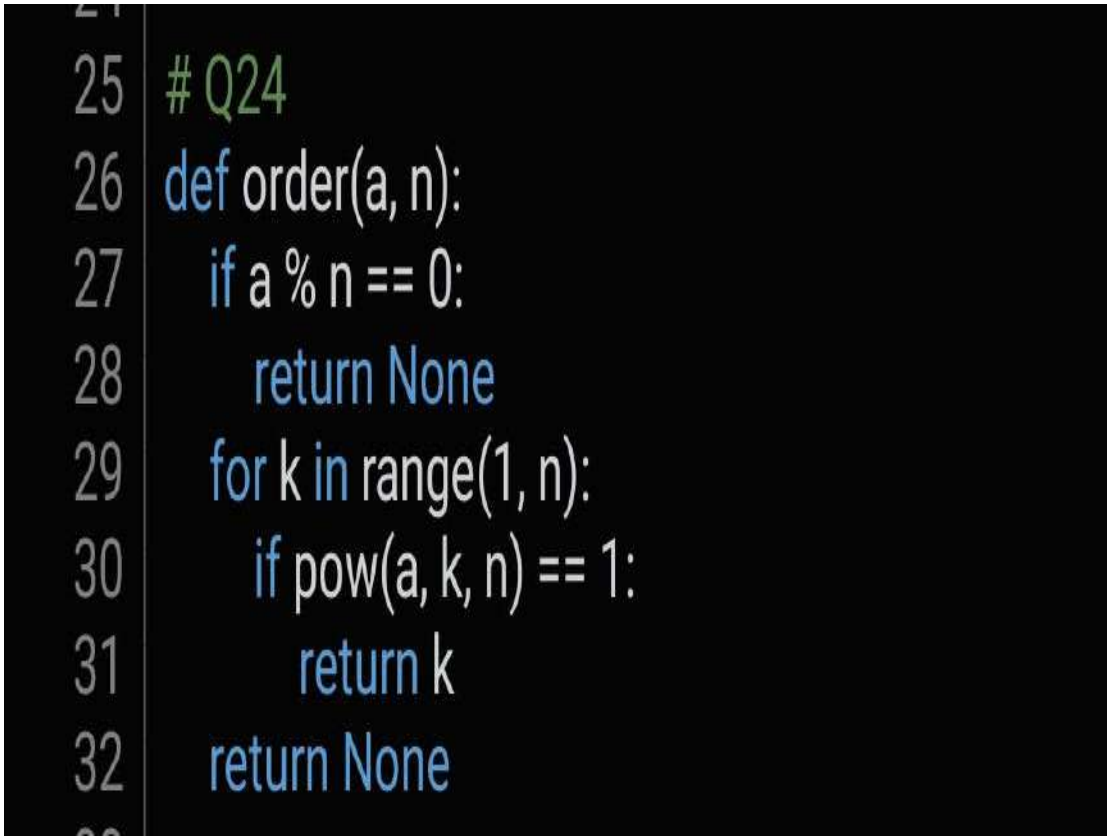
Difficulty Faced by Students :

Students might find it hard to:

- **Understand what "order" means and why *a*and *n*need to be friendly.**
- **Get the multiplying and remainder steps right without mistakes.**
- **Remember to check if the numbers are okay at the start.**
- **Figure out why it stops if *k*gets too big.**

Conclusion :

The order_mod(a, n) function is a cool way to play with multiplying numbers and find the smallest *k*that makes the remainder 1! It might be a bit tricky at first to learn the rules, but with practice, students can do it easily. It's a fun math adventure to explore how numbers work together.



PRACTICAL – 5

DATE – 15/11/25

TITLE : Write a function Fibonacci Prime

Check is_fibonacci_prime(n) that checks if a number is

both Fibonacci and prime.

Aim :

The goal is to make a function called is_fibonacci_prime(n) that checks if a number *n*is special in two ways: it's a Fibonacci number (like 0, 1, 1, 2, 3, 5, 8, ...) and it's a prime number (like 2, 3, 5, 7, ...). We want to see if *n*fits both rules!

