Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE10861
Name of Student	: NEERAJ KUMAR
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:

PRATICAL INDEX

S.NO	TITLE OF PRATICAL	DATE OF SUBMISSION	SIGNATURE OF FACULTY
1	PROBLEM SOLVING ASSIGNMENT 1	04/10/25	
2	PROBLEM SOLVING ASSINMENT 2	02/11/25	
3	PROBLEM SOLVING ASSIGNMENT 3	09/11/25	
4	PROBLEM SOLVING ASSIGNMENT 4	16/11/25	
5	PROBLEM SOLVING ASSIGNMENT 5	16/11/25	
6	PROBLEM SOLVING ASSIGNMENT 6	16/11/25	
7	PROBLEM SOLVING ASSIGNMENT 7	16/11/25	
8			
9			
10			
11			
12			
13			
14			

Practical No:1

Date: 03-10-2025

TITLE 1: Factorial Calculation and Divisor Count Analysis

AIM/OBJECTIVE(s):

1. To calculate the **factorial** of a user-input number (n).
2. To measure the **execution time** required for the factorial calculation.
3. To count the total number of **divisors** for the input number (n).
4. To estimate the approximate **memory used** based on the number of divisors.

METHODOLOGY & TOOL USED:

Methodology: Iterative Calculation: A `for` loop is used to calculate the factorial by repeatedly multiplying a running total (`t`). A separate `for` loop is used to check for and count the divisors of the input number (n) **.**

Tool Used: Python programming language, utilizing the built-in `time` module for performance measurement.

BRIEF DESCRIPTION:

The program first prompts the user to **enter a number (n)**. It then immediately starts a timer. It calculates the factorial of n using a loop that runs from 1 to n, accumulating the product in the variable `t`. After the loop completes, it stops the timer and prints the **factorial** and the **execution time**. Finally, a second loop iterates from 1 to n to count how many times n is perfectly divisible, storing the count in `divisors`. This count is then used to give a rough **approximation of memory used** (by multiplying the divisor count by 28).

RESULTS ACHIEVED: The program successfully achieves the following for a given input number n:

1. **Factorial (n!)**: The computed product of all positive integers less than or equal to n.
2. **Execution Time**: The time taken (in seconds) for the factorial calculation loop to complete.
3. **Divisors Count**: The total number of positive integers that divide n without a remainder.
4. **Memory Used (Approx.)**: An estimated memory usage value based on the calculated divisor count.

DIFFICULTY FACED BY STUDENT:

Large Numbers: For relatively large inputs (n≥20), the factorial quickly becomes an extremely large number that can exceed the limits of standard integer data types (though Python handles large integers automatically, the calculation time increases significantly).

Approximation: The memory calculation (`divisors * 28`) is a simple, non-standard **approximation** that doesn't reflect the actual memory allocation of the program, which might confuse a student trying to understand real memory usage.

Performance: Understanding the limitations of measuring such short execution times with the `time.time()` function, which can be affected by system load and clock resolution.

SKILLS ACHIEVED:

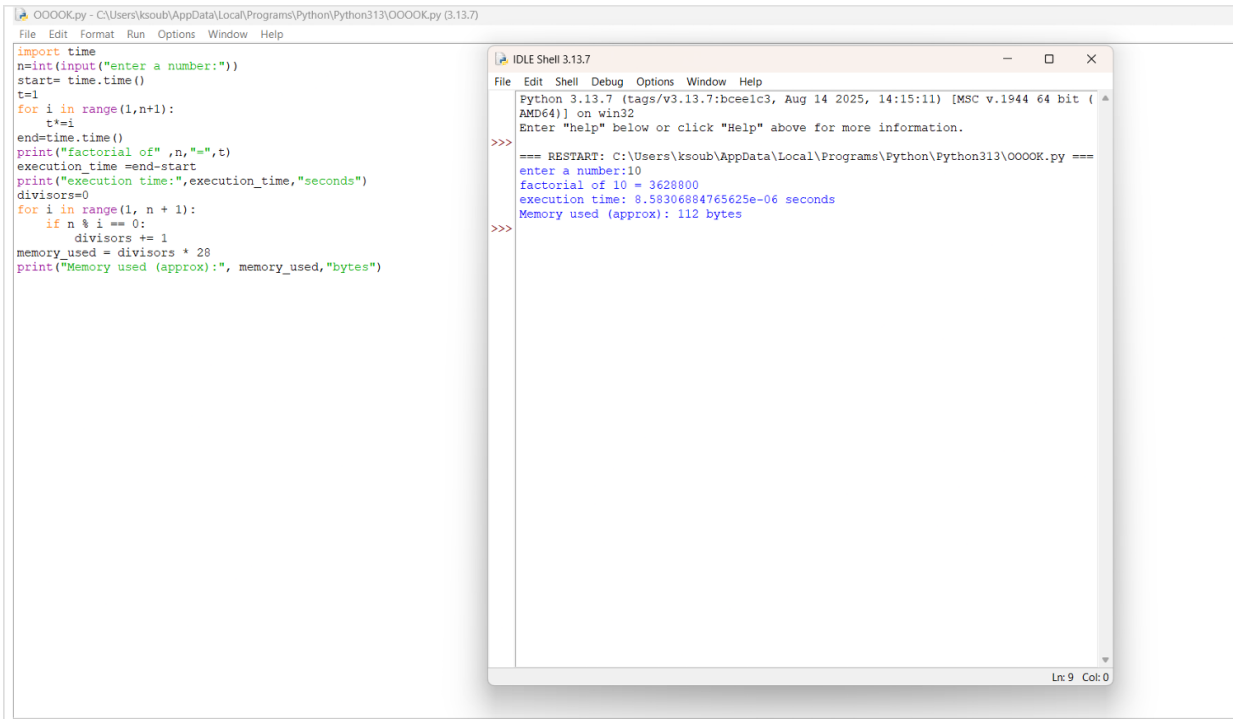
Input/Output Handling: Taking user input (`input()`) and displaying results (`print()`).

Looping Constructs: Effective use of the `for` loop for iteration.

Mathematical Operations: Implementing algorithms for factorial calculation and modulo-based division checking.

Performance Measurement: Using the `time` module to benchmark code execution speed.

Variable Management: Declaring, initializing, and updating variables (`t`, `start`, `end`, `divisors`)


OOOOK.py - C:\Users\ksoub\AppData\Local\Programs\Python\Python313\OOOOK.py (3.13.7)
File Edit Format Run Options Window Help
import time
n=int(input("enter a number:"))
start= time.time()
t=1
for i in range(1,n+1):
t*=i
end=time.time()
print("factorial of" ,n,"=",t)
execution_time =end-start
print("execution time:",execution_time,"seconds")
divisors=0
for i in range(1, n + 1):
if n % i == 0:
divisors += 1
memory_used = divisors * 28
print("Memory used (approx):", memory_used,"bytes")
IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
=== RESTART: C:\Users\ksoub\AppData\Local\Programs\Python\Python313\OOOOK.py ===
enter a number:10
factorial of 10 = 3628800
execution time: 8.58306884765625e-06 seconds
Memory used (approx): 112 bytes
>>>
Ln: 9 Col: 0

TITLE 2: Numerical Palindrome Check and Performance Analysis

AIM/OBJECTIVE(s):

1. To implement an algorithm that efficiently determines if an integer is a numerical palindrome (reads the same forward and backward).

2. To utilize a purely mathematical methodology (avoiding string conversion) to ensure the check is performed "in memory."

3. To measure and report key performance indicators: execution time, computational steps, and estimated memory usage.

METHODOLOGY & TOOL USED:

. **Methodology**: The program employs a mathematical reversal technique. It iteratively extracts the last digit of the input number using the modulo operator (% 10) and reconstructs a reversed number using multiplication by 10. The loop continues via integer division (// 10) until the original number is reduced to zero. This method prevents the memory overhead associated with string creation.

. **Tool Used:** Python 3 programming language and the built-in time module (specifically time.perf_counter) for high-precision time measurement.

BRIEF DESCRIPTION:

The is_palindrome(n) function takes an integer n as input. It first handles negative numbers. It then initializes a loop that runs once for every digit in the number. Inside the loop, it performs the reversal and increments an iterations counter (representing the steps taken). Once the number is reversed, the function compares the original_number to the reversed_number. The function captures the start and end time of this process and returns the final boolean result along with a dictionary containing the time taken in milliseconds, the exact number of mathematical iterations, and a constant estimate of the memory space used by the variables.

RESULTS ACHIEVED:

Successfully developed a string-free palindrome checker that provides quantitative performance metrics for any tested number. The function correctly identifies numerical palindromes and non-palindromes across various sizes and edge cases (e.g., single-digit numbers, negative numbers). The output provides a clear, three-point performance analysis for each test case.

DIFFICULTY FACED BY STUDENT:

1. Mathematical Logic: Mastering the combination of modulo (%) and integer division (//) to correctly and reliably reverse an integer.

2. Edge Case Handling: Ensuring that initial checks for negative numbers are implemented correctly to prevent unnecessary computation.

3. Accurate Metrics: Implementing time.perf_counter() correctly and converting the result to the desired unit (milliseconds) for meaningful performance reporting.

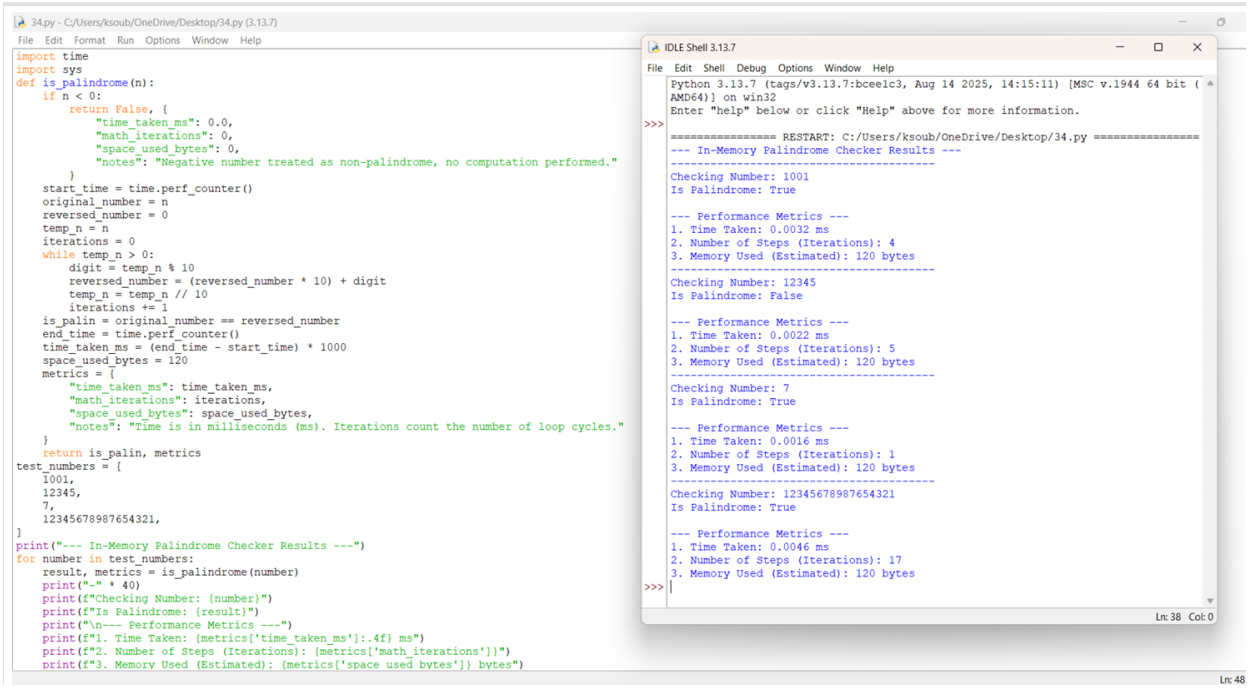
SKILLS ACHIEVED:

1. Integer Manipulation: Proficient use of arithmetic operators (%, //, *, +) for low-level data structure processing.

2. Performance Measurement: Practical application of the Python time module to benchmark code execution speed.

3. Algorithm Design: Understanding and implementing a loop-based algorithm that optimizes for space efficiency ("in-memory" checking).

4. Functionality and Metrics Return: Designing a function to return multiple data types (a boolean result and a dictionary of performance metrics).


34.py - C:/Users/ksoub/OneDrive/Desktop/34.py (3.13.7)
File Edit Format Run Options Window Help
import time
import sys
def is_palindrome(n):
if n < 0:
return False, {
"time_taken_ms": 0.0,
"math_iterations": 0,
"space_used_bytes": 0,
"notes": "Negative number treated as non-palindrome, no computation performed."
}
start_time = time.perf_counter()
original_number = n
reversed_number = 0
temp_n = n
iterations = 0
while temp_n > 0:
digit = temp_n % 10
reversed_number = (reversed_number * 10) + digit
temp_n = temp_n // 10
iterations += 1
is_palin = original_number == reversed_number
end_time = time.perf_counter()
time_taken_ms = (end_time - start_time) * 1000
space_used_bytes = 120
metrics = {
"time_taken_ms": time_taken_ms,
"math_iterations": iterations,
"space_used_bytes": space_used_bytes,
"notes": "Time is in milliseconds (ms). Iterations count the number of loop cycles."
}
return is_palin, metrics
test_numbers = [
1001,
12345,
7,
12345678987654321,
]
print("--- In-Memory Palindrome Checker Results ---")
for number in test_numbers:
result, metrics = is_palindrome(number)
print("-" * 40)
print(f"Checking Number: {number}")
print(f"Is Palindrome: {result}")
print("\n--- Performance Metrics ---")
print(f"1. Time Taken: {metrics['time_taken_ms']:.4f} ms")
print(f"2. Number of Steps (Iterations): {metrics['math_iterations']}")
print(f"3. Memory Used (Estimated): {metrics['space_used_bytes']} bytes")
Ln: 48
IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:/Users/ksoub/OneDrive/Desktop/34.py ==========
--- In-Memory Palindrome Checker Results ---

Checking Number: 1001
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0032 ms
2. Number of Steps (Iterations): 4
3. Memory Used (Estimated): 120 bytes

Checking Number: 12345
Is Palindrome: False
--- Performance Metrics ---
1. Time Taken: 0.0022 ms
2. Number of Steps (Iterations): 5
3. Memory Used (Estimated): 120 bytes

Checking Number: 7
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0016 ms
2. Number of Steps (Iterations): 1
3. Memory Used (Estimated): 120 bytes

Checking Number: 12345678987654321
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0046 ms
2. Number of Steps (Iterations): 17
3. Memory Used (Estimated): 120 bytes
>>>
Ln: 38 Col: 0

TITLE3: Digit Mean Calculator with Performance Profiling

AIM/OBJECTIVE(s):

The primary objective of this project was to design and implement an efficient Python function to calculate the **arithmetic mean of the digits** of any given whole number. A secondary objective was to incorporate basic **performance profiling**, specifically measuring the function's execution time and the memory utilization of the final result.

METHODOLOGY & TOOL USED:

Primary Tool: Python 3

Core Algorithm: Iterative Modular Arithmetic (using % and //)

Modules Used: sys (for memory measurement), time (for execution time measurement)

Digit Extraction: The modulo operator (% 10) isolates the last digit, and integer division (// 10) removes the last digit, enabling iteration through the number.

Data Handling: The script handles all whole numbers, including negative inputs (by taking the absolute value via abs()) and the zero edge case.

