# Lab Manual

## Practical and Skills Development

# CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

| | |
|---|---|
| **Registration No** | : 25BAI10858 |
| **Name of Student** | : Abhinav Singh |
| **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:

For Week 5,6,7,8 **Practical Index**

| S. No. | Title of Practical | Date of Submission | Signature of Faculty |
|---|---|---|---|
| 16 | Function to Calculate Aliquot Sum in Python | 13/11/25 | |
| 17 | Function to Check Amicable Numbers in Python | 13/11/25 | |
| 18 | Function to Calculate Multiplicative Persistence in Python | 13/11/25 | |
| 19 | Function to Check Highly Composite Numbers in Python | 13/11/25 | |
| 20 | Function to Modular Exponentiation in Python | 13/11/25 | |
| 21 | Function for Modular Multiplicative Inverse in Python | 13/11/25 | |
| 22 | Function for Chinese Remainder Theorem Solver in Python | 13/11/25 | |
| 23 | Function to Check Quadratic Residue in Python | 13/11/25 | |
| 24 | Function to Find the Order of a Number Modulo n in Python | 13/11/25 | |
| 25 | Function to Check Fibonacci Prime in Python | 13/11/25 | |
| 26 | Function to Generate Lucas Numbers in Python | 15/11/25 | |
| 27 | Function to Generate Lucas Numbers in Python | 15/11/25 | |
| 28 | Function to Calculate Collatz Sequence Length in Python | 15/11/25 | |
| 29 | Function to Calculate Polygonal Numbers in Python | 15/11/25 | |

| | | | |
|---|---|---|---|
| **30** | Function to Check Carmichael Numbers in Python | **15/11/25** | |
| **31** | Implement Miller-Rabin Probabilistic Primality Test in Python | **15/11/25** | |
| **32** | Implement Pollard's Rho Integer Factorization Algorithm in Python | **15/11/25** | |
| **33** | Function to Approximate Riemann Zeta Function in Python | **15/11/25** | |
| **34** | Function to Compute Partition Number in Python | **15/11/25** | |

**Practical No: 16**

**Date: __13/11/2025________**

**TITLE:**
Function to Calculate Aliquot Sum in Python

**AIM/OBJECTIVE(s):**
To write a function aliquot_sum(n) that returns the sum of all proper divisors of $n$ (excluding $n$ itself).

**METHODOLOGY & TOOL USED:**

- Methodology: Iterate from 1 to $n-1$, summing those which divide $n$ evenly.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The aliquot sum is the sum of all proper divisors of a number. For example, proper divisors of 12 are 1, 2, 3, 4, and 6; their sum is 16.

**RESULTS ACHIEVED:**
The function correctly calculates aliquot sums.
E.g., aliquot_sum(12) returns 16.

**DIFFICULTY FACED BY STUDENT:**

- Properly iterating and identifying divisors, especially for large numbers.
- Avoiding inclusion of the number itself.

**SKILLS ACHIEVED:**

- Practiced use of loops and conditionals for divisor calculations.
- Gained fundamental understanding of divisor functions and their applications in number theory.

**Practical No: 17**

**Date: ____13/11/2025______**

**TITLE:**
Function to Check Amicable Numbers in Python

**AIM/OBJECTIVE(s):**
To write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of $a$ equals $b$ and vice versa).

**METHODOLOGY & TOOL USED:**

- Methodology: Calculate the aliquot sum of both numbers and compare each result with the opposite number.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Amicable numbers are a pair where the sum of proper divisors of one equals the other, and vice versa. For example, 220 and 284 are amicable since sum of proper divisors of 220 is 284, and of 284 is 220.

**RESULTS ACHIEVED:**
The function correctly determines if two numbers are amicable, e.g., are_amicable(220, 284) returns True.

**DIFFICULTY FACED BY STUDENT:**

- Understanding the relationship of divisor sums between two numbers.
- Accurate calculation and comparison for large numbers or edge cases.

**SKILLS ACHIEVED:**

- Practiced sum calculations and relational logic with divisor concepts.
- Improved understanding of rare number pairs in number theory.

**Practical No: 18**

**Date: __13/11/2025________**

**TITLE:**
Function to Calculate Multiplicative Persistence in Python

**AIM/OBJECTIVE(s):**
To write a function multiplicative_persistence(n) that counts how many steps it takes until the digits of a number multiply to a single digit.