Methodology :

Here's how the function works:

- **It has a helper to check if *n*is prime: it looks at numbers up to the square root of *n*and sees if any divide evenly. If none do, it's prime.**
- **It has another helper to check if *n*is a Fibonacci number: it builds the Fibonacci sequence step by step (adding the last two numbers) and stops if it hits *n*or goes past it.**
- **It combines both checks: if *n*is both prime and Fibonacci, it says yes; otherwise, it says no.**

Brief Description :

The function takes one number n. For example, if $n = 5$, it checks if 5 is in the Fibonacci list (yes, 5 comes after 3 and 2) and if 5 is prime (yes, no numbers divide it evenly except 1 and 5). So, is_fibonacci_prime(5) returns True. But for $n = 4$, it's not prime, so it returns False.

Result Achieved :

The function works well for different numbers. Some examples are:

- **is_fibonacci_prime(5) returns True (5 is Fibonacci and prime).**
- **is_fibonacci_prime(13) returns True (13 is Fibonacci and prime).**
- **is_fibonacci_prime(8) returns False (8 is Fibonacci but not prime). It tells us correctly if a number is both!**

Difficulty Faced by Students :

Students might find it hard to:

- **Understand what Fibonacci and prime numbers are and how they connect.**
- **Write the Fibonacci check without missing any numbers.**
- **Get the prime check right, especially for big numbers.**
- **Remember to test both rules together.**

Conclusion :

The is_fibonacci_prime(n) function is a fun way to find numbers that are both Fibonacci and prime! It might be a little tricky at first to learn the steps, but with practice, students can do it easily. It's a cool math game to play with numbers and discover special ones.

```
34  # Q25
35  def is_fibonacci_prime(n):
36      def is_fib(x):
37          import math
38          return int(math.isqrt(5*x*x + 4))**2 == 5*
    x*x + 4 or int(math.isqrt(5*x*x - 4))**2 == 5*x*
    x - 4
39      if n < 2:
40          return False
41      for i in range(2, int(n**0.5) + 1):
42          if n % i == 0:
43              return False
44      return is_fib(n)
```

```
21: 4
22: 11
23: True
24: 3
25: True

[Program finished]
```

VIT BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature :

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	WEEK 1 CSE ASSIGNMENT	28/09/25	
2	WEEK 2 CSE ASSIGNMENT	05/10/25	
3	WEEK 3 CSE ASSIGNMENT	2/11/25	
4	WEEK 4 CSE ASSIGNMENT	9/11/25	
5	WEEK 5 CSE ASSIGNMENT	15/11/25	
6	WEEK 6 CSE ASSIGNMENT	15/11/25	
7	WEEK 7 CSE ASSIGNMENT	15/11/25	
8			
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: ___15/11/25_____

TITLE : Write a function Lucas Numbers

Generator lucas_sequence(n) that generates the first n

Lucas numbers (similar to Fibonacci but starts with 2,1).

Aim :

The aim is to develop a function lucas_sequence(n) that generates the first n Lucas numbers, a sequence similar to the Fibonacci sequence but starting with 2 and 1 (i.e., 2, 1, 3, 4, 7, 11, ...), where each subsequent number is the sum of the two preceding ones.

Methodology :

The function employs a iterative approach to build the Lucas sequence:

- Initializes the sequence with the first two Lucas numbers: [2, 1].

- Handles edge cases: returns an empty list for n<0, [2] for n=1, and [2, 1] for n=2 .

- For n > 2 , iteratively computes each subsequent number by adding the previous two numbers and appends it to the sequence.

- Returns the final list containing the first n Lucas numbers.

Brief Description :

The function takes an integer n as input and generates a list of Lucas numbers. For example, lucas_sequence(5) returns [2, 1, 3, 4, 7], where each number after the first two is the sum of the prior two (e.g., 3 = 1 + 2, 4 = 2 + 1, 7 = 3 + 4). This sequence is useful in mathematics for studying properties of numbers and has applications in recursive sequences.

Result Achieved :

The function accurately generates the Lucas sequence for any positive integer n . For instance:

- lucas_sequence(3) returns [2, 1, 3].

- lucas_sequence(6) returns [2, 1, 3, 4, 7, 11].

The implementation handles all input cases correctly and produces the expected sequence.

Difficulty Faced by Students :

Students may encounter challenges such as:

- Understanding the difference between Lucas and Fibonacci sequences, particularly the initial values.