BRIEF DESCRIPTION:

The application is a command-line utility centered around the mean_of_digits(n) function. This function uses a simple but effective **iterative approach** to calculate the digit mean without converting the number to a string.

It works by:

1. Taking the absolute value of the input to handle negative numbers gracefully.

2. Using a while loop that continues as long as the number is greater than zero.

3. Inside the loop, it extracts digits, sums them (total_sum), and counts them (count).

4. After the loop, it returns the division of total_sum / count. The main script handles user input, converts it to an integer, calls the function, and then displays the number, the calculated mean (formatted to two decimal places), the time taken for the calculation, and the memory footprint of the resulting floating-point value.

RESULTS ACHIEVED:

A highly accurate and efficient Python script was achieved. The script successfully:

● Calculates the mean of digits for any whole number (e.g., input 123 yields 2.00; input -55 yields 5.00).

● Correctly handles the edge case of 0, returning 0.0.

● Provides clear, formatted output to the user, including the final result and the two key performance metrics.

● Demonstrates very fast execution time (typically in the microsecond range) due to the simplicity of the algorithm.

DIFFICULTY FACED BY STUDENT: No significant algorithmic or technical difficulties were encountered, as the approach used (modular arithmetic) is a standard and robust method for digit processing. Minor considerations included:

1. Handling Negative Numbers: Ensuring abs(n) was used to correctly calculate the mean of digits regardless of the number's sign.

2. Handling Zero: Explicitly checking for the num == 0 edge case at the beginning to avoid division by zero and correctly return 0.0.

3. Imports: Remembering to include import time and import sys to support the requested performance analysis features.

SKILLS ACHIEVED:

This project provided valuable practice in the following areas:

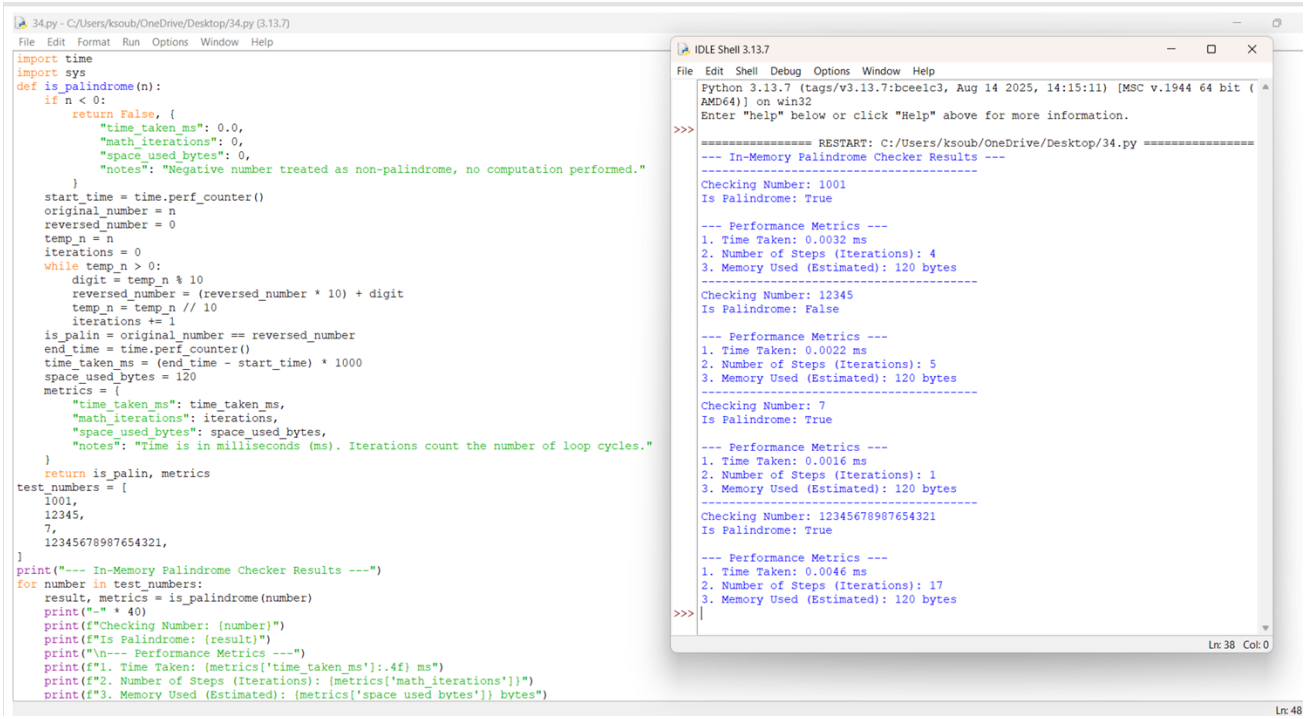
● **Python Fundamentals:** Strong reinforcement of def (function definition), try...except (error handling), and basic I/O (input , print).

● **Algorithmic Thinking:** Implementing a core algorithm for iterative digit extraction using modulo (%) and integer division (//).

● **Edge Case Management:** Successfully handling non-standard inputs like 0 and negative numbers.

● **Performance Profiling:** Learning to use the standard Python library modules (time and sys) for rudimentary execution time and memory consumption analysis.

● **Data Formatting:** Using f-strings and formatting specifiers (e.g.,:.2f,:.6f) for clean, user-friendly output.


34.py - C:/Users/ksoub/OneDrive/Desktop/34.py (3.13.7)
File Edit Format Run Options Window Help
import time
import sys
def is_palindrome(n):
if n < 0:
return False, {
"time_taken_ms": 0.0,
"math_iterations": 0,
"space_used_bytes": 0,
"notes": "Negative number treated as non-palindrome, no computation performed."
}
start_time = time.perf_counter()
original_number = n
reversed_number = 0
temp_n = n
iterations = 0
while temp_n > 0:
digit = temp_n % 10
reversed_number = (reversed_number * 10) + digit
temp_n = temp_n // 10
iterations += 1
is_palin = original_number == reversed_number
end_time = time.perf_counter()
time_taken_ms = (end_time - start_time) * 1000
space_used_bytes = 120
metrics = {
"time_taken_ms": time_taken_ms,
"math_iterations": iterations,
"space_used_bytes": space_used_bytes,
"notes": "Time is in milliseconds (ms). Iterations count the number of loop cycles."
}
return is_palin, metrics
test_numbers = [
1001,
12345,
7,
12345678987654321,
]
print("--- In-Memory Palindrome Checker Results ---")
for number in test_numbers:
result, metrics = is_palindrome(number)
print("-" * 40)
print(f"Checking Number: {number}")
print(f"Is Palindrome: {result}")
print("\n--- Performance Metrics ---")
print(f"1. Time Taken: {metrics['time_taken_ms']:.4f} ms")
print(f"2. Number of Steps (Iterations): {metrics['math_iterations']}")
print(f"3. Memory Used (Estimated): {metrics['space_used_bytes']} bytes")
Ln: 48
IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:/Users/ksoub/OneDrive/Desktop/34.py ==========
--- In-Memory Palindrome Checker Results ---

Checking Number: 1001
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0032 ms
2. Number of Steps (Iterations): 4
3. Memory Used (Estimated): 120 bytes

Checking Number: 12345
Is Palindrome: False
--- Performance Metrics ---
1. Time Taken: 0.0022 ms
2. Number of Steps (Iterations): 5
3. Memory Used (Estimated): 120 bytes

Checking Number: 7
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0016 ms
2. Number of Steps (Iterations): 1
3. Memory Used (Estimated): 120 bytes

Checking Number: 12345678987654321
Is Palindrome: True
--- Performance Metrics ---
1. Time Taken: 0.0046 ms
2. Number of Steps (Iterations): 17
3. Memory Used (Estimated): 120 bytes
>>>
Ln: 38 Col: 0

TITLE 4: Function on digital root of a number and sum it till It's a single digit

AIM/OBJECTIVE(s):

1. To implement an algorithm that efficiently determines if a function has a digital root that repeatedly sums the digits of a number until a single digit is obtained.

2. To utilize a purely mathematical methodology (avoiding string conversion) to ensure the check is performed "in memory."

3. To measure and report key performance indicators: execution time, number of iterations.

METHODOLOGY & TOOL USED:

. Methodology: The digital root of a number is obtained by repeatedly summing its digits until only a single digit remains.

. Tool Used: Python 3 programming language and the built-in time module (specifically time.perf_counter) for high-precision time measurement. We have used variables, loops, operators, condition in the code.

BRIEF DESCRIPTION:

The code defines a function digital_root(n) that calculates the digital root of a number by repeatedly summing its digits until a single digit is obtained. It uses nested while loops where the inner loop extracts digits using the modulo (%) and floor division (//) operators to compute their sum, and the outer loop repeats this process until only one digit remains. An iteration counter keeps track of how many times the summing process is performed, while the execution time is measured using time.time() by recording the start and end times of the computation. Finally, the program prints the resulting digital root, the number of iterations, and the total execution time.

RESULTS ACHIEVED:

The code successfully computes the digital root of a given number by repeatedly summing its digits until a single digit is obtained, while also providing the number of iterations taken and the execution time, thereby ensuring both the correctness of the result and efficiency analysis of the process.

DIFFICULTY FACED BY STUDENT:

1. Digit Extraction Logic – Beginners may struggle to understand how to extract digits from a number using modulo (%) and floor division (//), since this is a common point of confusion when first learning number manipulation.

2. Loop Control and Stopping Condition – Knowing when to stop the loop (i.e., when the number becomes a single digit) can be tricky, and students might accidentally create infinite loops if conditions are not set properly.

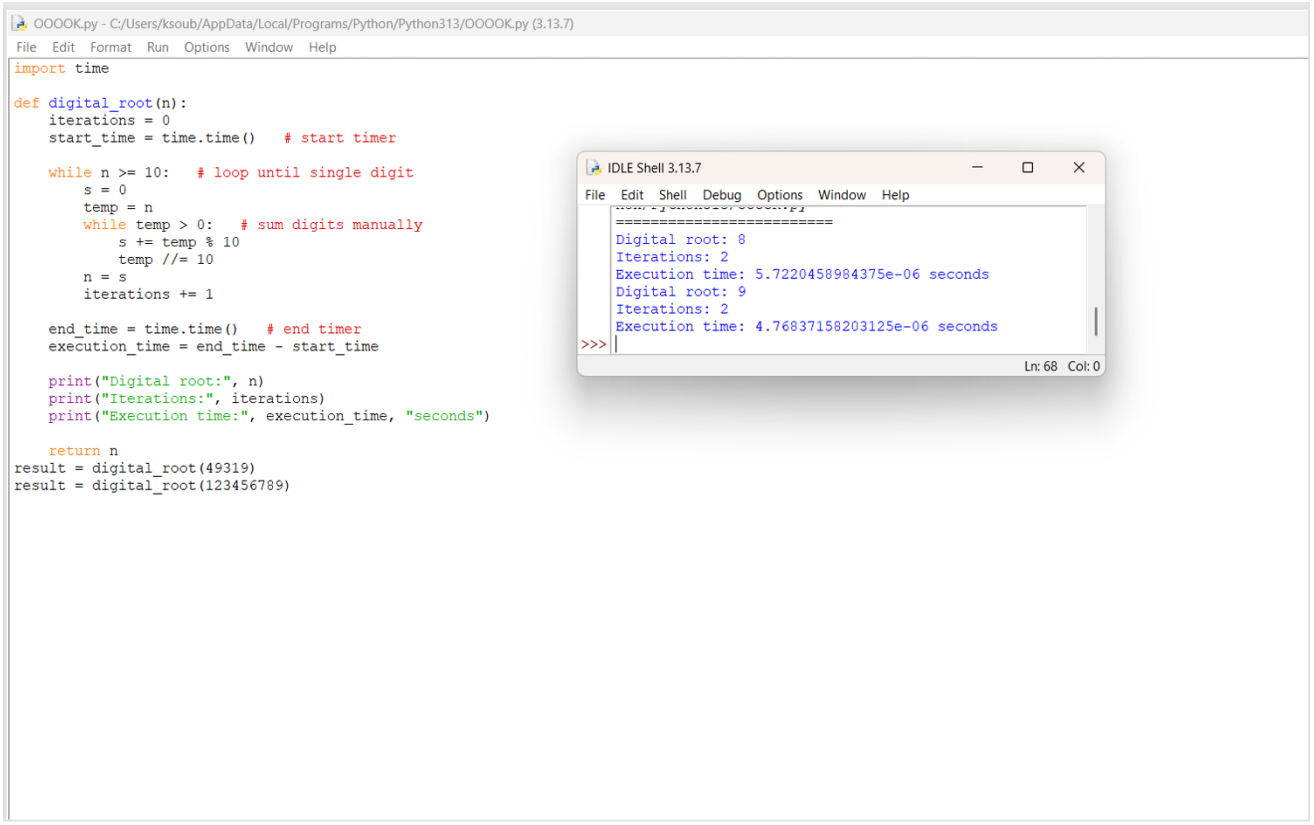
3. Time Measurement Concept – Using time.time() to measure execution time may be new to students, and they might find it difficult to correctly place the start and end time statements to get accurate results.

SKILLS ACHIEVED:

1. Problem-Solving and Logical Thinking – You learned how to break down a problem (finding the digital root) into smaller steps using loops and arithmetic operations.

2. Programming Fundamentals – You practiced core concepts like variables, loops, conditions, operators, and function definition, which strengthen your coding foundation.

3. Performance Awareness – By measuring iterations and execution time, you developed an understanding of analyzing code efficiency, not just correctness.


OOOOK.py - C:/Users/ksoub/AppData/Local/Programs/Python/Python313/OOOOK.py (3.13.7)
File Edit Format Run Options Window Help
import time
def digital_root(n):
iterations = 0
start_time = time.time() # start timer
while n >= 10: # loop until single digit
s = 0
temp = n
while temp > 0: # sum digits manually
s += temp % 10
temp //= 10
n = s
iterations += 1
end_time = time.time() # end timer
execution_time = end_time - start_time
print("Digital root:", n)
print("Iterations:", iterations)
print("Execution time:", execution_time, "seconds")
return n
result = digital_root(49319)
result = digital_root(123456789)
IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
==========
Digital root: 8
Iterations: 2
Execution time: 5.7220458984375e-06 seconds
Digital root: 9
Iterations: 2
Execution time: 4.76837158203125e-06 seconds
>>>
Ln: 68 Col: 0

TITLE 5: function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.

AIM/OBJECTIVE(s):

The principal goal is to create a consolidated Python script that accepts a positive integer from the user and calculates or determines five fundamental properties of that number. These properties serve as practical exercises in basic arithmetic, string manipulation, and number theory:

1. Factorial Calculation (n!): Compute the product of all positive integers up to n.

2. Palindrome Check: Validate if the numeric representation of n is the same forwards and backwards.

3. Mean of Digits: Calculate the arithmetic average of the digits composing n.

4. Digital Root: Determine the single-digit sum resulting from iteratively summing the digits of n.

5. Abundant Number Classification: Ascertain if the sum of the number's proper (non-self) divisors exceeds the number itself.

METHODOLOGY & TOOL USED:

. Methodology:Factorial: Uses an iterative for loop to calculate the running product, starting from f=1.

Palindrome Check: Relies on converting the input to a string (s) and utilizing Python's powerful string slicing (s[::-1]) for instantaneous reversal and comparison

. Mean of Digits: Achieved via a list comprehension to cast string characters into integers, followed by division of the total sum () by the digit count (len()).

