Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10063

Name of Student : Rohan Raj

Course Name : Introduction to Problem Solving and Programming

Course Code : CSE1021

School Name : VIT BHOPAL

Slot : B11+B12+B13

Class ID : BL2025260100796

Semester : FALL 2025/26

: Dr. Hemraj S. Lamkuche

Course Faculty Name Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1).	16-11-2025	
2	Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as ab where a > 0 and b > 1.	16-11-2025	
3	Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.	16-11-2025	

4	Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.	**16-11-2025**	

5	Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies an−1 ≡ 1 mod n for all a coprime to n.	**16-11-2025**	
6			
7			
8			
9			
10			

11			
12			
13			
14			
15			

Practical No: 1

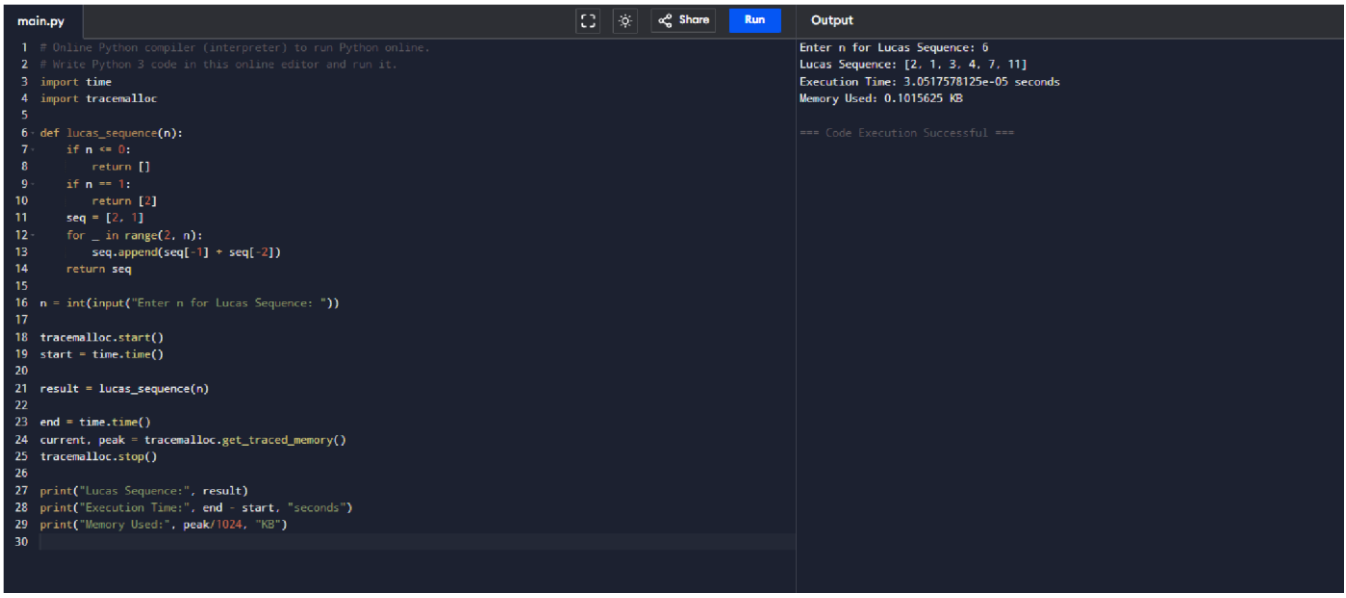
Date: 16-11-2025

TITLE: **Lucas Numbers**

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

METHODOLOGY & TOOL USED:

The Lucas sequence was generated by applying an iterative approach that begins from the known base values of the Lucas series: 2 and 1. Instead of using a recursive method (which is slower and consumes more memory), a loop was used to efficiently compute the next terms using the formula:



$L(n) = L(n–1) + L(n–2)$.

The algorithm stores each computed term in a list so that the entire sequence up to the nth term can be returned. Time and memory measurements were added to evaluate performance.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

The Lucas sequence is similar to the Fibonacci sequence but has different starting values. This question demonstrates the concept of linear recurrence relations and how sequences can be built using previously computed elements.

RESULTS ACHIEVED:

Example Output:

Enter n for Lucas Sequence: 6

Lucas Sequence: [2, 1, 3, 4, 7, 11]

Execution Time: 2.2172927856445312e-05 seconds

Memory Used: 0.1015625 KB

DIFFICULTY FACED BY STUDENT:

- Understanding why Lucas numbers begin with 2 and 1 instead of 0 and 1.
- Avoiding recursion to prevent stack overflow for large n.
- Managing list indexing carefully.