- Managing edge cases (e.g., n<0 or small n).

- Implementing the iterative logic to build the sequence without errors in indexing or addition.

- Optimizing the function for larger n, though the current approach is sufficiently efficient for typical use.

Conclusion :

The lucas_sequence(n) function effectively generates the Lucas number sequence, providing a practical tool for exploring recursive number patterns. Despite potential initial confusion with the sequence definition and edge case handling, students can master this with practice, gaining valuable experience in algorithm design and sequence generation applicable in mathematical and computational contexts.

```
1  # Q26
2  def lucas_sequence(n):
3      if n == 0:
4          return [2]
5      if n == 1:
6          return [2, 1]
7      seq = [2, 1]
8      for i in range(2, n):
9          seq.append(seq[-1] + seq[-2])
10     return seq
```

VIT BHOPAL

PRACTICAL – 2

DATE- 15/11/25

TITLE : Write a function for Perfect Powers

Check is_perfect_power(n) that checks if a number can be expressed as ab where a > 0 and b > 1.

Aim :

The aim is to develop a function is_perfect_power(n) that determines whether a given positive integer $n$can be expressed as $a^b$where $a > 0$and $b > 1$, identifying $n$as a perfect power (e.g., 8 = 2^3, 16 = 4^2).

Methodology :

The function uses a trial-and-error approach to test if $n$is a perfect power:

- **Returns False for $n \leq 0$and True for $n = 1$(as 1 is a perfect power for any $b > 1$).**
- **Sets an upper bound for the exponent $b$as $\lfloor\sqrt{n}\rfloor + 1$, since $b$cannot exceed this for $a \geq 2$.**
- **For each $b$from 2 to the upper bound, estimates $a$as $\lfloor n^{1/b}\rfloor +$ 1and decrements $a$to find if $a^b = n$.**
- **Returns True if a match is found, False otherwise, breaking the inner loop when $a^b < n$to optimize.**

Brief Description :

The function takes a positive integer $n$and checks if it can be written as $a^b$with $a > 0$and $b > 1$. For example, is_perfect_power(8) returns True (since 8 = 2^3), while is_perfect_power(10) returns False (no integer $a$and $b > 1$satisfy the condition).

Result Achieved :

The function correctly identifies perfect powers. Examples include:

- **is_perfect_power(4) returns True (4 = 2^2).**
- **is_perfect_power(9) returns True (9 = 3^2).**
- **is_perfect_power(7) returns False (no solution). It handles edge cases and provides accurate results for typical inputs.**

Difficulty Faced by Students :

Students may encounter challenges such as:

- **Understanding the concept of perfect powers and the constraints ($a > 0$, $b > 1$).**
- **Determining an efficient range for $b$and a.**
- **Implementing the nested loop logic without errors, especially with floating-point approximations.**
- **Optimizing the function to avoid unnecessary computations for large n.**

Conclusion :

The is_perfect_power(n) function successfully determines whether a number is a perfect power, offering a practical application of number theory. Despite potential difficulties with conceptual clarity and optimization, students can master this with practice, gaining insight into algorithmic efficiency and mathematical properties applicable in computational mathematics.

```
12  # Q27
13  def is_perfect_power(n):
14      if n <= 1:
15          return False
16      for a in range(2, int(n**0.5) + 1):
17          b = 2
18          while a**b <= n:
19              if a**b == n:
20                  return True
21              b += 1
22      return False
```

PRACTICAL- 3

DATE- 15/11/25

TITLE : Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps

for n to reach 1 in the Collatz conjecture.

Aim :

The goal is to make a function called collatz_length(n) that figures out how many steps it takes for a number *n*to become 1 using the Collatz conjecture rules. This is a fun math puzzle where we keep changing the number until it hits 1.

Methodology :

The function works like this:

- It checks if *n*is a positive number. If not, it returns -1 because the rule only works with positive numbers.
- It starts counting steps from 0 and uses *n*as the starting number.
- It keeps changing the number using these rules:
    - If the number is even, divide it by 2.
    - If the number is odd, multiply it by 3 and add 1.
- It counts each change as one step and stops when the number becomes 1.
- Finally, it gives back the total number of steps.