**METHODOLOGY & TOOL USED:**

- Methodology: While the number has more than one digit, repeatedly multiply its digits and increase a counter until the result is a single digit.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Multiplicative persistence refers to the number of times the digits of a number must be multiplied together until the result is a single digit. For example, for 39: $3 \times 9 = 27$, $2 \times 7 = 14$, $1 \times 4 = 4$. Total steps: 3.

**RESULTS ACHIEVED:**
The function successfully computes multiplicative persistence for any positive integer, e.g., multiplicative_persistence(39) returns 3.

**DIFFICULTY FACED BY STUDENT:**

- Extracting and multiplying digits correctly, handling transitions between intermediate results.
- Avoiding infinite loops and handling single-digit edge cases.

**SKILLS ACHIEVED:**

- Mastered digit manipulation and loop-based transformations.
- Learned about persistence concepts and iterative reduction in number theory.

**Practical No: 19**

**Date: __13/11/2025________**

**TITLE:**
Function to Check Highly Composite Numbers in Python.

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

**METHODOLOGY & TOOL USED:**

- Methodology: For each number less than $n$, count its positive divisors and compare with count for $n$; return True if $n$ is highest.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
A highly composite number has a greater quantity of positive divisors than all smaller natural numbers. For example, 12 has 6 divisors, which is more than any smaller number.

**RESULTS ACHIEVED:**
The function identifies highly composite numbers correctly,
e.g., is_highly_composite(12) returns True.

**DIFFICULTY FACED BY STUDENT:**

- Efficiently counting divisors for multiple numbers and comparing results.
- Managing computational time for larger values.

**SKILLS ACHIEVED:**

- Developed logic for divisor counting and comparative programming.
- Learned about special classes of natural numbers in mathematics.

**Practical No: 20**

**Date: __13/11/2025________**

**TITLE:**
Function for Modular Exponentiation in Python

**AIM/OBJECTIVE(s):**
To write a function mod_exp(base, exponent, modulus) that efficiently calculates $(base^{exponent}) \bmod modulus$.

**METHODOLOGY & TOOL USED:**

- Methodology: Use the technique of exponentiation by squaring to minimize calculations and handle large exponents or moduli.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Modular exponentiation is an efficient way to compute large powers modulo a number. For example, $(3^4) \bmod 5 = 1$. This function is essential for cryptographic and computational purposes.

**RESULTS ACHIEVED:**
The function successfully calculates modular exponentiation for any given inputs. Example: mod_exp(3, 4, 5) returns 1.

**DIFFICULTY FACED BY STUDENT:**

- Understanding how to optimize exponentiation for large powers in Python.
- Ensuring correct order of operations and modulus calculations.

**SKILLS ACHIEVED:**

- Mastered mathematical optimization techniques.
- Learned the use of modulo operator and recursive/iterative problem-solving.

**Practical No: 21**

**Date: __13/11/2025________**

**TITLE:**
Function for Modular Multiplicative Inverse in Python

**AIM/OBJECTIVE(s):**
To write a function mod_inverse(a, m) that finds a number $x$ such that $(a \times x) \equiv 1 \text{mod } m$.

**METHODOLOGY & TOOL USED:**

- Methodology: Use the Extended Euclidean Algorithm to find the modular inverse of $a$ modulo $m$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The modular multiplicative inverse is an integer $x$ such that $(a \times x) \text{mod } m = 1$. This concept is essential in cryptography and number theory.

**RESULTS ACHIEVED:**
The function correctly finds modular inverses; for example, mod_inverse(3, 11) returns 4 because $3 \times 4 \equiv 1 \text{mod } 11$.

**DIFFICULTY FACED BY STUDENT:**

- Implementing the Extended Euclidean Algorithm properly.
- Handling cases where the inverse does not exist (when $a$ and $m$ are not coprime).

**SKILLS ACHIEVED:**

- Learned algorithmic implementation and handling modular mathematics.
- Improved understanding of modular arithmetic basics.

**Practical No: 22**

**Date: __13/11/2025________**

**TITLE:**
Function for Chinese Remainder Theorem Solver in Python

**AIM/OBJECTIVE(s):**
To write a function crt(remainders, moduli) that solves a system of congruences $x \equiv r_i \bmod m_i$ for integer $x$.

**METHODOLOGY & TOOL USED:**

- Methodology: Use the Chinese Remainder Theorem algorithm to compute a solution modulo the product of all given moduli, provided the moduli are pairwise coprime.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The Chinese Remainder Theorem (CRT) offers a way to find a solution to simultaneous modular equations. For example, solving $x \equiv 2 \bmod 3$, $x \equiv 3 \bmod 5$, and $x \equiv 2 \bmod 7$.