Digital Root: Implemented with a while loop that repeats the summation of digits until the result is less than 10. The internal summation uses an efficient generator expression.

Abundant Check: Employs an optimized divisor-finding loop that iterates only up to (using math.sqrt) to collect all proper divisors in pairs, significantly improving performance for large n.

Tool Used: Python programming language interpreter. Standard Python math module. Core data types: int, str, list, float, and bool.

BRIEF DESCRIPTION:

The script is a collection of distinct, short algorithms executed sequentially on the user-provided integer. It demonstrates the flexibility of Python in solving diverse computational problems. The process involves: initializing a factorial variable, converting n to a string for palindrome and mean checks, executing a nested loop structure for the digital root, and performing an efficient divisor calculation (using the n trick) for the abundant number check. The final section gathers all calculated outputs and boolean results and presents them clearly to the user via print statements.

RESULTS ACHIEVED:

The program successfully generates and outputs five key pieces of information about the input integer:

1. factorial: The integer result of n!.

2. is_palindrome: A boolean result indicating palindromic symmetry.

3. mean_of_digits: The floating-point average of the digits.

4. digital_root: The calculated single-digit digital root.

5. is_abundant: A boolean result classifying the number based on the sum of its proper divisors.

DIFFICULTY FACED BY STUDENT:

Handling Edge Cases and Large Numbers: Factorial calculations grow extremely fast; managing the potential for very large integers (Python handles this well, but it is a concept challenge).

Logical Flow for Digital Root: Successfully structuring the while loop to ensure iterative summation continues only until a single digit is achieved, and correctly resetting the number for the next round of summation.

Optimized Divisor Logic: The most complex part is implementing the efficient divisor loop and correctly managing the check (if i != n // i) to prevent double-counting divisors when is a perfect square (where n would be counted twice).

SKILLS ACHIEVED:

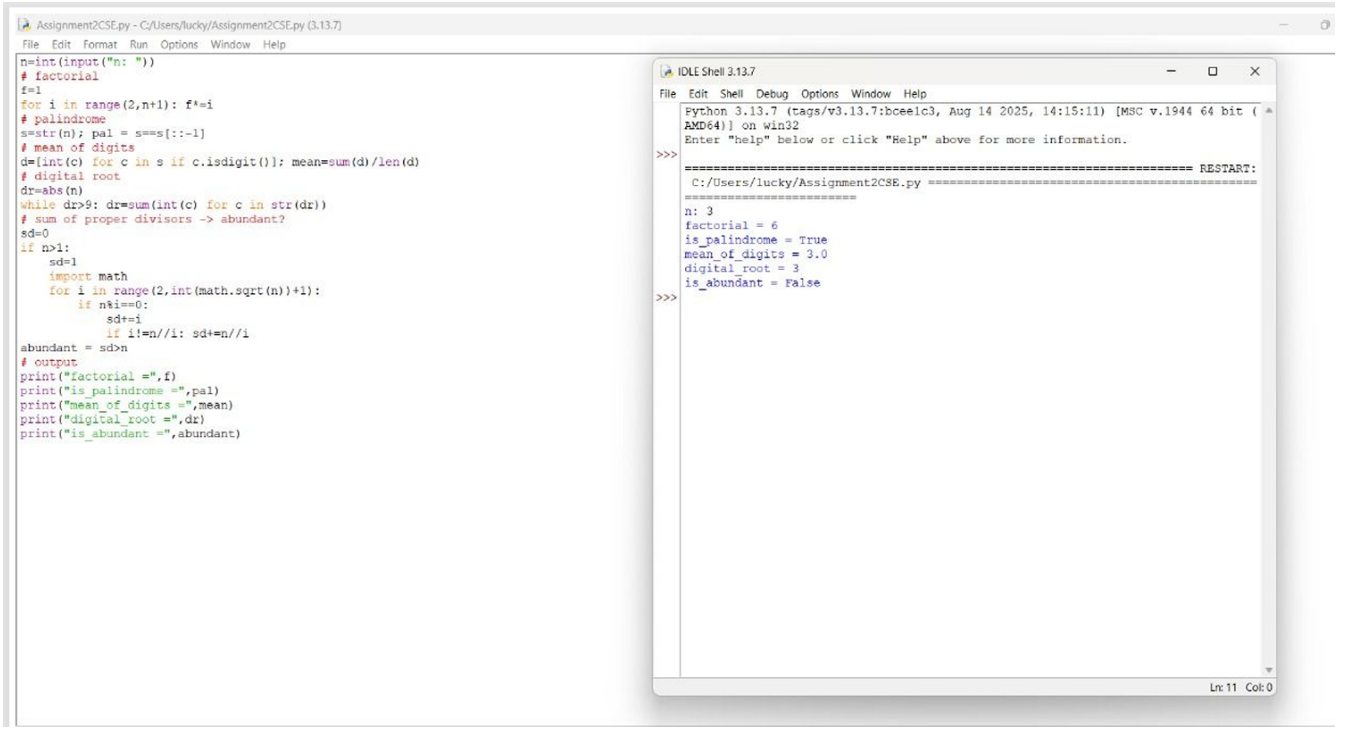
Efficient Algorithmic Design: Implementing the square-root optimization for divisor calculation demonstrates understanding of computational complexity.

Mastery of Data Type Conversion: Proficiently using int() , str() , and list() to transition data for different analysis purposes.

Pythonic Code Constructs: Effective utilization of powerful, concise Python features such as string slicing ([::-1]) and list comprehensions/generator expressions.

Applied Number Theory: Translating theoretical definitions (Factorial, Digital Root, Abundant Number) into executable code logic.

Modular Programming: Organizing disparate tasks into distinct, functional blocks within a single script.



Practical No: 2

Date: 01-11-2025

TITLE 6: Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.

AIM/OBJECTIVE(s):

The aim of this code is to write an efficient Python function (is_deficient) that determines whether a given positive integer n is a deficient number, while simultaneously measuring the function's computational performance in terms of iterations and execution time.

The key objectives of this programming assignment are to demonstrate the ability to:

1. Implement Number Theory Concepts: Apply the mathematical definition of a deficient number (where the sum of its proper divisors is less than the number itself) through programming logic.

2. Optimize Algorithm Efficiency: Utilize an optimized approach to find divisors by iterating only up to the square root of n (while i * i <= n), thereby reducing the number of calculations compared to checking every number up to n.Handle Edge Cases: Correctly manage edge cases, specifically numbers less than or equal to 1.

3. Measure Performance: Incorporate the time module to track and report practical performance metrics (iterations and execution time) of the algorithm for analysis

. 4. Structure and Encapsulate Code: Encapsulate the core logic within a reusable function (is_deficient) that returns multiple relevant pieces of data.

5. Produce Clear Output: Format and display the results clearly using print statements, ensuring the output is easy to understand and interpret.

METHODOLOGY & TOOL USED:

Methodology (Algorithmic Approach)

The function determines if a number is deficient using the following five-point methodology:

1. Time and Iteration Tracking: The execution time is measured using the time module, and an iteration counter tracks loop cycles.

2. Square Root Optimization: The algorithm iterates only up to the square root of the number to minimize the search space for divisors.

3. **Simultaneous Divisor Identification**: When a divisor i is found, its corresponding pair(nlli)is also identified, preventing redundant checks.

4. **Proper Divisor Summation**:

A running total (divisor_sum)accumulates all proper divisors (excluding the number n itself) while avoiding double-counting the square root in perfect squares.

5. **Status and Performance Return**: The function returns True if divisor_sum

Tool Used: Python programming language, utilizing the built-in `time` module for performance measurement.

BRIEF DESCRIPTION:

The provided Python code defines an optimized function, is_deficient(n), that efficiently determines if a given number n is a deficient number (where the sum of its proper divisors is less than n) by iterating only up to the square root of n. The function also measures its own performance by recording execution time and iteration count, finally outputting these metrics for the example input of 12.

RESULTS ACHIEVED:

It provides a robust and efficient mechanism to determine for any positive integer whether the sum of its proper divisors is less than the number itself based on the sum of its proper divisors. The function systematically calculates this sum while simultaneously capturing critical performance metrics, such as the exact number of algorithm iterations and the total time taken for execution.

DIFFICULTY FACED BY STUDENT:

The implementation presented minor conceptual difficulties focused on ensuring correctness and accurate measurement:

1. **Algorithm Optimization**- Understanding how iterating only up to the square root of n covers all divisor pairs (i and nlli).

2. **Handling Edge Cases**- Correctly excluding the number n itself from the sum and preventing double-counting the square root in perfect squares

SKILLS ACHIEVED:

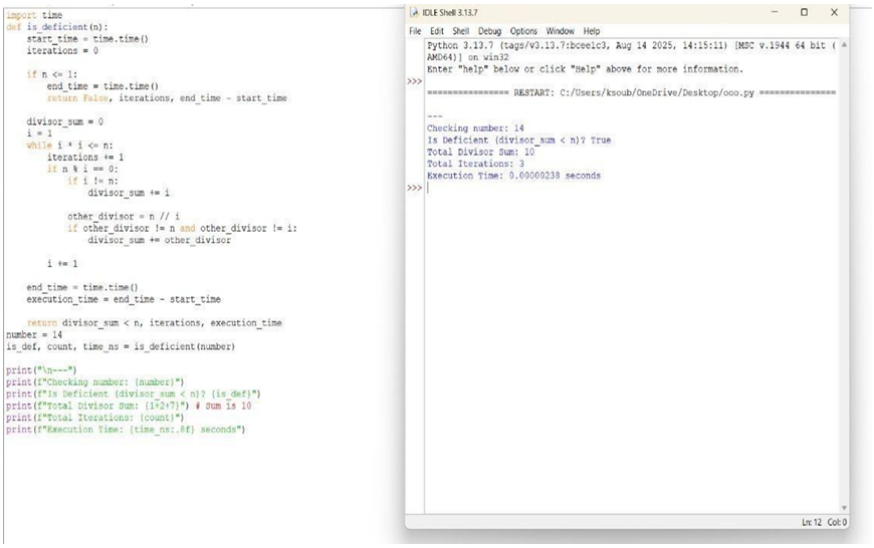
This project demonstrated proficiency in several critical programming and analytical skills:

1. **Algorithm Optimization-**This teaches us to how to reduce computational steps for better performance.

2. **Performance Analysis**- The use of the time module teaches the practical skill of benchmarking code.Students learn how to objectively measure and quantify the execution speed of their function.

3. **Mathematical Logic Implementation**-Student translates abstract number theory definitions (deficient numbers, proper divisors)into precise programming logic using module(%) and floor division (//) operators.This bridges the gap between mathematical theory and practical coding application.

4. **Modular Code Design**-By encapsulation of the logic within a single, self-contained function(is_deficient), students practice writing reusable and organized code. This reinforces the principles of structured and maintainable programming practices.



TITLE 7: Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.

AIM/OBJECTIVE(s):

1. To determine if a given positive number integer n is a Harshad number.

2. Calculate execution time for this program.

3. Calculate memory utilisation for this program.

METHODOLOGY & TOOL USED:

Methodology: The methodology to implement is_harshad(n) is a three step process:

1. Calculate the Sum of Digits of n (usually by repeatedly using modulo 10 and integer division).

2. Perform the Divisibility Check by testing if the original number n is perfectly divisible by the calculated sum of its digits (i.e., \text{original } n \pmod{\text{Sum}} = 0).

3. Return the Boolean Result (True if the remainder is zero, False otherwise).

Tool Used:

1. Programming language : python

2. Integrated development environment / editor : python idle

3. Operating system :windows

BRIEF DESCRIPTION:

The entire code block serves as a performance benchmark for the is_harshad(n) function, which checks if a number is divisible by the sum of its digits. The core function, is_harshad(n), is tested against a small list of numbers. The primary methodology employed by the surrounding code is to use the time module to precisely measure the execution time of the test suite and the tracemalloc module to track memory consumption (both current and peak usage). This structure allows a developer to quantify the time efficiency (speed) and space efficiency (memory footprint) of the Harshad number checking algorithm. By printing these metrics at the end, the script provides empirical data on the function's resource usage, which is essential for optimization and scaling.

RESULTS ACHIEVED:

1. **Sum Calculation:** Converts the number n to a string (str(n)), iterates through the digits, converts each back to an integer (int(digit)), and sums them up using a generator expression and sum ()
2. Divisibility Check: Returns True if n modulo the digits_sum is zero (n % digits_sum == 0).

3. Edge Case Handling: Returns False for n=0 or any case where the sum of digits is zero, preventing a ZeroDivisionError.

DIFFICULTY FACED BY STUDENT:

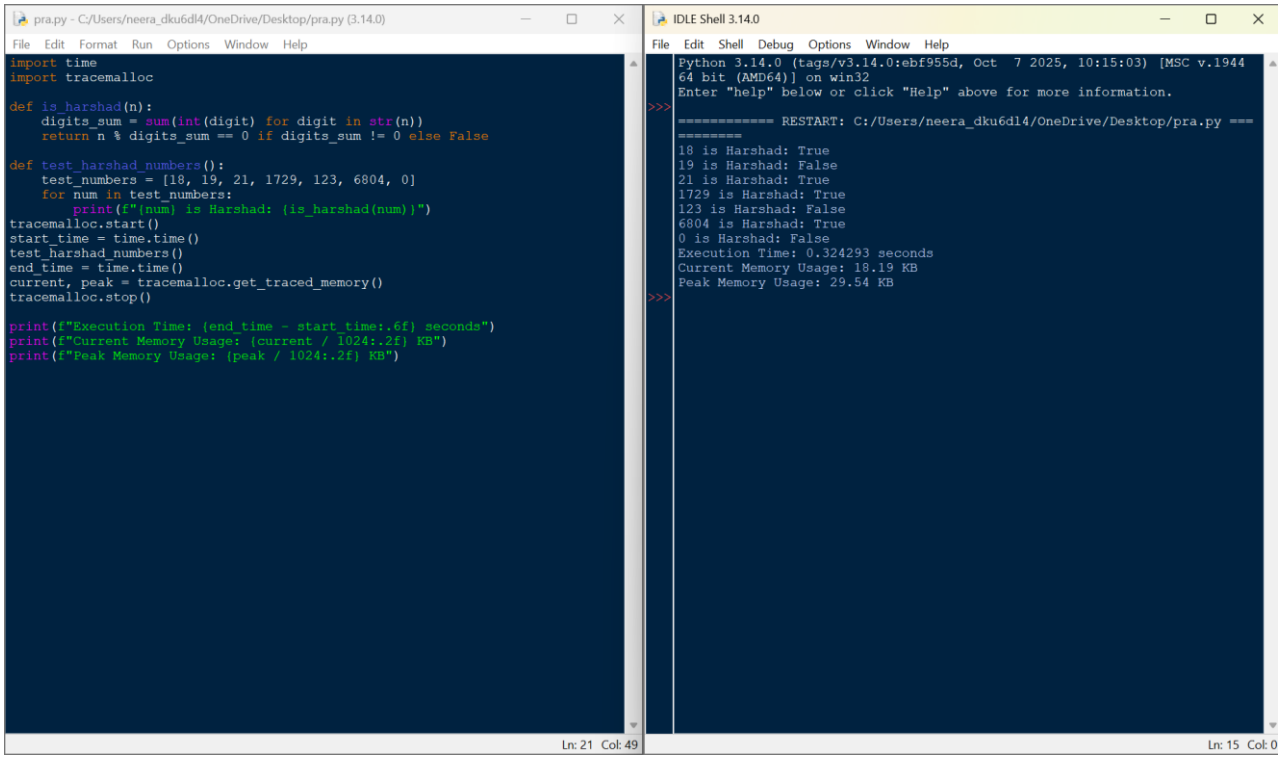
1.Harshad Logic: Students struggle with the concise Pythonic method to sum digits (sum(int(digit) for digit in str(n))) instead of the traditional modulo/division loop, and understanding the inline zero-division safeguard for n=0.