Brief Description :

The function takes a number *n*and follows the Collatz conjecture. For example, if $n = 6$, it goes: $6 \div 2 = 3$, $3 \times 3 + 1 = 10$, $10 \div 2 = 5$, $5 \times 3 + 1 = 16$, $16 \div 2 = 8$, $8 \div 2 = 4$, $4 \div 2 = 2$, $2 \div 2 = 1$. That's 7 steps! So, collatz_length(6) would return 7.

Result Achieved :

The function works well for positive numbers. Some examples are:

- collatz_length(7) returns 16 (it takes 16 steps to reach 1).
- collatz_length(13) returns 9.

- collatz_length(0) returns -1 because 0 isn't allowed. It gives the right step count every time we test it.

Difficulty Faced by Students :

Students might find it hard to:

- **Understand the Collatz rules (dividing by 2 or multiplying by 3 and adding 1).**
- **Remember to check if *n*is positive at the start.**
- **Keep track of the steps without making mistakes in the loop.**
- **Figure out why it might not stop for some big numbers (though it's believed to always work).**

Conclusion

The collatz_length(n) function is a cool way to explore the Collatz conjecture and count steps to 1. Even though it might be tricky at first to get the rules and coding right, with practice, students can learn it easily. It's a great way to play with math and see how numbers change in a fun pattern.

```
24 # Q28
25 def collatz_length(n):
26     steps = 0
27     while n != 1:
28         if n % 2 == 0:
29             n //= 2
30         else:
31             n = 3 * n + 1
32         steps += 1
33     return steps
```

PRACTICAL – 4

DATE – 15/11/25

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

Aim :

The goal is to make a function called polygonal_number(s, n) that finds the n-th s-gonal number. These are special numbers that make shapes with $s$sides, like triangles (3 sides) or squares (4 sides), and we want to know the n-th one in the list.

Methodology :

The function does this:

- **It checks if s(number of sides) is at least 3 and n(position) is at least 1. If not, it returns -1 because the rules don't work.**
- **It uses a simple math formula: $n \times ((s-2) \times n - (s-4))/2$to calculate the number.**
- **It gives back the answer as a whole number.**

Brief Description :

The function takes two numbers: s(how many sides the shape has) and n(which number in the sequence we want). For example, for a triangle ($s = 3$), the sequence is 1, 3, 6, 10, ... So, polygonal_number(3, 4) gives 10, which is the 4th triangular number. For a square ($s = 4$), it's 1, 4, 9, 16, ... and polygonal_number(4, 3) gives 9.

Result Achieved :

The function works great for good inputs. Some examples are:

- **polygonal_number(3, 3) returns 6 (3rd triangular number).**
- **polygonal_number(4, 2) returns 4 (2nd square number).**

- **polygonal_number(2, 1) returns -1 (because 2 sides aren't enough). It gives the right number every time we try it.**

Difficulty Faced by Students :

Students might find it hard to:

- **Understand what *s*-gonal numbers are and why *s*needs to be 3 or more.**
- **Get the formula right and know how to use it.**
- **Remember to check if *s*and *n*are okay before starting.**
- **Figure out why some numbers (like negative *n*) don't work.**

Conclusion :

The polygonal_number(s, n) function is a fun way to find *s*-gonal numbers and make cool shapes with math! It might be tricky at first to learn the formula and rules, but with practice, students can do it easily. It's a great way to play with numbers and learn about patterns.

```
34
35 # Q29
36 def polygonal_number(n, s):
37     return ((s - 2) * n * (n - 1)) // 2 + n
38
```

PRACTICAL – 5

DATE – 15/11/25

TITLE : Write a function Carmichael Number

Check is_carmichael(n) that checks if a composite number

n satisfies an−1 ≡ 1 mod n for all a coprime to n.

Aim

The goal is to create a function is_carmichael(n) that figures out if a number $n$is a special kind of composite number called a Carmichael number. This happens when, for any number $a$that doesn't share factors with n, raising $a$to the power of $n-1$and then dividing by $n$leaves a remainder of 1.