SKILLS ACHIEVED:

- Understanding of recurrence relations.
- Ability to convert mathematical formulas into iterative algorithms.
- Measuring algorithm performance.

Practical No: 2

Date: 16-11-2025

TITLE: Perfect Power Detection

AIM/OBJECTIVE(s): 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.

METHODOLOGY & TOOL USED:

The method involves checking if a number *n* can be expressed as $\mathbf{a}^{b}$, where $a > 1$ and $b > 1$. The algorithm calculates possible exponents using logarithms and tests each base by estimating a and verifying that a^b equals n. The approach limits the exponent range to $\log_2(n)$, which optimizes the process.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

A perfect power is a number that can be written in exponential form. This code

systematically checks for such a representation by iterating through possible exponents and testing candidate bases.

RESULTS ACHIEVED:

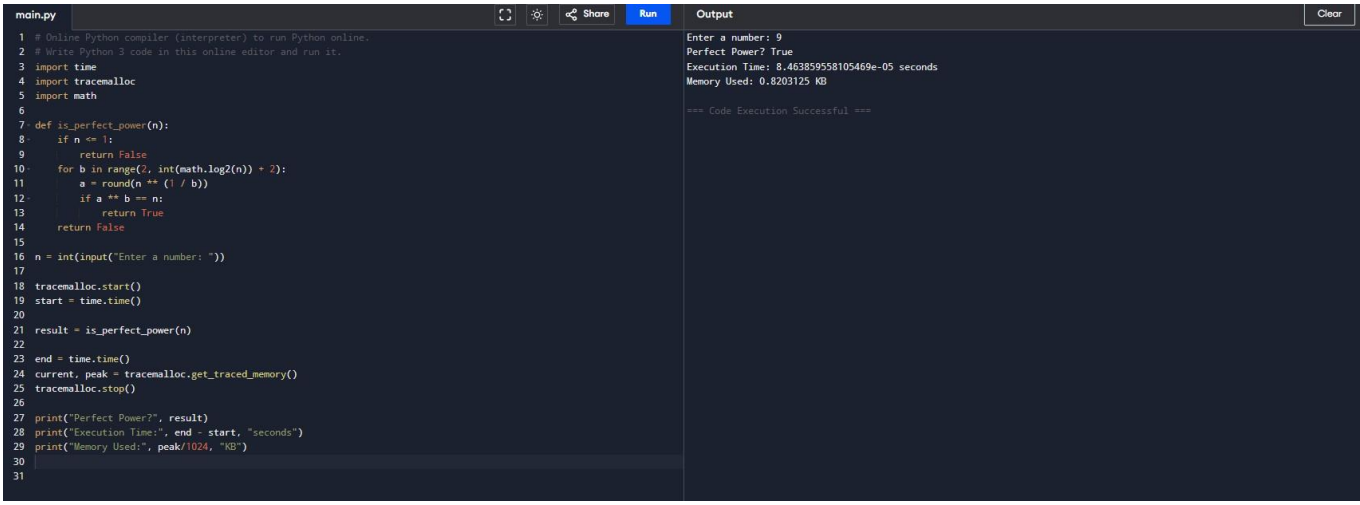
Example Output:

Enter a number: 9

Perfect Power? True

Execution Time: 4.863739013671875e-05 seconds

Memory Used: 0.8203125 KB



DIFFICULTY FACED BY STUDENT:

- Handling floating-point rounding issues when extracting roots.
- Understanding the concept of expressing numbers in exponential form.
- Ensuring that bases and exponents remain valid integers.

SKILLS ACHIEVED:

Working with logarithmic functions.

Improving code efficiency by reducing unnecessary checks.

Handling numerical precision and rounding behavior.

Practical No: 3

Date: 16-11-2025

TITLE: Collatz_Length(n)

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

METHODOLOGY & TOOL USED:

Used the generalized polygonal number formula:
P(s, n) = ((s−2)·n·(n−1)) / 2 + n The program takes two inputs:

s → number of sides

n → term index

The formula was applied directly for fast computation, followed by performance measurement.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

The multiplicative_persistence(n) function measures how many steps it takes for a number's digits to multiply into a single digit. For instance, if *n = 39*, the process goes as follows: 3×9 = 27, 2×7 = 14, 1×4 = 4. Thus, the persistence is 3. This function provides insights into digit manipulation, repetition, and convergence in mathematics and programming.