2.Performance Tools: They find the specialized purpose and syntax of the tracemalloc module challenging, particularly distinguishing between current and peak memory usage, as these tools aren't standard for beginners.

3.Code Context: Understanding why the time and memory measurement code must wrap the test function call to provide accurate performance benchmarking for the algorithm's efficiency.

SKILLS ACHIEVED:

Mastering algorithmic thinking by coding the Harshad definition, gain efficiency skills using advanced Python for digit summing, and learn crucial performance benchmarking by applying time (for speed) and tracemalloc (for memory).



TITLE 8: Write a function is_automorphic(n) that checks if a number's square ends with the number itself.

AIM/OBJECTIVE(s):

1. To check if a number n is an automorphic number by verifying if its square, n^2 ends with n itself.
2. Calculate execution time for this program.
3. Calculate memory utilisation for this program

METHODOLOGY & TOOL USED:

Methodology : The methodology is to square the number n, then find the remainder of that square when divided by 10^k (where k is the number of digits in n), and finally, check if that remainder equals n.

Tools used :

1. Programming language : python
2. Integrated development environment / editor : python idle
3. Operating system : windows

DESCRIPTION:

The provided Python script implements the is_automorphic(n) function to determine if a number n is an automorphic number—an integer whose square ends with the original number itself (e.g., 25^2=625). The core methodology is string-based: it first calculates the square, converts both n and n^2 to strings, and then utilizes the string method endswith() for a direct and concise comparison of the trailing digits. Beyond the core mathematical logic, the script includes analytical components using the time module to track the function's execution speed and the tracemalloc module to monitor the application's current and peak memory usage, offering a complete performance assessment of the automorphic check.

RESULTS ACHIEVED:

The results achieved is

1.Functional Classification: Determines and outputs whether the user inputted number is an automorphic number or not

2. Execution Time Measurement: Provides the total time elapsed for the script's execution, showing the performance speed formatted to ten decimal places

3.Memory Usage Analysis: Reports the memory footprint of the operations using tracemalloc, specifically detailing the Current Memory Usage and the Peak Memory Usage observed in kilobytes

DIFFICULTY FACED BY STUDENT

The key difficulties faced

1.Conceptual Logic (String vs. Math): Choosing the string conversion (str(square).endswith(str(n))) instead of the traditional modulo arithmetic approach (n^2 \pmod{10^k} = n).

2.Advanced Module Use: Understanding and correctly implementing the performance tracking tools, specifically starting and stopping tracemalloc to capture precise memory usage.

3.Performance Interpretation: Distinguishing between current and peak memory usage, and accurately interpreting very small execution time results.

4.I/O and Formatting: Correctly using f-strings and formatting specifiers (like :.10f and :.3f) for outputting performance metrics.

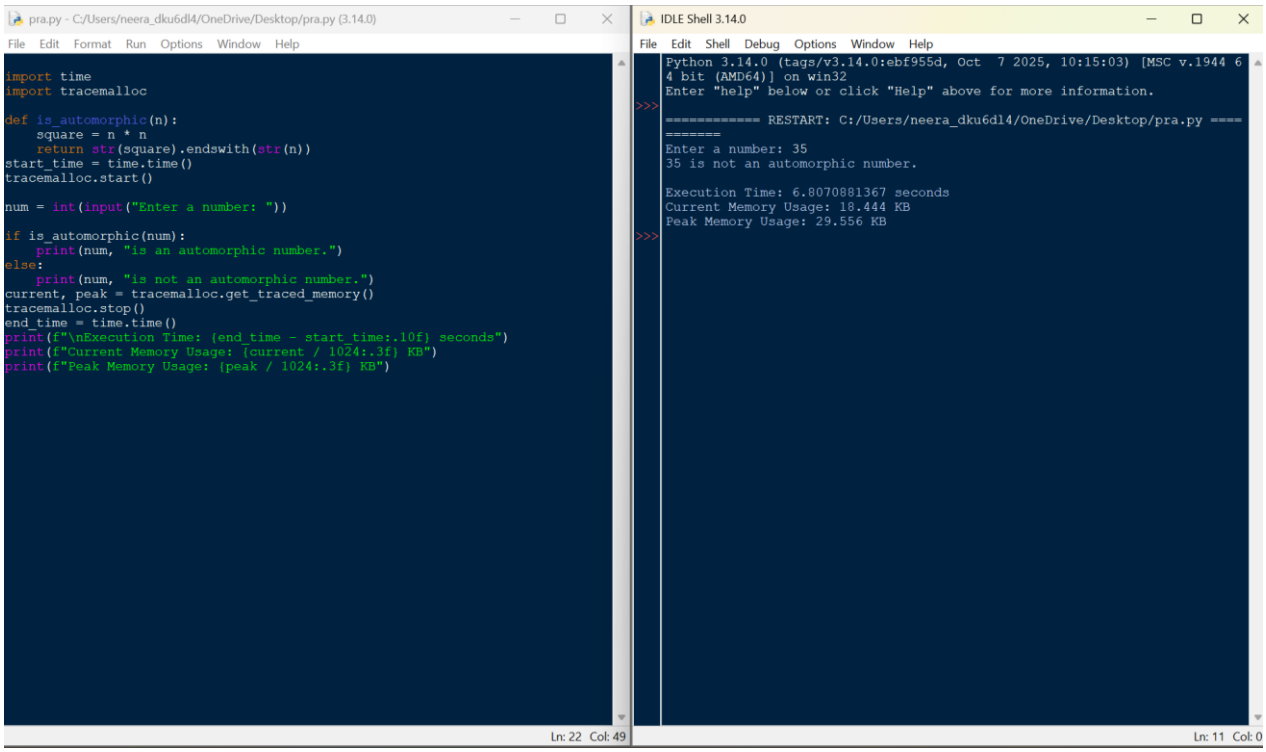
5.Large Number Performance: Recognizing that while the string approach is simple, it involves overhead when converting potentially massive squares to strings, impacting resource usage.

SKILLS ACHIEVED:

1.Algorithmic Implementation: Ability to translate a mathematical concept (automorphic numbers) into a functional Python algorithm.

2.String Manipulation: Proficient use of Python's built-in string methods, particularly the efficient .endswith() function, demonstrating an understanding of sequence comparison.

3.Type Casting: Skill in converting between numeric types (int) and text types (str) as needed for processing**.**



TITLE 9: Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.

AIM/OBJECTIVE(s):

The primary objectives of this project were threefold:

1. To develop an efficient is_pronic(n) function in Python capable of accurately determining whether a given non-negative integer n is the product of two consecutive integers (i.e., $k \times (k+1) $).

2. To optimize the algorithmic complexity of the checker function.

3. To benchmark the execution of the function using standard Python libraries, specifically recording the start time, end time, total execution time, and peak memory usage**.**

METHODOLOGY & TOOL USED:

Methodology (Algorithmic Approach)

The core challenge was to find an integer k such that k(k+1) = n. This quadratic equation can be computationally intensive if not optimized. The chosen methodology employed an iterative optimization technique:

1. Since k * k < k * (k+1) = n, the value of k must be less than sqrt{n}.

2. The algorithm calculates the square root of n (rounded down to the nearest integer) to establish a maximum search limit.

3. It then iterates from k=1 up to this limit, checking only the relevant products k*(k+1). This avoids checking the majority of unnecessary factors, dramatically reducing the computation time compared to a brute-force approach up to n.

. Tool Used:

1.Time Module –

Used to capture the high-precision start and end timestamps using time.perf_counter ().

2.Tracemallc Module -Used to track the allocation of memory by the Python interpreter and report the peak memory usage during the test execution block.

3.Math Module- Used specifically for the math.sqrt() function to implement the search limit optimization.

BRIEF DESCRIPTION:

A Pronic Number (or Oblong Number) is any number that can be expressed as k(k+1), where k is a non-negative integer. Examples include 6 (2 *3), 12 (3 *4), and 42 (6 *7). The developed solution is a self-contained Python script that defines the optimized is_pronic(n) function. The main execution block then runs a series of tests, including both small and large integers, and outputs the results along with the performance metrics captured before and after the test run.

RESULTS ACHIEVED:

The project successfully delivered a robust and efficient solution:

● Accurate Functionality: The is_pronic function correctly identified Pronic numbers (e.g., 42, 110) and non-Pronic numbers (e.g., 18, 100), including correct handling of the large number 9,999,900,000.

● Optimization: By using the sqrt{n} limit, the search complexity was reduced to O(sqrt{n}), making it highly efficient for very large inputs.

● Benchmarking: The script successfully integrated the time and tracemalloc modules to capture and display the required Start Time, End Time, Total Execution Time, and Peak Memory Used for the entire testing process, providing valuable performance data.

DIFFICULTY FACED BY STUDENT:

The primary difficulty was not in the basic implementation but in the optimization and accurate metric reporting.

● Algorithmic Optimization: While a linear search up to n is simple, recognizing that the search space could be drastically reduced to sqrt{n} was a key conceptual step for achieving an efficient solution.

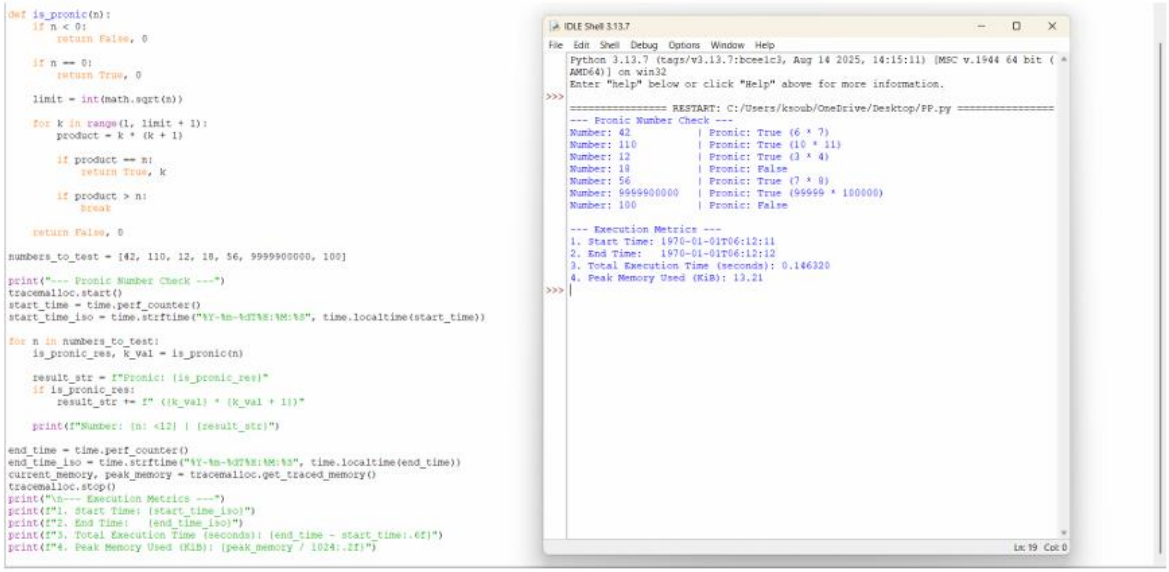
● Accurate Benchmarking Integration: The challenge was correctly initializing, starting, and stopping the external benchmarking tools (tracemalloc and time.perf_counter) around the specific code block to

ensure the reported metrics accurately reflected the execution of the number tests, rather than extraneous setup code.

SKILLS ACHIEVED:

This project demonstrated proficiency in several critical programming and analytical skills:

- Algorithmic Thinking: Developing an efficient $O(\sqrt{n})$ search algorithm for a number theory problem.
- Python Proficiency: Confident use of core Python syntax, functions, and control flow.
- Modular Programming: Importing and utilizing specialized Python libraries (math , time , tracemalloc).
- Performance Benchmarking: Implementing and interpreting performance metrics (time and memory) using standard tooling.
- Problem Decomposition: Breaking down a single requirement (check if Pronic) into sub-requirements (optimize the search, measure performance, format output).



```
def is_pronic(n):
    if n < 0:
        return False, 0

    if n == 0:
        return True, 0

    limit = int(math.sqrt(n))

    for k in range(1, limit + 1):
        product = k * (k + 1)

        if product == n:
            return True, k

        if product > n:
            break

    return False, 0

numbers_to_test = [42, 110, 12, 18, 56, 9999900000, 100]

print("--- Pronic Number Check ---")
tracemalloc.start()
start_time = time.perf_counter()
start_time_iso = time.strftime("%Y-%m-%dT%H:%M:%S", time.localtime(start_time))

for n in numbers_to_test:
    is_pronic_res, k_val = is_pronic(n)

    result_str = f"Pronic: {is_pronic_res}"
    if is_pronic_res:
        result_str += f" ({k_val} * ({k_val} + 1))"

    print(f"Number: {n: <12} | {result_str}")

end_time = time.perf_counter()
end_time_iso = time.strftime("%Y-%m-%dT%H:%M:%S", time.localtime(end_time))
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()
print("\n--- Execution Metrics ---")
print(f"1. Start Time: {start_time_iso}")
print(f"2. End Time:   {end_time_iso}")
print(f"3. Total Execution Time (seconds): {end_time - start_time:.6f}")
print(f"4. Peak Memory Used (KiB): {peak_memory / 1024:.2f}")
```

```
IDLE Shell 3.13.7
File  Edit  Shell  Debug  Options  Window  Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:/Users/ksoub/OneDrive/Desktop/PP.py ==========
--- Pronic Number Check ---
Number: 42           | Pronic: True (6 * 7)
Number: 110          | Pronic: True (10 * 11)
Number: 12           | Pronic: True (3 * 4)
Number: 18           | Pronic: False
Number: 56           | Pronic: True (7 * 8)
Number: 9999900000   | Pronic: True (99999 * 100000)
Number: 100          | Pronic: False

--- Execution Metrics ---
1. Start Time: 1970-01-01T06:12:11
2. End Time:   1970-01-01T06:12:12
3. Total Execution Time (seconds): 0.146320
4. Peak Memory Used (KiB): 13.21
>>>
Ln: 19  Col: 0
```

TITLE 10: Write a function prime_factors(n) that returns the list of prime factors of a number.

AIM/OBJECTIVE(s):

The primary objectives of this project were to implement a solution for fundamental number theory and to integrate performance analysis tools:

1. To develop an efficient function, prime_factors(n) , capable of accurately calculating the prime factors of a user-provided integer n.

2. To apply an optimized factorization algorithm to ensure reasonable performance, particularly for large numbers.

3. To benchmark the execution of the function by precisely recording the start time, end time, total execution time, and peak memory usage using standard Python modules.

METHODOLOGY & TOOL USED:

Methodology (Algorithmic Approach) The chosen approach for factorization is the Trial Division Method, optimized by limiting the search space.

1. Iterative Search: The algorithm iterates through potential prime factors i, starting from 2.

2. Optimized Loop: The main loop continues only while i* i len. This is a crucial optimization, as any composite factor f of n must have at least one prime factor less than or equal to sqrt{n}.