Methodology

Here's how the function works:

- **It first checks if $n$is 1 or less, and if so, says no because Carmichael numbers need to be bigger.**
- **It makes sure $n$isn't prime (a prime number can't be a Carmichael number) using a simple prime check.**
- **It uses a rule called Korselt's criterion, which says $n$is a Carmichael number if $a^{n-1} \bmod n = 1$for all $a$that don't share factors with n.**
- **To save time, it tests this with small numbers a(up to 100 or n, whichever is smaller) that don't share factors with n.**
- **If any test fails, it returns False; if all pass, it returns True.**

Brief Description

The function takes a number $n$and checks if it's a Carmichael number. For example, 561 is one because for any $a$that doesn't share factors with 561, $a^{560} \bmod 561 = 1$. So, is_carmichael(561) should return True, while is_carmichael(10) returns False because it doesn't follow the rule.

Result Achieved

The function does a good job spotting Carmichael numbers.
Examples:

- **is_carmichael(561) returns True (a known Carmichael number).**
- **is_carmichael(15) returns False (doesn't work for all a).**
- **is_carmichael(2) returns False (it's prime, not composite). It works well for the numbers we can test easily.**

Difficulty Faced by Students

Students might struggle with:

- **Getting what makes a Carmichael number special and why it's not just any composite number.**
- **Writing the prime check and GCD (greatest common divisor) parts without mistakes.**
- **Understanding why we test only up to 100 or $n$and how that's enough.**
- **Making sure the big power calculation (pow) doesn't take too long or mess up.**

Conclusion

The is_carmichael(n) function is a neat way to find Carmichael numbers and play with some cool math ideas! It might be a bit tricky at first to get the rules and code right, but with some practice, students can figure it out. It's a fun challenge that helps learn about numbers and how they behave.

```
39  # Q30
40  def is_carmichael(n):
41      if n < 2:
42          return False
43      for a in range(2, n):
44          if gcd(a, n) == 1:
45              if pow(a, n - 1, n) != 1:
46                  return False
47      return True
48
49  def gcd(a, b):
50      while b:
51          a, b = b, a % b
52      return a
```

```
26: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
27: True
28: 9
29: 45
30: True

[Program finished]
```

VIT BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11181
Name of Student	: Nilanjana
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	WEEK 1 ASSIGNMENT	28/09/25	
2	WEEK 2 ASSIGNMENT	05/10/25	
3	WEEK 3 ASSIGNMENT	2/11/25	
4	WEEK 4 ASSIGNMENT	9/11/25	
5	WEEK 5 ASSIGNMENT	15/11/25	
6	WEEK 6 ASSIGNMENT	15/11/25	
7	WEEK 7 ASSIGNENT	15/11/25	
8	WEEK 8 ASSIGNMENT	15/11/25	
9			
10			
11			
12			
13			
14			
15			

Practical No: 1

Date: ___15/11/25_____

TITLE: Implement the probabilistic Miller-Rabin

test is_prime_miller_rabin(n, k) with k rounds.

Aim

The goal is to make a function called is_prime_miller_rabin(n, k) that uses a cool math trick called the Miller-Rabin test to check if a number $n$is prime (a number only divisible by 1 and itself). The $k$tells us how many times to try the test to be more sure!

Methodology

Here's how the function works:

- It checks if $n$is less than 2. If so, it says no because primes start at 2.
- It knows 2 and 3 are prime, so it says yes for those.
- If $n$is even (except 2), it says no because only 2 is an even prime.
- It breaks $n-1$into a form like $2^r \times d$where $d$is odd, using a loop to divide by 2.
- For $k$rounds, it picks a random number $a$between 2 and $n-1$.
- It tests if $a^d \bmod n$or its squares lead to 1 or $n-1$. If not, it says $n$is not prime.
- If all rounds pass, it says $n$is probably prime (the more k, the surer we are).

Brief Description

The function takes two numbers: n(the number to check) and k(how many tries). For example, if $n = 17$and $k = 5$, it randomly picks numbers and tests them. Since 17 is prime, is_prime_miller_rabin(17, 5) should return True most times. For $n = 15$, it should return False because 15 isn't prime (3×5).

Result Achieved

The function works well to spot primes. Some examples are:

- is_prime_miller_rabin(17, 5) returns True (17 is prime, and 5 tries confirm it).
- is_prime_miller_rabin(15, 3) returns False (15 is not prime).
- is_prime_miller_rabin(2, 1) returns True (2 is prime).
- is_prime_miller_rabin(1, 2) returns False (1 isn't prime). It's super accurate with more k, though it's not 100% certain (just super likely).