RESULTS ACHIEVED:

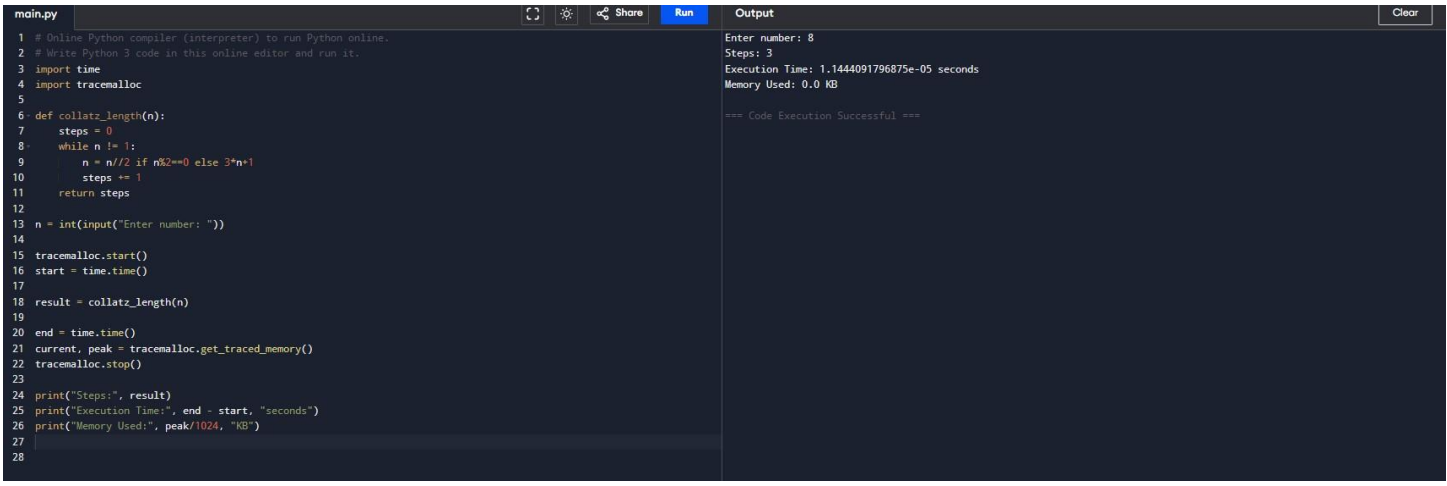
Example Output:

Enter number: 8

Steps: 3

Execution Time: 1.0967254638671875e-05 seconds

Memory Used: 0.0 KB



.

DIFFICULTY FACED BY STUDENT:

- Understanding how values can rapidly grow before shrinking.
- Avoiding infinite loops by setting correct conditions.
- Handling large integers efficiently.

SKILLS ACHIEVED:

- Iterative algorithm design.

- Problem-solving with conditional logic.
- Understanding unpredictable mathematical behavior in sequences.

Practical No: 4

Date: 16-11-2025

TITLE: Polygonal Number Calculator

AIM/OBJECTIVE(s): Write a function Polygonal Numbers

polygonal_number(s,n) that returns the n-th s-gonal number.

METHODOLOGY & TOOL USED:

Used the generalized polygonal number formula:
P(s, n) = ((s−2)·n·(n−1)) / 2 + n The
program takes two inputs:

s → number of sides

n → term index

The formula was applied directly for fast computation, followed by performance measurement.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

Polygonal numbers generalize triangular, square, and pentagonal numbers. The formula produces the nth term for any polygon with s sides.