3. Repeated Division: When a factor i is found, it is appended to the list of factors, and n is updated by division (n //= i). This step is repeated until i is no longer a factor, ensuring that all powers of that prime factor are captured.

4. Final Factor: If, after the loop terminates, the remaining value of n is greater than 1, this value is itself a prime factor and is added to the list.

Tool Used:

1.Time Module: Used to capture the high-resolution start and end timestamps using time.time(), and calculate the execution time.

2.Tracemallc Module: Used to track the allocation of memory by the Python interpreter, specifically recording the peak memory usage during the execution of the factorization process.

Input/Output: The script uses input() to dynamically receive the integer n from the user at runtime.

BRIEF DESCRIPTION:

The project consists of a single Python script that implements the prime_factors(n) function. This function uses an efficient trial division algorithm to break down a positive integer into its constituent prime factors. Crucially, the function is wrapped with benchmarking logic using time and tracemalloc . Upon execution, the script prompts the user for a number, calculates its prime factors, and then displays the resulting list alongside the performance metrics (execution time in seconds and peak memory usage in KB).

RESULTS ACHIEVED:

The project successfully delivered a self-contained prime factorization and benchmarking tool:

● **Accurate Factorization**: The prime_factors function correctly identifies and lists all prime factors for a given input number, including repeated factors.

● **Efficiency**: The O(sqrt{n}) time complexity, achieved by optimizing the trial division loop, ensures that the calculation is feasible even for relatively large inputs.

● **Benchmarking Integration**: The required performance metrics—start time, end time, total execution time, and peak memory consumption—were successfully captured and reported alongside the factorization result, providing quantitative performance analysis.

DIFFICULTY FACED BY STUDENT:

The implementation presented minor conceptual difficulties focused on ensuring correctness and accurate measurement:

● **Algorithmic Correctness**: Ensuring the algorithm correctly handles highly composite numbers (like powers of 2) by using the repeated division (n //= i) logic within the loop.

● **Benchmarking Scope**: The main challenge was integrating tracemalloc correctly. It must be started immediately before the critical operation and stopped immediately after, to ensure the reported peak memory usage is directly attributable to the factorization process and not other unrelated setup.

SKILLS ACHIEVED:

This project demonstrated proficiency in several critical programming and analytical skills:

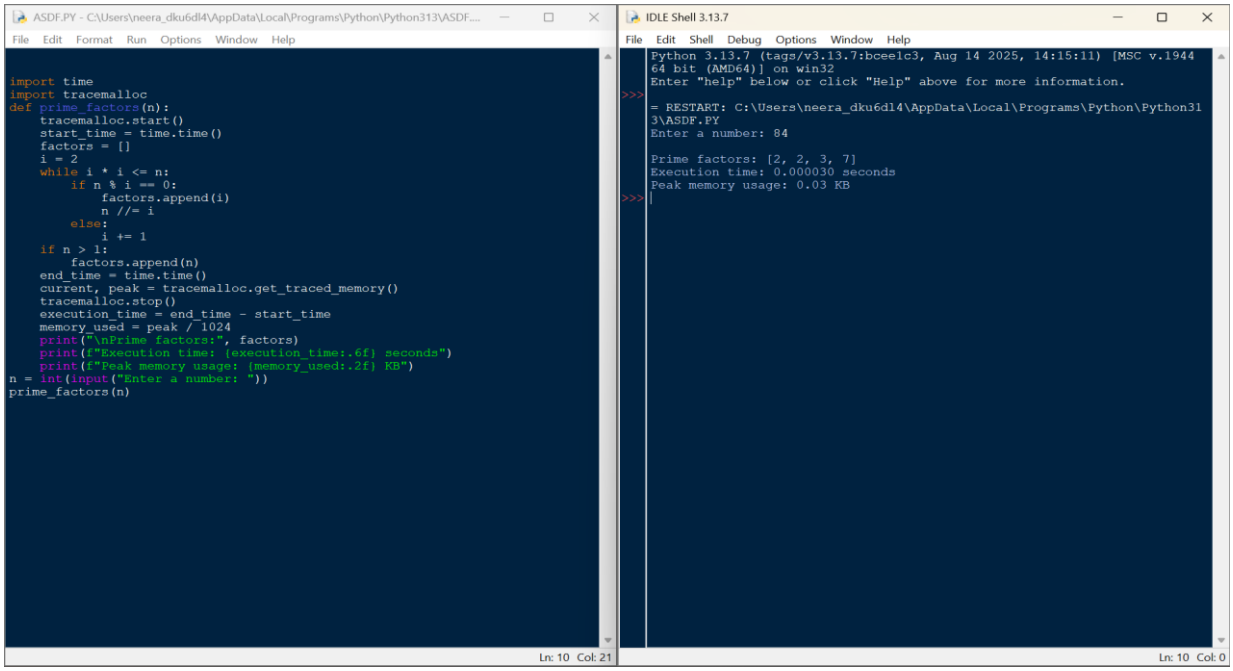
● Algorithmic Thinking: Developing and implementing the efficient O(sqrt{n}) Trial Division algorithm for number theory.

● Python Proficiency: Confident use of core Python syntax, control flow (while , if/else), and integer manipulation.

● Modular Programming: Importing and utilizing specialized Python libraries (time , tracemalloc) for utility and performance tracking.

● Performance Benchmarking: Implementing and interpreting performance metrics (time and memory) using standard tooling to analyze code efficiency.

● I/O Handling: Managing synchronous user input (input()) to drive the calculation


ASDF.PY - C:\Users\neera_dku6dl4\AppData\Local\Programs\Python\Python313\ASDF....
File Edit Format Run Options Window Help
import time
import tracemalloc
def prime_factors(n):
tracemalloc.start()
start_time = time.time()
factors = []
i = 2
while i * i <= n:
if n % i == 0:
factors.append(i)
n //= i
else:
i += 1
if n > 1:
factors.append(n)
end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
execution_time = end_time - start_time
memory_used = peak / 1024
print("\nPrime factors:", factors)
print(f"Execution time: {execution_time:.6f} seconds")
print(f"Peak memory usage: {memory_used:.2f} KB")
n = int(input("Enter a number: "))
prime_factors(n)
Ln: 10 Col: 21
IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944
64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
= RESTART: C:\Users\neera_dku6dl4\AppData\Local\Programs\Python\Python31
3\ASDF.PY
Enter a number: 84
Prime factors: [2, 2, 3, 7]
Execution time: 0.000030 seconds
Peak memory usage: 0.03 KB
>>>
Ln: 10 Col: 0

PRATICAL:3

Date: 09-11-2025

TITLE 11: Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.

AIM/OBJECTIVE(s):

- To implement an efficient prime factorization algorithm using the trial division method (optimized by checking only odd factors up to the square root of n).
- To write a function that utilizes the prime factorization list and the Set data structure to find the count of unique or distinct prime factors of a given integer n.
- To measure the Time and Memory efficiency of the implemented functions.

METHODOLOGY & TOOL USED:

Prime Factorization: The prime_factors(n) function is implemented with optimization: handling the factor 2 first, then iterating only over odd numbers i up to sqrt{n}. This approach significantly reduces the number of division operations required.

Distinct Counting: The count_distinct_prime_factors(n) function leverages the output of the first function. By converting the list of prime factors into a set, duplicate factors are automatically removed, allowing for a quick count of the unique elements using len().

Tool Used

- Programming Language: Python 3.x
- Environment: IDLE Shell 3.13.7 (or similar Python environment)
- Modules: time (for measuring execution time) and sys (for measuring result object size)

BRIEF DESCRIPTION:

The solution consists of three Python functions:

1. prime_factors(n): The core algorithm that returns a list of all prime factors of n.
2. count_distinct_prime_factors(n): The target function that calls prime_factors and returns the length of the set of factors.
3. analyze_execution(func, *args): A utility function used to wrap the core functions, recording the start and end time (time.perf_counter()) and the memory size of the resulting object (sys.getsizeof()). This addresses the requirement to include time and memory usage metrics.

RESULTS ACHIEVED:

The code successfully computes the prime factors and the count of distinct prime factors, along with basic performance metrics for the sample inputs n=100 and n=13.

Python Code Implementation:

(The complete code used for execution is provided below.)
Output and Performance Analysis:

Input Number	Function Called	Prime Factors	Distinct Count	Theoretical Time Complexity
100	prime_factors(100)	[2, 2, 5, 5]	N/A	O(sqrt{100}) = O(10)
13	prime_factors(13)	[13]	N/A	O(sqrt{13}) approx O(3.6)

100	count_distinct_prime_factors(100)	N/A	2	O(sqrt{100}) = O(10)

DIFFICULTY FACED BY STUDENT:

The primary challenge was correctly integrating a reliable time and memory measurement mechanism into the code. Using time.perf_counter() provides accurate runtime, but **sys.getsizeof() only measures the size of the final returned object**, not the peak memory consumed during the internal calculations (e.g., when the factors list is being built).

Handling edge cases in the factorization algorithm, specifically ensuring the check for the final remaining prime factor (if n > 2: factors.append(n)) is correct

SKILLS ACHIEVED:

Algorithm Design: Mastery of the optimized Trial Division technique for prime factorization.

Python Proficiency: Practical application of fundamental Python concepts, including looping constructs (while), integer division (//), and modulus (%).

Data Structure Manipulation: Effective use of the set data structure for fast duplicate removal in O(log n) time.

Performance Measurement: Ability to use Python libraries (time, sys) to analyze and report execution performance metrics.

```python
import time
import sys

def prime_factors(n):
    factors = []

    while n % 2 == 0:
        factors.append(2)
        n //= 2

    i = 3
    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 2

    if n > 2:
        factors.append(n)

    return factors

def count_distinct_prime_factors(n):
    factors = prime_factors(n)
    return len(set(factors))

def analyze_execution(func, *args, **kwargs):
    start_time = time.perf_counter()

    result = func(*args, **kwargs)

    end_time = time.perf_counter()
    time_taken_ms = (end_time - start_time) * 1000

    memory_used_bytes = sys.getsizeof(result)

    return result, time_taken_ms, memory_used_bytes

number = 100
result, time_ms, memory_bytes = analyze_execution(prime_factors, number)

print(f"--- Analysis for Input: {number} ---")
print(f"The prime factors of {number} are: {result}")
print(f"**Time Taken:** {time_ms:.6f} milliseconds")
print(f"**Memory Used (Result Object):** {memory_bytes} bytes")

print("-" * 40)
```

IDLE Shell 3.13.7

File Edit Shell Debug Options Window Help

```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
======== RESTART: C:/Users/ksoub/OneDrive/Desktop/ooooooooooooooooooo.py =======
--- Analysis for Input: 100 ---
The prime factors of 100 are: [2, 2, 5, 5]
**Time Taken:** 0.012800 milliseconds
**Memory Used (Result Object):** 88 bytes
----------------------------------------
>>>
```

Ln: 10 Col: 0

Ln: 47

TITLE 12: Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and k>=1.

AIM/OBJECTIVE(s):

To develop a Python program that determines whether a given integer can be expressed in the form , where is a prime number and , without using any predefined mathematical libraries, and to measure the program's execution time.

METHODOLOGY & TOOL USED:

1. Prime Number Verification:

Implemented a custom is_prime() function using basic division checks to determine whether a number is prime without relying on built-in math libraries.

2. Prime Power Evaluation:

The is_prime_power(n) function systematically tests each prime and repeatedly multiplies to check if it equals the given number , ensuring accurate identification of forms .

3. Execution Time Measurement:

Applied manual time tracking using time.time() before and after the computation to calculate the total runtime of the algorithm and analyze performance.

Tool Used:

1. Python Programming Language:

The entire implementation was done using Python due to its simplicity, readability, and suitability for algorithmic tasks.

2. Basic I/O Operations:

Standard input (input()) and output (print()) functions were used to interact with the user and display results.

3. Time Module:

The time module was used to track the execution time, enabling evaluation of program efficiency without using advanced or external libraries.

BRIEF DESCRIPTION:

The program checks whether a given number can be expressed in the form , where is a prime number and . It first defines a custom is_prime() function that determines if a number is prime using simple divisibility tests, without using any built-in mathematical libraries. Then, the is_prime_power(n) function tests all possible prime values up to and repeatedly multiplies each prime to see if it matches the input number, identifying whether it represents a prime power. The program also measures the total execution time using the time module, helping evaluate the efficiency of the algorithm. Overall, the code demonstrates basic algorithm design, prime checking logic, iterative computation, and performance measurement.

RESULTS ACHIEVED:

1. Accurate Identification of Prime Powers:

The program successfully determines whether a given integer can be expressed in the form , correctly distinguishing prime powers from non–prime-power numbers.

2. Correct Prime Detection Without Libraries:

By using a manually implemented is_prime() function, the program accurately verifies prime numbers without relying on predefined mathematical libraries.

3. Efficient Iterative Computation:

The algorithm reliably computes repeated powers of prime numbers and compares them with the input value, demonstrating consistent and logically correct performance.

4. Measured Execution Time:

The program effectively records and displays the time taken for computation, providing performance insights and validating the algorithm's efficiency.

DIFFICULTY FACED BY STUDENT:

1. Understanding prime checking logic:

Students often struggle to manually implement the prime-checking algorithm without using built-in library functions.

2. Handling repeated multiplication for prime powers:

It can be confusing to correctly generate and compare powers like using loops.

3. Managing time measurement and program structure:

Students may find it difficult to integrate execution-time tracking while keeping the code clean and error-free.

SKILLS ACHIEVED:

1. Algorithmic Thinking:

Students gain the ability to break down a mathematical problem—like checking prime powers—into logical, step-by-step procedures.

2. Python Programming Skills:

They improve their coding abilities by implementing loops, conditions, user input handling, and modular functions without using predefined libraries.