**RESULTS ACHIEVED:**
The function finds the smallest $x$ satisfying all the given congruences. Example: For remainders and moduli , solution is $x = 23$.

**DIFFICULTY FACED BY STUDENT:**

- Implementing modular arithmetic and the CRT algorithm for arbitrary systems.
- Ensuring moduli are coprime and handling invalid input gracefully.

**SKILLS ACHIEVED:**

- Mastered systems of modular equations and CRT logic.
- Improved skills in code-based number theory problem-solving.

**Practical No: 23**

**Date: __13/11/2025________**

**TITLE:**

Function to Check Quadratic Residue in Python

**AIM/OBJECTIVE(s):**

To write a function is_quadratic_residue(a, p) that checks if $x^2 \equiv a \bmod p$ has a solution.

**METHODOLOGY & TOOL USED:**

- Methodology: For each integer $x$ from 0 to $p-1$, check if $(x^2) \bmod p = a$. Alternatively, use Euler's Criterion if $p$ is an odd prime.
- Tool: Python programming language

**BRIEF DESCRIPTION:**

A quadratic residue modulo $p$ is a number $a$ for which there exists an integer $x$ such that $x^2 \equiv a \bmod p$. For example, 4 is a quadratic residue mod 7 because $2^2 \equiv 4 \bmod 7$.

**RESULTS ACHIEVED:**

The function correctly determines quadratic residues.
Example: is_quadratic_residue(4, 7) returns True,
while is_quadratic_residue(3, 7) returns False.

**DIFFICULTY FACED BY STUDENT:**

- Efficiently iterating possible values or correctly applying Euler's Criterion.
- Understanding and handling cases where $p$ is not prime.

**SKILLS ACHIEVED:**

- Gained experience in residue checking, brute-force and optimization techniques.
- Improved modular arithmetic reasoning and mathematical programming.

**Practical No: 24**

**Date: __13/11/2025________**

**TITLE:**
Function to Find the Order of a Number Modulo n in Python

**AIM/OBJECTIVE(s):**
To write a function order_mod(a, n) that finds the smallest positive integer $k$ such that $a^k \equiv 1 \bmod n$.

**METHODOLOGY & TOOL USED:**

- Methodology: Starting from $k = 1$, successively compute $a^k \bmod n$ until the result is 1, returning the smallest $k$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The order of $a$ modulo $n$ is the least positive integer $k$ for which $a^k$ is congruent to 1 modulo $n$. This is important in group theory and cryptography.

**RESULTS ACHIEVED:**
The function correctly computes orders. Example: order_mod(2, 7) returns 3 since $2^3 = 8 \equiv 1 \bmod 7$.

**DIFFICULTY FACED BY STUDENT:**

- Implementing efficient exponentiation and loop structure.
- Recognizing when no such $k$ exists.

**SKILLS ACHIEVED:**

- Mastered loop-based search and modular exponentiation.
- Gained understanding of group order concepts in number theory.

**Practical No: 25**

**Date: __13/11/2025________**

**TITLE:**
Function to Check Fibonacci Prime in Python

**AIM/OBJECTIVE(s):**
To write a function is_fibonacci_prime(n) that checks if a number is both a Fibonacci number and prime.

**METHODOLOGY & TOOL USED:**

- Methodology: First, check if $n$ is a Fibonacci number using mathematical properties or iteration; then verify if $n$ is prime.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
A Fibonacci prime is a number that appears in the Fibonacci sequence and is also a prime. For example, 13 is both a Fibonacci number and prime.

**RESULTS ACHIEVED:**
The function accurately identifies Fibonacci primes.
Example: is_fibonacci_prime(13) returns True; is_fibonacci_prime(21) returns False.

**DIFFICULTY FACED BY STUDENT:**

- Efficiently checking the Fibonacci property for any $n$.
- Writing a robust prime checking function.

**SKILLS ACHIEVED:**

- Practiced combining two mathematical concepts in a single algorithm.
- Improved efficiency in sequence checking and primality logic.

**Practical No: 26**

**Date: __15/11/2025________**

**TITLE:**
Function to Generate Lucas Numbers in Python

**AIM/OBJECTIVE(s):**
To write a function lucas_sequence(n) that generates the first $n$ Lucas numbers (sequence starts with 2, 1, like Fibonacci).

**METHODOLOGY & TOOL USED:**

- Methodology: Use a loop to iteratively sum the two previous terms, initializing with 2 and 1.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Lucas numbers follow a recurrence relation similar to Fibonacci,
with $L_0 = 2$, $L_1 = 1$, and $L_n = L_{n-1} + L_{n-2}$ for $n \geq 2$.