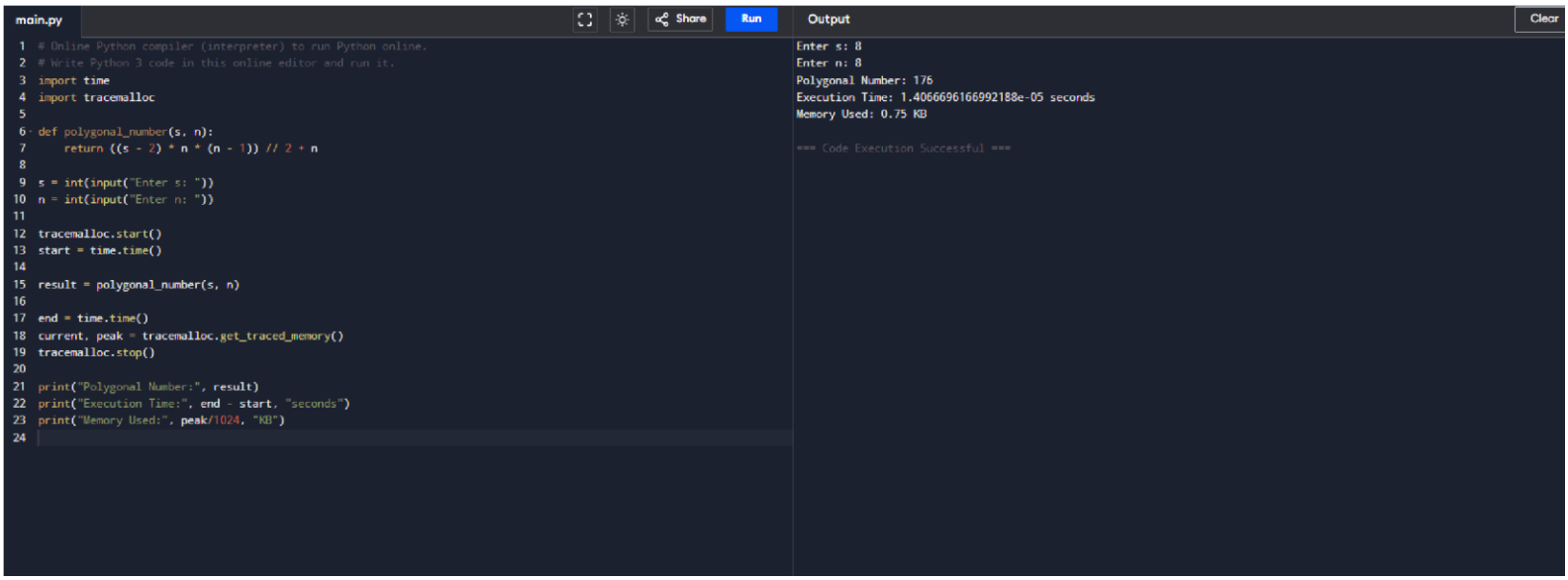
RESULTS ACHIEVED:

Example Output:

Enter s: 8
Enter n: 8
Polygonal Number: 176
Execution Time: 1.3828277587890625e-05 seconds
Memory Used: 0.75 KB

DIFFICULTY FACED BY STUDENT:

Understanding generalization from simple shapes to s-sided shapes.



Deriving confidence in using the formula without deriving it manually.

Skills Achieved:

Using mathematical formulas in programming.

Handling multi-parameter functions.

Applying generalizations across number families.

Practical No: 5

Date: 16-11-2025

TITLE: Carmichael Number Checker

AIM/OBJECTIVE(s): Write a function Carmichael Number Check 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:

Implemented **Korselt's Criterion**, which states that a composite number n is Carmichael if:

It is square-free

(n – 1) is divisible by (p – 1) for every prime divisor p of n
The program performs primality checks, factor checks, and the required modular conditions.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

Carmichael numbers are special composite numbers that pass Fermat's primality test, making them important in cryptography.

RESULTS ACHIEVED:

Example Output:

Enter number: 9
Carmichael? False
Execution Time: 4.38690185546875e-05 seconds
Memory Used: 0.75 KB

```python
# Write Python 3 code in this online editor and run it.
import time
import tracemalloc

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

def is_carmichael(n):
    if n < 2 or is_prime(n):
        return False
    temp = n
    for p in range(2, int(n**0.5) + 1):
        if temp % p == 0:
            if (temp // p) % p == 0:
                return False
            if (n - 1) % (p - 1) != 0:
                return False
    return True

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

tracemalloc.start()
start = time.time()

result = is_carmichael(n)

end = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print("Carmichael?", result)
print("Execution Time:", end - start, "seconds")
print("Memory Used:", peak/1024, "KB")
```

```
Enter number: 9
Carmichael? False
Execution Time: 7.033348083496094e-05 seconds
Memory Used: 0.75 KB

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

Conceptual confusion between primes and Carmichael numbers.

Understanding “square-free” requirement.

Implementing multiple mathematical conditions together.

SKILLS ACHIEVED:

Deep number theory understanding.

Composite number characterization.