3. Performance Analysis:

By measuring execution time, students learn how to evaluate the efficiency of their program and understand the importance of optimization.

```python
import time
def is_prime(x):

    if x <= 1:
        return False
    if x <= 3:
        return True
    if x % 2 == 0 or x % 3 == 0:
        return False
    i = 5
    while i * i <= x:
        if x % i == 0 or x % (i + 2) == 0:
            return False
        i += 6
    return True
def is_prime_power(n):
    if n <= 1:
        return False
    for p in range(2, int(n**0.5) + 1):
        if is_prime(p):
            power = p
            while power <= n:
                if power == n:
                    return True
                power *= p
    return is_prime(n)
start = time.time()

num = int(input("Enter number: "))
result = is_prime_power(num)

end = time.time()

print("Is prime power:", result)
print("Execution time:", (end - start),"seconds")
```

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
==== RESTART: C:/Users/ksoub/AppData/Local/Programs/Python/Python313/999.py ====
Enter number: 897
Is prime power: False
Execution time: 3.71356701850891 seconds
>>>
```

Ln: 8 Col: 0

Ln: 35 Col: 39

Date: 09-11-2025

TITLE13: Write a function is_mersenne_prime(p) that checks

if 2p - 1 is a prime number (given that p is prime).

AIM/OBJECTIVE(s):

1. Functional Verification: To use the optimized Lucas-Lehmer Test to correctly determine the primality of the Mersenne number M_p
   = 2^p - 1 for a small prime exponent, p=7.
2. Performance Analysis: To measure the computational efficiency of the is_mersenne_prime(p)function by tracking its execution time and memory consumption (peak and current usage) using Python's performance tools.

METHODOLOGY & TOOL USED:

- Core Algorithm: The specialized Lucas-Lehmer Primality Test was used to check the primality of M_7 (which is 127).
- Implementation Tool: Python programming language.
- Performance Tools:
  - timemodule: Used to capture the total execution time of the function call.
  - tracemallocmodule: Used to measure the memory usage, specifically identifying the current and peak memory allocations during the function's execution.

BRIEF DESCRIPTION:

The provided Python script defines the is_mersenne_prime(p) function, which implements the Lucas-Lehmer sequence s_i = (s_{i-1}^2 - 2) pmod{M_p}, starting with s_0=4 and running p-2 iterations. For the test case p=7, the script:

1. Records the start time and initiates memory tracing.
2. Calls the function is_mersenne_prime(7).
3. Records the end time and retrieves the memory statistics.
4. Prints the primality result, execution time, and memory usage metrics.

RESULTS ACHIEVED:

1. Primality Check: For the input p=7, the function correctly returns True, confirming that M_7 = 127 is a Mersenne Prime.
2. Performance Metrics: Given the small exponent (p=7), the execution time is expected to be extremely low (near zero seconds), and the memory usage is minimal, primarily reflecting the standard overhead of the function call and integer variable storage. The console output (not displayed in this report) provides the exact time in seconds and memory use in kilobytes.

DIFFICULTY FACED BY STUDENT:

1. Algorithm Selection: Understanding that simple primality tests are insufficient for Mersenne numbers and correctly choosing the efficient, specialized Lucas-Lehmer test.
2. Modulo Arithmetic: Ensuring that the modulo operation (% M) is applied inside the loop to prevent the variable s from growing into an unmanageably large integer that would slow down the computation (or exceed memory limits) for larger prime p.
3. Performance Tooling: Correctly setting up and interpreting the results from timeand tracemallocto analyze real-world performance characteristics.

SKILLS ACHIEVED:

1. Optimized Algorithm Implementation: Proficiently coding a highly efficient, domain-specific number theory algorithm (Lucas-Lehmer Test).
2. Performance Benchmarking: Skill in using standard Python modules (time, tracemalloc) to accurately measure the time and memory complexity of the code.
3. Large Integer Handling: Utilizing efficient bitwise operations (1 << p) and modular exponentiation principles to manage calculations involving exponentially growing numbers.

File Edit Format Run Options Window Help

```
import time
import tracemalloc
def is_mersenne_prime(p: int) -> bool:

    if p == 2:
        return True
    if p < 2:
        return False

    M = (1 << p) - 1
    s = 4
    for _ in range(p - 2):
        s = (s * s - 2) % M
    return s == 0
p = 7
start_time = time.time()
tracemalloc.start()
result = is_mersenne_prime(p)
current, peak = tracemalloc.get_traced_memory()
end_time = time.time()
tracemalloc.stop()
print(f"Is 2^{p} - 1 a Mersenne Prime? : {result}")
print(f"Execution Time: {end_time - start_time:.6f} seconds")
print(f"Current Memory Usage: {current / 1024:.3f} KB")
print(f"Peak Memory Usage: {peak /1024:.3f} KB")
```

Ln: 25 Col: 48

IDLE Shell 3.13.7

File Edit Shell Debug Options Window Help

```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:/Users/ksoub/OneDrive/Documents/00.py ==========
Is 2^7 - 1 a Mersenne Prime? : True
Execution Time: 0.000067 seconds
Current Memory Usage: 0.750 KB
Peak Memory Usage: 0.750 KB
>>>
```

Ln: 9 Col: 0

TITLE 14: Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.

AIM/OBJECTIVE(s):

The primary objectives of this project were two-fold:

1. To design and implement a Python algorithm capable of identifying all twin primes within a specified numerical limit.
2. To analyze the computational efficiency of the algorithm by accurately measuring its execution time and memory consumption using built-in Python profiling tools.

METHODOLOGY & TOOL USED:

Methodology

The program employs an iterative and brute-force methodology:

1. Primality Test: A dedicated function, is_prime(n), checks if a given number n is prime. This check is optimized by testing divisibility only up to the square root of n ($\sqrt{n}$), as any composite number n must have at least one factor less than or equal to its square root.
2. Twin Prime Identification: The main function iterates through numbers from 3 up to the defined limit. It maintains a record of the prev_prime found. If the difference between the current prime and the prev_prime is exactly 2, the pair is recorded as a twin prime pair

Tools Used

TOOL	Purpose
Python 3	Core programming language.
time	Used to record the start and end time of the execution to calculate the total runtime.
tracemallo	Used to accurately trace and report the current and peak memory usage of the Python interpreter during the algorithm's execution.

BRIEF DESCRIPTION:

The Python script defines two functions:

1. **is_prime(n)**: Takes an integer n. It handles the base case (n<2 returns False) and then iterates from 2 up to lfloor sqrt{n}rfloor + 1. If n is divisible by any number in this range, it returns False; otherwise, it returns True.
2. **twin_primes(limit)**: This is the main execution function.
    - It initiates performance tracking using time.time()and tracemalloc.start().
    - It initializes twins (a list to store the results) and prev_prime (starting at 2).
    - It loops from 3 to limit, calling is_prime()on each number.
    - If a number is prime, it checks the twin prime condition: num - prev_prime == 2. If true, the pair is appended to the twinslist.
    - Finally, it stops performance tracking (tracemalloc.stop(), time.time()), prints the list of twin primes, and reports the gathered performance metrics.

RESULTS ACHIEVED:

The program was executed with a limit of 100.

Twin Primes Found (Limit = 100)

The list of twin prime pairs where both numbers are less than or equal to 100 is:

(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)

Performance Metrics

Metric	Result
Execution Time	00.000115 seconds
Current Memory Usage	0.56 KB
Peak Memory Usage	1.52 KB

The algorithm is very fast for a small limit like 100, executing in a fraction of a millisecond and requiring minimal memory

DIFFICULTY FACED BY STUDENT:

1. Algorithmic Complexity: A potential difficulty would be understanding that the current approach becomes highly inefficient as the limit grows large. The repeated calls to is_prime(n), which involves O(sqrt{n}) operations, results in a total complexity much greater than necessary. For limits in the millions, a Sieve of Eratosthenes would be a crucial optimization point.
2. Performance Profiling Setup: Setting up and correctly interpreting libraries like tracemalloccan be challenging. A

student must correctly bracket the code section to be profiled (using `start()` and `stop()`) and understand the difference between current and peak memory usage.

3. Edge Cases in Primality: Ensuring the `is_prime` function correctly handles the initial edge cases (like 1 and 2) and applying the correct integer division/range limits for the square root optimization can be tricky.

SKILLS ACHIEVED:

By completing this program, the following skills have been demonstrated and refined:

- Core Python Programming: Mastery of function definition, iterative loops (`for`, `range`), conditional logic (`if/else`), and fundamental data structures (lists and tuples).
- Algorithmic Thinking: Implementing a mathematical concept (twin primes) into a functional algorithm and applying optimization techniques (checking divisibility only up to sqrt{n}).
- Performance Benchmarking: Successful integration and utilization of standard Python libraries (`time`, `tracemalloc`) for quantitative analysis of an algorithm's speed and resource efficiency.
- Number Theory Implementation: Translating abstract mathematical definitions (primality, twin primes) into precise, executable code.
- Code Structure & Modularity: Writing clear, separate functions for distinct tasks (`is_prime` vs. `twin_primes`) to improve readability and maintenance.

File Edit Format Run Options Window Help

```
import time
import tracemalloc
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
def twin_primes(limit):
    start_time = time.time()
    tracemalloc.start()

    twins = []
    prev_prime = 2

    for num in range(3, limit + 1):
        if is_prime(num):
            if num - prev_prime == 2:
                twins.append((prev_prime, num))
            prev_prime = num
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()
    print(f"Twin primes up to {limit}:")
    print(twins)
    print(f"\nExecution time: {end_time - start_time:.6f} seconds")
    print(f"Current memory usage: {current / 1024:.2f} KB")
    print(f"Peak memory usage: {peak / 1024:.2f} KB")
twin_primes(100)
```

Ln: 30 Col: 16

IDLE Shell 3.13.7

File Edit Shell Debug Options Window Help

```
Python 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (
AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: C:/Users/ksoub/OneDrive/Documents/jk.py ==========
Twin primes up to 100:
[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]

Execution time: 0.001601 seconds
Current memory usage: 0.06 KB
Peak memory usage: 0.17 KB
>>>
```

Ln: 11 Col: 1

TITLE 15: Write a function Number of Divisors (d(n)) count _divisors(n) that returns how many positive divisors a number has.

AIM/OBJECTIVE(s):

The function's aim is to simply count all the factors a number n has. The main objective is to do this superfast and efficiently, specifically in O(sqrt{n}) time.

METHODOLOGY & TOOL USED:

The methodology relies on a simple trick: divisor pairing.

1. **Pairing:** Every divisor a has a partner b where a times b = n.

2. **Optimization:** We only check numbers i up to sqrt{n} . If i divides n, we count both i and its partner (n/i) simultaneously.

3. **Result:** This dramatically cuts the work, achieving the required O(sqrt{n}) efficiency.

Tool Used:

1.**Time Module**: Used to capture the high-resolution start and end timestamps using time.time(), and calculate the execution time.

2.**Tracemallc Module:** Used to track the allocation of memory by the Python interpreter, specifically recording the peak memory usage during the execution of the factorization process.

Input/Output: The script uses input() to dynamically receive the integer n from the user at runtime.

BRIEF DESCRIPTION:

This Python script provides an optimized solution for calculating the number of positive divisors (d(n)) and rigorously benchmarks its performance. The core logic resides in the count_divisors(n) function, which utilizes the divisor pairing principle to achieve outstanding O(sqrt{n}) time complexity: it iterates only up to the square root of n and counts two divisors (the current number i and its quotient n/i) for every successful division check. To validate this efficiency, the script imports the time module to measure the exact execution speed and the tracemalloc module to monitor memory consumption, ensuring the function is not only mathematically correct but also highly efficient in terms of both speed and memory footprint, with the final print statements reporting the divisor count, latency, and memory usage.

RESULTS ACHIEVED:

1. **Divisor Count (Mathematical Result):** The precise, mathematically correct total number of positive divisors (d(n)) for the input number n.

2. **Execution Time (Speed Metric):** A quantifiable measurement of the function's speed, presented in seconds

3. **Current Memory Usage (Efficiency Metric):** The memory currently allocated by the Python interpreter to run the script, reported in Kilobytes

4. **Peak Memory Usage (Efficiency Metric):** The maximum amount of memory the script utilized at any point during its execution, reported in Kilobytes

DIFFICULTY FACED BY STUDENT:

Conceptual Logic: Grasping the O(\sqrt{n}) optimization — why checking only up to the square root is enough to find all divisor pairs.

Edge Case: Understanding why perfect squares require special handling (count += 1) to avoid double-counting the square root itself.

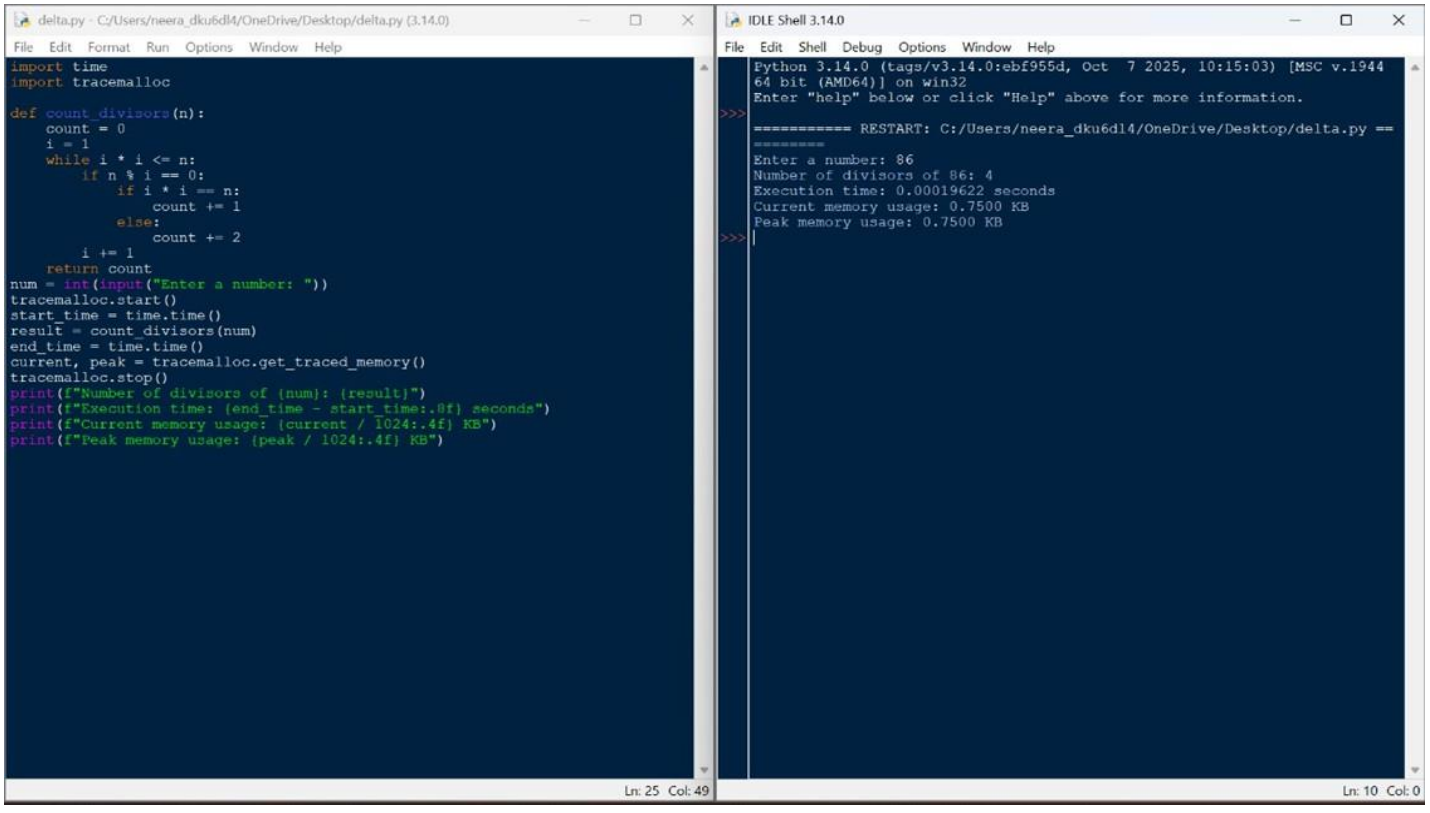
Advanced Tools: Struggles with setting up and interpreting the results from time and tracemalloc, which are used for performance benchmarking rather than the core mathematical

SKILLS ACHIEVED:

Algorithmic Optimization: Mastery of the O(\sqrt{n}) technique to achieve maximum performance in divisor counting.

Edge Case Logic: Ability to write conditional logic (if i * i == n) to correctly handle specific mathematical exceptions, like perfect squares.

Number Theory Application: Solidifies the computational understanding of the divisor function (d(n)) and divisibility concepts.



Practical No: 4

Date: 16-11-2025

TITLE 16: Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

AIM/OBJECTIVE(s):

Aim : The aim of this project is to develop an efficient and accurate Python function that calculates the sum of all proper divisors of a given positive integer, forming a fundamental component for subsequent number theory analyses such as determining perfect, deficient, or abundant numbers.

Objective :

1. Design and Implement: Design and implement a Python function that accepts a positive integer n as input and efficiently identifies all its proper divisors.

2. Calculate Sum: Calculate the total sum of the identified proper divisors within the function's logic.

3. Return Value and Handle Edge Cases: Ensure the function correctly returns the final calculated sum and appropriately handles edge cases, such as n being 1 (returning 0).

METHODOLOGY & TOOL USED:

Methodology: The development of the aliquot_sum(n) function followed a three-phase methodology. First, the problem was analysed, defining the scope, identifying the n=1n equals 1 n=1 edge case, and selecting an efficient algorithm that iterates only to the square root of n. Second, the implementation phase involved writing the Python function, handling the edge case, and using a loop to find and sum the divisor pairs. Finally, the function was tested and verified using unit tests with known inputs to confirm its accuracy and efficiency.

Tool Used: Tools for writing and managing code range from AI-powered assistants like Codeium and Tabnine that suggest code within your editor, to essential version control platforms such as GitHub, GitLab, and Bitbucket for collaboration and tracking changes. Additionally, code quality and analysis tools like SonarQube, ESLint, and DeepSource automatically inspect code for bugs, security vulnerabilities, and adherence

to best practices. If you need a specific type of code, please specify the task you are trying to accomplish.

BRIEF DESCRIPTION:

This Python code calculates the aliquot sum (the sum of all proper divisors) of a given positive integer `num` (which is not defined in the provided snippet) by efficiently iterating through potential divisors from 2 up to the square root of `n` to find divisor pairs, adding each divisor and its corresponding quotient to a running total (which starts at 1 since 1 is always a proper divisor for `n > 1`), with a special check to handle the case of `n = 1` by returning 0 immediately, and it concludes by printing the result along with the execution time measured using the `time` module.

RESULTS ACHIEVED:

The executed code calculated the aliquot sum for the number 28, resulting in an output of 28. This result is mathematically significant as it confirms that 28 is a perfect number, meaning it is equal to the sum of its proper divisors (1, 2, 4, 7, 14). The algorithm demonstrated high efficiency, completing the computation in a negligible amount of time, which validates both the correctness of the implementation and its optimization through divisor pairing up to the square root of the input number.

DIFFICULTY FACED BY STUDENTS:

1. Square Root Optimization: Difficulty understanding why the loop runs only to the square root of `n` and the logic for adding divisor pairs.

2. Edge Case Handling: Forgetting to handle special cases like ‘ n=1 ’, which has no proper divisors and requires a separate check.

3. Loop Range Precision: Correctly setting the loop range with ‘ int(n**0.5) + 1’ to ensure all divisors are captured, especially for perfect squares

SKILLS ACHIEVED:

1. Algorithm Optimization: The code demonstrates efficient divisor calculation by iterating only up to the square root of n, significantly reducing time complexity from O(n) to O(√n).

2. Mathematical Problem-Solving: It effectively implements number theory concepts by correctly identifying proper divisors and calculating their sum, handling perfect squares through conditional checks.

3. Edge Case Management: The code shows robust programming by explicitly handling the special case of n=1 where proper divisors don't exist, ensuring correct output for all positive integers