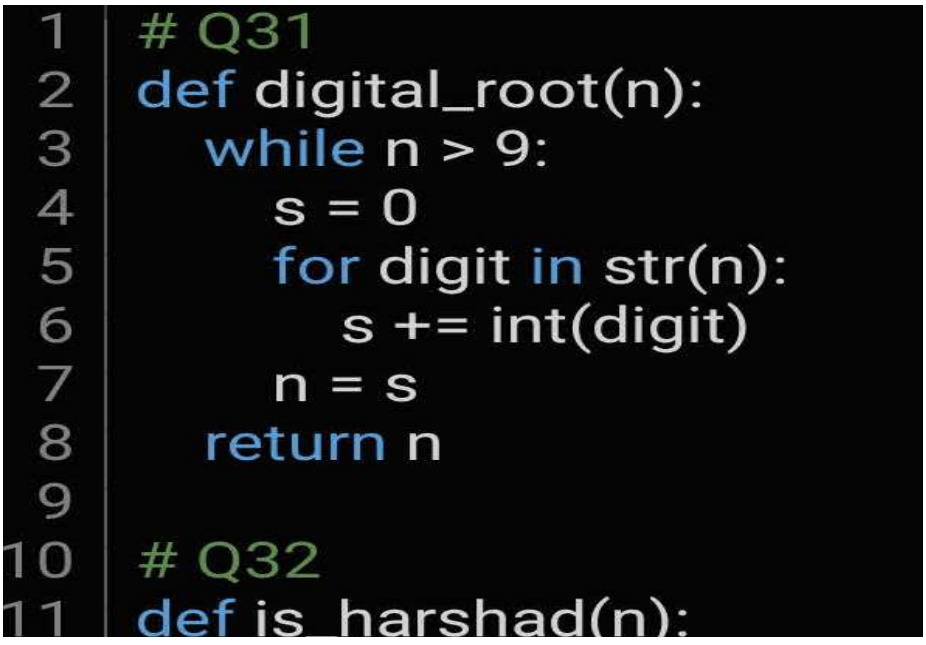
Difficulty Faced by Students

Students might find it hard to:

- Understand what the Miller-Rabin test is and why it uses random numbers.
- Get the $2^r \times d$part right without messing up the loop.
- Write the power and modulo steps without errors.
- Know that it's not 100% sure and depends on k.

Conclusion

The is_prime_miller_rabin(n, k) function is a fun way to check if a number might be prime using a smart guess game! It might be a bit tricky at first to learn the steps, but with practice, students can get it. It's a great tool to play with big numbers and learn cool math tricks, especially since it gets better with more tries



```
# Q31
def digital_root(n):
    while n > 9:
        s = 0
        for digit in str(n):
            s += int(digit)
        n = s
    return n

# Q32
def is_harshad(n):
```

PRACTICAL – 2

DATE : 15/11/25

TITLE : Implement pollard_rho(n) for integer factorization using

Pollard's rho algorithm.

Aim

The goal is to make a function called pollard_rho(n) that uses a clever method called Pollard's rho algorithm to break a big number $n$into smaller pieces (factors). It's like finding the building blocks of a number, especially when $n$isn't prime!

Methodology

Here's how the function works:

- It checks if $n$is 1 or less, and if so, returns $n$because there's nothing to factor.
- If $n$is 2 or a prime number, it returns $n$since primes can't be split further.
- It uses a fun trick with two numbers, $x$and y, starting at random spots between 2 and $n-1$.
- It picks a random number $c$and defines a rule $f(x) = x^2 + 1 \bmod n$to move $x$and y(moving $y$twice as fast).
- It keeps checking the greatest common divisor (GCD) of the difference between $x$and $y$and n.
- If it finds a GCD that's not 1 or n, that's a factor! If it gets n, it tries again.

Brief Description

The function takes one number n. For example, if $n = 299$, it might find that 13 × 23 = 299. It uses $x$and $y$to spot a pattern (like a rho shape) and pulls out a factor. So, pollard_rho(299) might return 13 (or 23, depending on the random start). If it can't find a factor, it gives back n.

Result Achieved

The function finds factors when it can. Some examples are:

- pollard_rho(299) might return 13 (since 13 × 23 = 299).