**RESULTS ACHIEVED:**
The function generates correct sequences.
Example: lucas_sequence(5) returns [2, 1, 3, 4, 7].

**DIFFICULTY FACED BY STUDENT:**

- Correctly handling starting values different from Fibonacci.
- Implementing iterative logic for sequence generation.

**SKILLS ACHIEVED:**

- Practiced sequence generation and recurrence relations.
- Improved loop logic for dynamic programming problems.

**Practical No: 27**

**Date: __15/11/2025________**

**TITLE:**
Function to Check Perfect Powers in Python

**AIM/OBJECTIVE(s):**
To write a function is_perfect_power(n) that checks if a number can be expressed as $a^b$ where $a > 0$ and $b > 1$.

**METHODOLOGY & TOOL USED:**

- Methodology: For possible exponents $b$ starting from 2 up to $\log_2(n)$, check if there exists some integer $a$ such that $a^b = n$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
A perfect power is a number that can be written as a positive integer raised to a power greater than 1. For example, 27 is a perfect power because $3^3 = 27$.

**RESULTS ACHIEVED:**
The function accurately checks perfect powers.
Example: is_perfect_power(27) returns True,
while is_perfect_power(14) returns False.

**DIFFICULTY FACED BY STUDENT:**

- Efficient exponentiation and root calculation.
- Looping and handling edge cases for different powers.

**SKILLS ACHIEVED:**

- Improved proficiency in exponent and root logic implementation.
- Gained skills in mathematical validation algorithms.

**Practical No: 28**

**Date: 15/11/2025**

**TITLE:**
Function to Calculate Collatz Sequence Length in Python

**AIM/OBJECTIVE(s):**
To write a function collatz_length(n) that returns the number of steps for $n$ to reach 1 in the Collatz conjecture.

**METHODOLOGY & TOOL USED:**

- Methodology: While $n$ is not 1, apply $n/2$ if $n$ is even or $3n + 1$ if odd, counting steps until $n = 1$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The Collatz sequence is built by the following process: if $n$ is even, divide by 2; if odd, multiply by 3 and add 1; repeat until $n$ equals 1, counting the steps.

**RESULTS ACHIEVED:**
The function computes the sequence length for any input.
Example: collatz_length(6) returns 8 (steps: 6, 3, 10, 5, 16, 8, 4, 2, 1).

**DIFFICULTY FACED BY STUDENT:**

- Handling the rapid growth or reduction of $n$ and large intermediate values.
- Correct loop and condition logic for all possible $n$.

**SKILLS ACHIEVED:**

- Practiced iterative logic and conditional branching in Python.
- Learned about famous unsolved problems in mathematics.

**Practical No: 29**

**Date: 15/11/2025**

**TITLE:**
Function to Calculate Polygonal Numbers in Python

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

**METHODOLOGY & TOOL USED:**

- Methodology: Compute using formula for the $n$-th $s$-gonal number: $P(s,n) = \frac{n[(s-2)(n-1)+2]}{2}$
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Polygonal numbers generalize triangular and square numbers to any number of sides ($s$), using a mathematical formula.

**RESULTS ACHIEVED:**
Function correctly computes polygonal numbers.
Example: polygonal_number(3, 5) returns 15 (5th triangular number).

**DIFFICULTY FACED BY STUDENT:**

- Proper application of the formula for different $s$ values.
- Ensuring integer results and handling special cases (e.g., $s = 3$ for triangles).

**SKILLS ACHIEVED:**

- Learned generalization of numeric sequences.
- Enhanced skill in direct mathematical formula implementation.

**Practical No: 30**

**Date: 15/11/2025**

**TITLE:**
Function to Check Carmichael Numbers in Python

**AIM/OBJECTIVE(s):**
To write a function is_carmichael(n) that checks if a composite number $n$ satisfies $a^{n-1} \equiv 1 \bmod n$ for all $a$ coprime to $n$.

**METHODOLOGY & TOOL USED:**

- Methodology: Confirm if $n$ is composite, then for each integer $a$ coprime to $n$, verify $a^{n-1} \bmod n = 1$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
Carmichael numbers exhibit Fermat-like properties for all coprime bases despite being composite. Example: 561 is the smallest Carmichael number.

**RESULTS ACHIEVED:**
The function accurately identifies Carmichael numbers.
Example: is_carmichael(561) returns True.

**DIFFICULTY FACED BY STUDENT:**

- Efficient prime and coprimality checking.
- Exponential time for large values; optimizing coprime enumeration.