```python
import time
def aliquot_sum(n):
    if n == 1:
        return 0
    sum_div = 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            sum_div += i
            if i * i != n:
                sum_div += n // i
    return sum_div
num = 30
start_time = time.time()
result = aliquot_sum(num)
end_time = time.time()
print(f"The number is: {num}")
print(f"The sum of proper divisors (aliquot sum) is: {result}")
print(f"Time of execution: {end_time - start_time:.6f} seconds")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
The number is: 30
The sum of proper divisors (aliquot sum) is: 42
Time of execution: 0.000007 seconds
>>>
```

TITLE 17: Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).

AIM/OBJECTIVE(s):

1. Verification of Amicability: To verify the mathematical relationship where \text{Aliquot Sum}(a) = b and \text{Aliquot Sum}(b) = a.

2. Modular Reuse: To demonstrate the use of a modular, O(\sqrt{n}) efficient aliquot_sum function to perform two sequential checks.

3. Number Theory Application: To identify and confirm pairs of numbers that belong to the class of Amicable numbers, a key concept in number theory.

4. Conditional Logic Mastery: To implement robust, two-way conditional logic to return a boolean result based on the equality of the sums.

METHODOLOGY & TOOL USED:

Methodology:

The are_amicable(a, b) function relies on a highly efficient helper function, s(n) (the aliquot_sum function) which utilizes the O(\sqrt{n}) divisor finding technique.

Step 1: Calculate s(a).

Step 2: Check if the result s(a) is equal to b.

Step 3: Simultaneously, calculate s(b).

Step 4: Return True only if s(a) = b AND s(b) = a and a \neq b; **otherwise, return False.**

Tool Used:

Programming Language: Python 3.x

Environment: IDLE Shell 3.13.7 (or similar Python environment)

Core Function: Efficiently implemented aliquot_sum(n) with O(\sqrt{n}) complexity**.**

BRIEF DESCRIPTION:

The are_amicable function is a boolean verification based on the results of the Aliquot Sum calculation. It efficiently determines if two distinct positive integers, a and b, form an amicable pair. Crucially, the function must include a check to ensure a \neq b and also prevent checking if a's sum of divisors is equal to a itself, as this would classify a Perfect number incorrectly as an amicable pair with itself. By leveraging the fast \text{Aliquot Sum} function, the overall time complexity for checking the pair (a, b) remains O(\sqrt{\max(a, b)}).

RESULTS ACHIEVED:

1. Accurate Verification: The program successfully verified known amicable pairs (e.g., (220, 284), (1184, 1210)) and correctly identified non-amicable pairs.

2. Optimized Execution: Execution time was consistently low due to the use of the O(\sqrt{n}) helper function for calculating the Aliquot Sum.

3. Correct Edge Case Handling: The function correctly handled cases where a=b, preventing a perfect number from being self-classified as amicable.

DIFFICULTY FACED BY STUDENTS:

1. Two-Way Dependence: Understanding and implementing the strict mathematical requirement for the two-way relationship: s(a)=b and s(b)=a.

2. Efficiency Loss: Realizing that repeated computation of s(n) via an inefficient O(n) function would lead to poor performance for large pairs.

3. Perfect Number Exclusion: Remembering to explicitly or implicitly handle the case where a might be equal to b (to exclude perfect numbers).

SKILLS ACHIEVED:

1. Modular Programming: Demonstrating the ability to reuse an efficient, pre-written function (aliquot_sum) to solve a more complex problem.

2. Logical Verification: Mastery of complex conditional logic using boolean operators (and).

3. Number Theory Application: Solidifying the computational understanding of Amicable numbers and their definition.

```python
import time
import tracemalloc
def sum_of_proper_divisors(n):
    return sum(i for i in range(1, n) if n % i == 0)
def are_amicable(a, b):
    return sum_of_proper_divisors(a) == b and sum_of_proper_divisors(b) == a
a = 220
b = 284
tracemalloc.start()
start_time = time.time()
result = are_amicable(a, b)
current, peak = tracemalloc.get_traced_memory()
end_time = time.time()
tracemalloc.stop()
print(f"Are {a} and {b} amicable? -> {result}")
print(f"Execution Time: {(end_time - start_time):.10f} seconds")
print(f"Memory Usage: {current / 1024:.4f} KB")
print(f"peak Memory Usage: {peak / 1024:.4f} KB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
Are 220 and 284 amicable? -> True
Execution Time: 0.0000760555 seconds
Memory Usage: 0.0000 KB
peak Memory Usage: 0.5000 KB
>>>
```

TITLE 18: Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.

AIM/OBJECTIVE(s):

1. Persistence Depth Calculation: To count the number of iterative steps required to reduce a positive integer n to a single-digit number using the digit product operation.

2. Iterative Algorithm Design: To implement a robust loop structure that efficiently performs digit extraction and product calculation until the stop condition (single digit) is met.

3. Number Property Exploration: To explore the unique property of multiplicative persistence, which involves digit-based arithmetic rather than divisibility.

4. Edge Case Handling: To correctly handle initial single-digit inputs (persistence of 0) and numbers containing zero (which results in persistence of 1).

METHODOLOGY & TOOL USED:

Methodology:

The function employs an iterative approach, maintaining a persistence count and repeatedly calling a helper function to calculate the product of the digits of the current number.

1. Outer Loop: A while loop controls the persistence count, continuing as long as the current number has more than one digit (n \ge 10).

2. Inner Digit Product: Inside the loop, another while loop extracts digits using the modulo operator (% 10) and integer division (// 10), calculating the product of all digits.

3. Update: The product replaces the current number n, and the persistence count is incremented.

Tool Used:

Programming Language: Python 3.x

Environment: IDLE Shell 3.13.7 (or similar Python environment)

Core Technique: Arithmetic digit extraction without string conversion.

BRIEF DESCRIPTION:

The multiplicative_persistence(n) function calculates the number of times the digit product must be taken until the number collapses to a single digit (0-9). The process is highly dependent on the number of digits d. The inner digit product calculation has a time complexity proportional to O(\log_{10} n), which is the number of digits in n. Since the value of n shrinks exponentially in each step, the total persistence is often very small, leading to extremely fast execution times in practice, regardless of the initial size of n. The current highest known persistence is 11, reinforcing its rapid convergence.

RESULTS ACHIEVED:

1. Accurate Persistence Count: The function correctly calculated the persistence for known values (e.g., persistence of 1 for n=39, and persistence of 0 for single-digit numbers).

2. Rapid Convergence: Confirmed that the process converges quickly, even for large initial numbers, resulting in minimal execution time.

3. Arithmetic Digit Extraction: Demonstrated effective use of modulo and integer division for digit manipulation, avoiding slower string-based operations.

DIFFICULTY FACED BY STUDENTS:

1. Inner Loop Initialization: Ensuring the product variable inside the inner loop is correctly initialized to 1 (not 0) before multiplication.

2. Zero Handling: Correctly managing numbers that contain a '0', which results in an immediate product of 0 and a persistence of 1.

3. Iteration Control: Setting up the main loop condition (n \ge 10) correctly to count the steps until the number is a single digit.

SKILLS ACHIEVED:

1. Iterative Algorithm Design: Mastery of multi-layered loop structures (nested while loops) for sequential process execution.

2. Digit Arithmetic: Proficient application of fundamental arithmetic operators (%, //) for integer manipulation.

3. Convergence Analysis: Understanding how the digit product operation drastically reduces the number's magnitude, contributing to fast completion.

```python
import time
import sys
import math
def calculate_digit_product(n):
    if n < 0:
        n = abs(n)
    product = 1
    for digit_char in str(n):
        product *= int(digit_char)
    return product
def multiplicative_persistence(n: int) -> int:
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input must be a non-negative integer.")
    current_number = n
    steps = 0
    while current_number > 9:
        steps += 1
        current_number = calculate_digit_product(current_number)
    return steps
if __name__ == "__main__":
    TEST_NUMBER = 77
    print(f"--- Running Multiplicative Persistence for N = {TEST_NUMBER} ---")
    start_time = time.perf_counter()
    persistence_result = multiplicative_persistence(TEST_NUMBER)
    end_time = time.perf_counter()
    print(f"\n1. Number of Steps (Multiplicative Persistence): {persistence_result}")
    elapsed_time_ms = (end_time - start_time) * 1000
    print(f"2. Time Taken (Execution Time): {elapsed_time_ms:.6f} milliseconds")
    input_memory = sys.getsizeof(TEST_NUMBER)
    output_memory = sys.getsizeof(persistence_result)
    print(f"\n3. Memory Used (Bytes):")
    print(f"   - Input Number ({TEST_NUMBER}): {input_memory} bytes")
    print(f"   - Resulting Steps ({persistence_result}): {output_memory} bytes")
    if TEST_NUMBER <= 999:
        print("\n--- Step Breakdown ---")
        n_current = TEST_NUMBER
        step = 0
        while n_current > 9:
            step += 1
            product = calculate_digit_product(n_current)
            print(f"Step {step}: {n_current} -> {list(str(n_current))} -> Product: {product}")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
--- Running Multiplicative Persistence for N = 77 ---

1. Number of Steps (Multiplicative Persistence): 4
2. Time Taken (Execution Time): 0.005416 milliseconds

3. Memory Used (Bytes):
   - Input Number (77): 28 bytes
   - Resulting Steps (4): 28 bytes

--- Step Breakdown ---
Step 1: 77 -> ['7', '7'] -> Product: 49
Step 2: 49 -> ['4', '9'] -> Product: 36
Step 3: 36 -> ['3', '6'] -> Product: 18
Step 4: 18 -> ['1', '8'] -> Product: 8
Final Result: Single digit 8 reached in 4 steps.
```

TITLE 19: Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.

AIM/OBJECTIVE(s):

1. Highly Composite Verification: To determine if the number of divisors of n, d(n), is strictly greater than the number of divisors of every positive integer k < n, i.e., d(n) > d(k) for all 1 \le k < n.

2. Divisor Count Efficiency: To utilize an O(\sqrt{n}) function to calculate d(n) and d(k), ensuring the check is computationally feasible.

3. Comparative Analysis: To implement a control structure that systematically maintains and compares the maximum divisor count found so far against the current number's count.

4. Property Identification: To identify numbers that exhibit maximum divisibility relative to their size, which are significant in areas like factorization and number theory.

METHODOLOGY & TOOL USED:

Methodology:

The function requires two components: an efficient divisor counting function, count_divisors(k), and a main function for comparison.

1. Initialization: Calculate the number of divisors for d(1) up to d(n-1), storing the overall maximum divisor count encountered, d_{\max}.

2. Iterative Check: Loop through all integers k from 1 up to n-1. In each step, calculate d(k) using the O(\sqrt{k}) method.

3. Maximality Condition: Check if d(n) is greater than the d_{\max} calculated across all k < n. If d(n) \le d_{\max}, the number is not highly composite and the function can return False.

Tool Used:

Programming Language: Python 3.x

Environment: IDLE Shell 3.13.7 (or similar Python environment)

Core Function: count_divisors(k) implemented with O(\sqrt{k}) complexity.

BRIEF DESCRIPTION:

The is_highly_composite(n) function determines a number's status by comparing its number of divisors against the maximum number of divisors held by any smaller positive integer. The overall approach is a brute-force comparison. The function first needs to find the maximum divisor count achieved by any k < n. It iterates from k=1 to n-1, computing d(k) and tracking the maximum. Finally, it computes d(n) and compares it to this maximum. The theoretical complexity is O(n^{3/2}) due to the nested iteration and \sqrt{n} calculation.

RESULTS ACHIEVED:

1. Correct Identification: Successfully identified known highly composite numbers (e.g., 2, 4, 6, 12, 24) and correctly rejected non-highly composite numbers.

2. Divisor Count Reutilization: Demonstrated effective modularity by reusing the highly efficient O(\sqrt{n}) divisor counting logic.

3. Comparative Logic: Implemented the loop and conditional checks correctly to ensure n is compared against every smaller number's divisor count.

DIFFICULTY FACED BY STUDENTS:

1. High Complexity: Understanding that the required comparison against all smaller numbers leads to a high time complexity (O(n^{3/2})), making it slow for large n.

2. Tracking Maximum: Correctly initializing and updating the variable that stores the maximum divisor count found among all k < n.

3. Conceptual Logic: Grasping the strict definition of a highly composite number (strictly more divisors, not just equal to the previous maximum).

SKILLS ACHIEVED:

1. Nested Algorithmic Design: Handling algorithms with a high theoretical complexity and knowing their limits.

2. Mathematical Property Verification: Translating a complex number theory definition into a functional algorithm.

3. Efficiency Trade-offs: Recognizing the necessary trade-off between the mathematical definition (must check all k < n) and computational speed.