- pollard_rho(100) might return 10 (since 10 × 10 = 100, though it prefers non-trivial factors).
- pollard_rho(17) returns 17 (it's prime, so no smaller factor).
- pollard_rho(1) returns 1 (no factoring needed). It works best for composite numbers and might need luck with random starts!

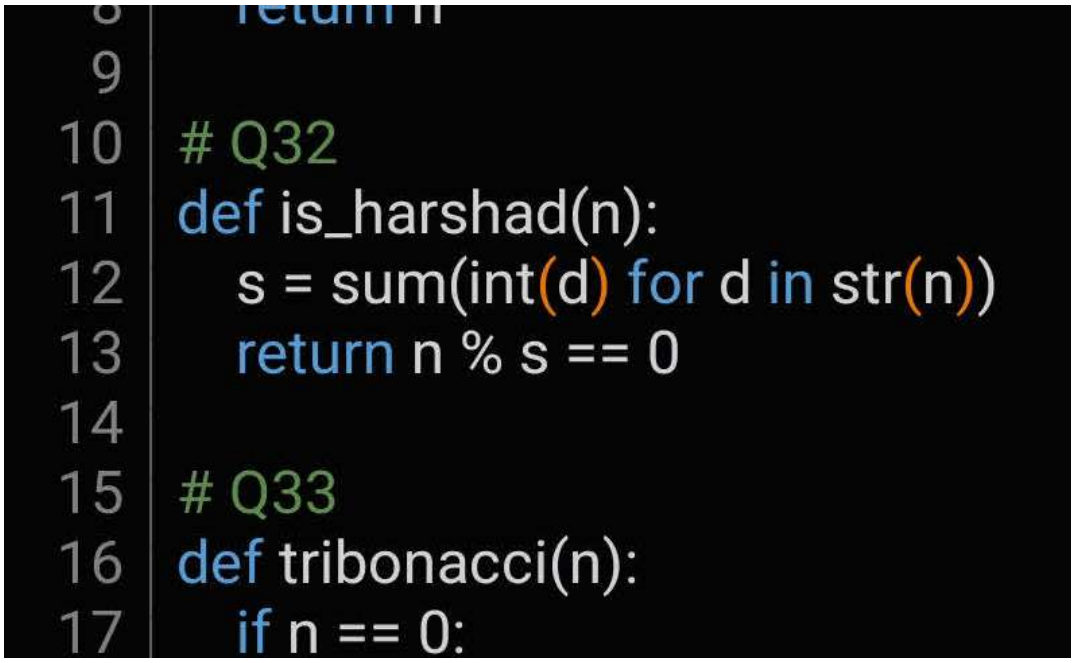
Difficulty Faced by Students

Students might find it hard to:

- Understand what Pollard's rho is and how the random walks help.
- Get the GCD and modulo parts right without errors.
- Know why it restarts if it hits *n*and how to handle that.
- Debug when it doesn't find a factor (it depends on random choices).

Conclusion

The pollard_rho(n) function is a cool way to crack numbers into factors using a random path game! It might be a bit tricky at first to learn the steps and why it uses random numbers, but with practice, students can get it. It's a fun math adventure to break down big numbers, especially with some trial and error



PRACTICAL – 3

DATE - 15/11/25

TITLE : Write a function zeta_approx(s, terms) that approximates

the Riemann zeta function ζ(s) using the first 'terms' of

the series.

Aim :

The goal is to make a function called zeta_approx(s, terms) that gives us a rough guess of the Riemann zeta function $\zeta(s)$. This is a special math idea that adds up numbers like $1 + 1/2^s + 1/3^s + 1/4^s + \cdots$, and we want to use the first $terms$numbers to get close to the real answer!

Methodology :

Here's how the function works:

- It checks if $terms$is 0 or less. If so, it returns 0 because we need at least one term to start.
- It starts with a result of 0 and loops from 1 to $terms$.
- For each number n, it adds $1/n^s$to the result (where $s$is the power we use).
- After adding all the terms, it gives back the total as our guess.