**SKILLS ACHIEVED:**

- Gained exposure to exceptional numbers in number theory.
- Learned robust modular exponentiation and gcd checks.

**Practical No: 31**

**Date: 15/11/2025**

**TITLE:**
Implement Miller-Rabin Probabilistic Primality Test in Python

**AIM/OBJECTIVE(s):**
To write a function is_prime_miller_rabin(n, k) that tests if $n$ is prime using the Miller-Rabin algorithm for $k$ rounds.

**METHODOLOGY & TOOL USED:**

- Methodology: Decompose $n-1$ into $2^r \cdot d$ (with $d$ odd), select random bases, and use strong probable prime checks for $k$ rounds.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The Miller-Rabin test is a randomized algorithm providing probabilistic evidence of primality. Increasing the number of rounds $k$ improves confidence in correctness.

**RESULTS ACHIEVED:**
The function probabilistically determines primality,
e.g., is_prime_miller_rabin(29, 5) returns True (likely prime).

**DIFFICULTY FACED BY STUDENT:**

- Careful decomposition of $n-1$.
- Implementing probabilistic branching and efficient modular exponentiation.

**SKILLS ACHIEVED:**

- Mastered probabilistic number theory methods.
- Practiced algorithmic implementation for cryptographic and mathematical purposes.

**Practical No: 32**

**Date: 15/11/2025**

**TITLE:**

Implement Pollard's Rho Integer Factorization Algorithm in Python

**AIM/OBJECTIVE(s):**

To write a function pollard_rho(n) that uses Pollard's rho algorithm for integer factorization.

**METHODOLOGY & TOOL USED:**

- Methodology: Use the sequence $x_{i+1} = (x_i^2 + 1) \bmod n$, compute gcd of $| x_i - y_i |$ and $n$ to seek a non-trivial factor.
- Tool: Python programming language

**BRIEF DESCRIPTION:**

Pollard's rho is a probabilistic factorization algorithm effective for composite numbers, particularly with small factors.

**RESULTS ACHIEVED:**

The function finds non-trivial divisors for composite inputs.
Example: pollard_rho(8051) returns 97 (a factor of 8051).

**DIFFICULTY FACED BY STUDENT:**

- Understanding and implementing a probabilistic approach.
- Efficient handling of cycles and large integers.

**SKILLS ACHIEVED:**

- Learned factorization concepts beyond trial division.
- Practiced randomization and algorithmic optimization.

**Practical No: 33**

**Date: 15/11/2025**

**TITLE:**
Function to Approximate Riemann Zeta Function in Python

**AIM/OBJECTIVE(s):**
To write a function zeta_approx(s, terms) that approximates the Riemann zeta function $\zeta(s)$ using the first specified number of terms of the series.

**METHODOLOGY & TOOL USED:**

- Methodology: Calculate the sum $\zeta(s) = \sum_{n=1}^{N} \frac{1}{n^s}$ for the requested number of terms $N$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The Riemann zeta function is defined as a convergent series for $s > 1$ and is a fundamental concept in analytic number theory.

**RESULTS ACHIEVED:**
The function returns close approximations. For example, zeta_approx(2, 1000) gives a value near $\pi^2/6$.

**DIFFICULTY FACED BY STUDENT:**

- Managing numerical precision for large term counts.
- Understanding series and summation concepts.

**SKILLS ACHIEVED:**

- Practiced series implementation and convergence analysis.
- Improved mathematical programming skills for approximation techniques.

**Practical No: 34**

**Date: 15/11/2025**

**TITLE:**
Function to Compute Partition Number in Python

**AIM/OBJECTIVE(s):**
To write a function partition_function(n) that calculates the number of distinct ways to write $n$ as a sum of positive integers.

**METHODOLOGY & TOOL USED:**

- Methodology: Use recursive dynamic programming (or generating functions) to count the number of partitions for any $n$.
- Tool: Python programming language

**BRIEF DESCRIPTION:**
The partition function $p(n)$ counts the number of unique ways an integer can be decomposed into sums of smaller integers, disregarding order.

**RESULTS ACHIEVED:**
The function yields accurate partition numbers.
Example: partition_function(4) returns 5 (ways: 4; 3+1; 2+2; 2+1+1; 1+1+1+1).

**DIFFICULTY FACED BY STUDENT:**

- Developing recursion or memoization for efficient calculation.
- Understanding combinatorics and additive number theory concepts.

**SKILLS ACHIEVED:**

- Mastered recursion and dynamic programming for combinatorial problems.
- Learned applications in pure and applied mathematics.