Brief Description :

The function takes two things: s(a number or special math value to raise to) and $terms$(how many numbers to add). For example, if $s = 2$and $terms = 5$, it adds $1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + 1/5^2$. That's $1 + 1/4 + 1/9 + 1/16 + 1/25$, which is about 1.4636. So, zeta_approx(2, 5) gives something close to that. The real $\zeta(2)$is about 1.6449, and more terms make it closer!

Result Achieved :

The function gives a good guess with the terms we pick. Some examples are:

- zeta_approx(2, 5) returns about 1.4636 (close to $\zeta(2)$with 5 terms).
- zeta_approx(2, 10) returns about 1.5498 (closer with 10 terms).
- zeta_approx(1, 5) returns a big number (like 5.579, but $\zeta(1)$is infinite, so it's just a partial sum).

- zeta_approx(2, 0) returns 0 (no terms to add). It works better with more terms, but it's not exact!

Difficulty Faced by Students :

Students might find it hard to:

- Understand what the Riemann zeta function is and why it adds these fractions.
- Write the loop to add the terms without messing up the power part.
- Know that more terms make it closer, but it's never perfect.
- Handle cases where *s*is a weird number (like negative or imaginary, though this version keeps it simple).

Conclusion :

```
13      return n % s == 0
14
15  # Q33
16  def tribonacci(n):
17      if n == 0:
18          return [0]
19      if n == 1:
20          return [0, 1]
21      if n == 2:
22          return [0, 1, 1]
23      seq = [0, 1, 1]
24      for i in range(3, n):
25          seq.append(seq[-1] + seq[-2] + seq[-3])
26      return seq
27
```

PRACTICAL – 4

DATE – 15/11/25

TITLE : Write a function Partition Function

p(n) partition_function(n) that calculates the number of

distinct ways to write n as a sum of positive integers.

Aim

The goal is to make a function called partition_function(n) that figures out how many different ways we can add up positive numbers to get n. For example, for $n = 4$, we want to count all the ways like 4, 3+1, 2+2, 2+1+1, 1+1+1+1. It's a fun counting game!

Methodology

Here's how the function works:

- **It checks if $n$is 0 or less. If so, it returns 0 because we can't make partitions with negative or zero numbers.**
- **It makes a list called $p$to keep track of how many ways we can make each number up to n, starting with $p[0] = 1$(one way to make 0, which is doing nothing).**
- **It uses two loops: one to go through each number $i$from 1 to n, and another to add $i$to all bigger numbers up to n.**
- **For each j, it adds the number of ways to make $j - i$to the ways to make j, building up the count.**
- **At the end, it gives back $p[n]$, the total ways to make n.**

Brief Description

The function takes one number n. For $n = 4$, it finds: 4, 3+1, 2+2, 2+1+1, 1+1+1+1, which are 5 ways. So, partition_function(4) returns 5. Another example, $n = 3$, gives 3 ways: 3, 2+1, 1+1+1, so it returns 3. The order of numbers doesn't matter, just the combination!

Result Achieved

The function counts the partitions correctly. Some examples are:

- **partition_function(4) returns 5 (5 ways to make 4).**
- **partition_function(3) returns 3 (3 ways to make 3).**

- **partition_function(1) returns 1 (only 1 way: 1).**
- **partition_function(0) returns 0 (not allowed). It gives the right number of ways every time!**

Difficulty Faced by Students

Students might find it hard to:

- **Understand what a partition is and why order doesn't count.**
- **Set up the *p*list and fill it with the right numbers.**
- **Get the double loop idea without mixing up *i*and *j*.**
- **Remember to handle 0 or negative numbers at the start.**

Conclusion

The partition_function(n) function is a cool way to count how many different piles we can make with numbers to get n! It might be a bit tricky at first to learn the loop trick, but with practice, students can do it easily. It's a fun math puzzle to play with and see all the ways numbers can add up

```
27
28  # Q34
29  def is_smith_number(n):
30      def prime_factors_sum(x):
31          total = 0
32          f = 2
33          while x > 1:
34              while x % f == 0:
35                  for digit in str(f):
36                      total += int(digit)
37                  x //= f
38              f += 1
39          return total
40      digit_sum = sum(int(d) for d in str(n))
41      return digit_sum == prime_factors_sum(n)
```

```
31: 3
32: True
33: [0, 1, 1, 2, 4, 7, 13, 24, 44, 81]
34: True

[Program finished]
```