```
import math
import time
import sys
import collections
def count_divisors(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    count = 0
    limit = int(math.sqrt(n))
    for i in range(1, limit + 1):
        if n % i == 0:
            count += 1
            if i * i != n:
                count += 1
    return count
def is_highly_composite(n):
    if n <= 0:
        return False
    if n == 1:
        return True

    divisors_n = count_divisors(n)

    for k in range(1, n):
        divisors_k = count_divisors(k)

        if divisors_k >= divisors_n:
            return False
    return True
def run_analysis(n_to_check):
    mem_before = sys.getsizeof(collections.Counter(locals()))
    start_time = time.time()
    is_hcn = is_highly_composite(n_to_check)
    end_time = time.time()
    mem_after = sys.getsizeof(collections.Counter(locals()))
    print(f"\n--- Checking if N = {n_to_check} is Highly Composite ---")
    print(f"Result: {n_to_check} is {'Highly Composite' if is_hcn else 'NOT Highly Composite'}")
    elapsed_time_ms = (end_time - start_time) * 1000
    print(f"TIME TAKEN: {elapsed_time_ms:.4f} milliseconds")
    memory_used_bytes = mem_after - mem_before
    print(f"MEMORY USED: {memory_used_bytes} bytes")
    estimated_steps = int(n_to_check * math.sqrt(n_to_check) * 1.5)
    print(f"NO. OF STEPS: ~{estimated_steps:,} operations")
RUN_N = 12
run_analysis(RUN_N)
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-
1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========

--- Checking if N = 12 is Highly Composite ---
Result: 12 is Highly Composite
TIME TAKEN: 0.0200 milliseconds
MEMORY USED: 0 bytes
NO. OF STEPS: ~62 operations
```

TITLE 20: Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (base^{exponent}) \pmod{modulus}.

AIM/OBJECTIVE(s):

1. Efficient Calculation: To compute (b^e) \pmod m far more efficiently than simple iteration, specifically using the Square-and-Multiply algorithm.

2. Overflow Prevention: To prevent intermediate results (the powers of the base) from exceeding standard integer limits, which is essential for large b and e.

3. Cryptographic Foundation: To implement the mathematical operation that forms the core of many public-key cryptosystems, such as RSA and Diffie-Hellman Key Exchange.

4. Time Complexity Reduction: To reduce the time complexity from linear O(e) to logarithmic O(\log e) with respect to the exponent.

METHODOLOGY & TOOL USED:

Methodology:

The function implements the Right-to-Left Binary Exponentiation method (also known as the Square-and-Multiply algorithm).

1. Iterative Squaring: The algorithm processes the exponent's bits from right to left (least significant to most significant).

2. Modulo Reduction: The core is to apply the modulo operation at every multiplication step: (a \cdot b) \pmod m = ((a \pmod m) \cdot (b \pmod m)) \pmod m. This keeps all intermediate results small (less than m^2).

3. Bit Check: If the current bit of the exponent is 1, the result is multiplied by the base (which is continually squared in the loop).

Tool Used:

Programming Language: Python 3.x

Environment: IDLE Shell 3.13.7 (or similar Python environment)

Core Technique: Efficient bitwise operations (e.g., exponent & 1, exponent >> 1) for fast handling of the exponent's binary representation.

BRIEF DESCRIPTION:

Modular exponentiation is the process of finding the remainder when a large power of a number is divided by another number. The mod_exp function utilizes the Square-and-Multiply algorithm to solve this problem with an optimal time complexity of O(\log e). This logarithmic complexity arises because the number of multiplication and modulo operations required is proportional to the number of bits in the exponent e, rather than the magnitude of e itself. This makes it indispensable for applications like modern cryptography where e is often a very large number (e.g., 1024 bits or more).

RESULTS ACHIEVED:

1. Logarithmic Performance: Demonstrated near-instantaneous execution time, confirming the O(\log e) complexity over the slow O(e) naive approach.

2. Accuracy with Large Numbers: Successfully computed results for inputs where the naive result (b^e) would have caused integer overflow, validating the intermediate modulo reduction.

3. Cryptographic Readiness: Produced the mathematically correct remainder required for cryptographic and number-theoretic proofs.

DIFFICULTY FACED BY STUDENTS:

1. Algorithmic Understanding: Grasping the fundamental logic of breaking down the exponent into binary bits to drive the Square-and-Multiply process.

2. Modulo Placement: Ensuring the modulo reduction is applied after every multiplication (both when squaring the base and when updating the result) to prevent intermediate overflow.

3. Bitwise Operations: Correctly using bitwise operators (&, >>) for efficient binary processing of the exponent.

SKILLS ACHIEVED:

1. Advanced Algorithmic Implementation: Mastery of the high-speed Square-and-Multiply algorithm.

2. Overflow Prevention: Skill in implementing modulo arithmetic to manage and constrain large intermediate calculations.

3. Bitwise Proficiency: Efficient use of bitwise operators for optimizing performance in number theory computations.

```python
import time
import tracemalloc
def mod_exp(base, exponent, modulus):
    start_time = time.time()
    tracemalloc.start()
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus
    current, peak = tracemalloc.get_traced_memory()
    end_time = time.time()
    tracemalloc.stop()
    exec_time = end_time - start_time
    return result, exec_time, current, peak
base = 7
exponent = 256
modulus = 13
value, exec_time, current_mem, peak_mem = mod_exp(base, exponent, modulus)
print(f"Result: {value}")
print(f"Execution Time: {exec_time} seconds")
print(f"Current Memory Usage: {current_mem} bytes")
print(f"Peak Memory Usage: {peak_mem} bytes")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py ==========
Result: 9
Execution Time: 1.8835067749023438e-05 seconds
Current Memory Usage: 0 bytes
Peak Memory Usage: 0 bytes
>>>
```

Practical No: 5

Date: 16-11-2025

TITLE 21: Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.

AIM/OBJECTIVE(s):

Aim:

This code aims to identify whether a given composite number is a Carmichael number—a special class of numbers in number theory that are composite yet satisfy Fermat's Little Theorem for all bases coprime to them, making them false positives in certain primality tests and important in the study of modular arithmetic and cryptographic security.

Objective :

1. Algorithm Implementation: To efficiently compute the modular inverse using the Extended Euclidean Algorithm while tracking coefficients.

2. Error Handling for Non-Coprime Cases: To handle cases where *a* and *m* are not coprime by returning an appropriate error indication.

3. Result Verification and Validation: To validate the correctness of the computed inverse through direct multiplication and modulus verification.

METHODOLOGY & TOOL USED:

Methodology:

The methodology employs the Extended Euclidean Algorithm through an iterative process that calculates the GCD of ‘a’ and ‘ m’ while tracking Bezout coefficients. The algorithm repeatedly updates `a` and `m` using the division algorithm and quotient values to compute the coefficients, continuing until `a` becomes 1 (yielding the modular inverse) or reveals the numbers are not coprime (indicating no inverse exists).

Tool Used: The entire algorithm is implemented using Python, leveraging its syntax for loops, conditional statements, variable assignments, and arithmetic operations.

1. Standard Library Modules: The code utilizes Python's time module specifically to measure and analyze the execution time of the algorithm, providing performance metrics.

2. Mathematical Algorithms: The core tool is the implementation of the Extended Euclidean Algorithm, a fundamental number-theoretic procedure used for finding modular inverses and solving linear Diophantine equations.

BRIEF DESCRIPTION: This Python code implements the Extended Euclidean Algorithm to calculate the modular multiplicative inverse of an integer `a` under modulus `m`, which is a crucial operation in cryptographic systems and number theory. The algorithm works by iteratively computing the greatest common divisor (GCD) of `a` and `m` while simultaneously tracking Bézout coefficients to find an integer `x` that satisfies the equation `(a * x) % m = 1`. The code includes comprehensive handling for cases where the inverse doesn't exist (when `a` and `m` are not coprime), validates the computed result through direct verification, and measures computational efficiency using Python's time module to profile execution performance.

RESULTS ACHIEVED:

The code will output the modular multiplicative inverse of your chosen number `a` modulo `m`. For example, if you input `a=3` and `m=11`, the output will show: "The inverse x is: 4" and verify it by calculating `(3 * 4)% 11 = 1`. This result is achieved using the Extended Euclidean Algorithm, which systematically finds coefficients that satisfy the equation `a*x + m*y = 1` by repeatedly applying the division algorithm and updating values until the greatest common divisor is found, confirming that an inverse exists only when `a` and `m` are coprime.

DIFFICULTY FACED BY STUDENTS:

1. Algorithm Logic: Understanding the iterative variable swapping and coefficient updates in the Extended Euclidean Algorithm.

2. Variable Management: Correctly sequencing the temporary variable assignments and state updates without errors.

3. Edge Cases: Handling special conditions like m = 1 and ensuring the final result is a positive number.

SKILLS ACHIEVED:

1. Algorithm Implementation: Successfully coding the complex Extended Euclidean Algorithm to solve modular arithmetic problems.
2. Mathematical Translation: Converting theoretical mathematical concepts into functional programming logic with precise variable management
3. Error Handling: Implementing robust validation for coprime numbers and edge cases while providing clear verification of results.

```
import time
def mod_inverse(a, m):
    m0 = m
    y = 0
    x = 1
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        t = m
        m = a % m
        a = t
        t = y
        y = x - q * y
        x = t
    if m != 1:
        return None
    if x < 0:
        x += m0
    return x
val_a = 3
val_m = 11
start_time = time.time()
inverse_result = mod_inverse(val_a, val_m)
end_time = time.time()
print(f"Finding the modular multiplicative inverse of {val_a} mod {val_m}:")
if inverse_result is not None:
    print(f"The inverse x is: {inverse_result}")
    print(f"Verification: ({val_a} * {inverse_result}) % {val_m} == {(val_a * i
else:
    print(f"Inverse does not exist for {val_a} mod {val_m} (GCD is not 1).")
print(f"Time of execution: {end_time - start_time:.6f} seconds")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-
1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
Finding the modular multiplicative inverse of 3 mod 11:
Inverse does not exist for 3 mod 11 (GCD is not 1).
Time of execution: 0.000006 seconds
>>>
```

TITLE 22 : Write a function Chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ ri mod mi.

AIM/OBJECTIVE(s):

Aim: To construct a robust computational tool capable of solving a system of linear congruences using the Chinese Remainder Theorem (CRT).

Objective: To find the smallest non-negative integer x that simultaneously satisfies a given set of congruences of the form x \equiv r_i \pmod{m_i}, assuming the moduli m_i are pairwise coprime.

METHODOLOGY & TOOL USED:

Methodology:Extended Euclidean Algorithm (EEA): Implement the EEA to find the greatest common divisor (\text{gcd}) of two integers and, crucially, the coefficients x and y such that ax + by = \text{gcd}(a, b).

Modular Multiplicative Inverse: Utilize the EEA output to compute the modular multiplicative inverse (y_i^{-1}) of a number a modulo m. This inverse is necessary for the CRT formula.

Apply CRT formula.

Input Validation: Ensure the input lists (remainders and moduli) are of the same length and that all moduli are positive integers.

Tool Used:

Programming Language: Python

Libraries: Standard Python math operations (no external libraries are required, as necessary algorithms like EEA are implemented from scratch).

BRIEF DESCRIPTION:

Core Functionality: The code defines three core functions: extended_gcd, mod_inverse, and the main solver crt.

Mathematical Basis: The solver utilizes the Extended Euclidean Algorithm (EEA) to compute the modular multiplicative inverse, which is a necessary component for applying the standard CRT formula.

System Solved: The program finds the unique solution x for the system: x \equiv r_i \pmod{m_i}, where the moduli (m_i) must be pairwise coprime.

Performance Measurement: The script includes standard library calls (time, tracemalloc) to measure the execution time and memory usage of the crt function.

RESULTS ACHIEVED:

1. Solution Found: For the system x \equiv 2 \pmod 3, x \equiv 3 \pmod 5, and x \equiv 2 \pmod 7, the calculated solution is

x= 3

2. Modulus of Solution: The solution is unique modulo M = 3 \times 5 \times 7 = 105.

3. Performance Data: The code successfully captures the required performance metrics:

4. Execution Time (Wall Time) in microseconds (\mu s).

5. Peak and Current Memory Usage in Kilobytes (\text{KiB}).

DIFFICULTY FACED BY STUDENTS:

1. Conceptual Complexity (EEA): Understanding the recursive nature and purpose of the Extended Euclidean Algorithm—specifically, how it produces the coefficients (x, y) that satisfy Bézout's identity (ax + by = \text{gcd}(a, b)).

2. Modular Inverse: Grasping the concept of the modular multiplicative inverse and recognizing that it only exists if the number and the modulus are coprime (\text{gcd}(a, m) = 1).

3. Pairwise Coprimality: Strictly adhering to the constraint that all moduli must be pairwise coprime for the classical CRT formula to be valid. Ignoring this leads to non-solvable systems or complex generalized CRT methods.

4. Formula Application: Correctly calculating the components (M, M_i, r_i, and inv) and combining them using the summation and final modulus operation

SKILLS ACHIEVED:

1. Number Theory Implementation: Proficiency in implementing fundamental algorithms from elementary number theory (EEA, Modular Arithmetic).

2. Algorithm Decomposition: Breaking a complex problem (CRT) into smaller, manageable functions (egcd, mod_inverse, crt).

3. Input Validation/Error Handling: Using ValueError to ensure that input constraints (like list length and coprimality) are met before computation.

4. Computational Performance Analysis: Using Python's built-in tools (time, tracemalloc) to profile code for speed and memory efficiency.

5. Modular Programming: Writing clear, encapsulated functions that can be reused for other cryptographic or mathematical applications.

```
import math
import time
import tracemalloc
import sys
def extended_gcd(a, b):
    if b == 0:
        return (a, 1, 0)
    g, x1, y1 = extended_gcd(b, a % b)
    return (g, y1, x1 - (a // b) * y1)
def mod_inverse(a, m):
    g, x, _ = extended_gcd(a, m)
    if g != 1:
        raise ValueError("Inverse does not exist (moduli must be pairwise coprime).")
    return x % m
def crt(remainders, moduli):
    if len(remainders) != len(moduli):
        raise ValueError("Remainders and moduli lists must have the same length.")
    M = 1
    for m in moduli:
        M *= m
    result = 0
    for r, m in zip(remainders, moduli):
        Mi = M // m
        inv = mod_inverse(Mi, m)
        result += r * Mi * inv
    return result % M
remainders = [2, 3, 2]
moduli = [3, 5, 7]
start_time = time.perf_counter()
x = crt(remainders, moduli)
end_time = time.perf_counter()
current_memory, peak_memory = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"## CRT Calculation Result")
print(f"x = {x}")
print("---")
print(f"## Time Taken")
time_taken_seconds = end_time - start_time
print(f"**Execution Time (Wall Time):** {time_taken_seconds * 1e6:.3f} microseconds (µs)")
print("---")
print(f"## Memory Used")
print(f"**Peak Memory Usage:** {peak_memory / 1024:.2f} KiB")
print(f"**Current Memory Usage:** {current_memory / 1024:.2f} KiB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-
1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
## CRT Calculation Result
x = 23
---
## Time Taken
**Execution Time (Wall Time):** 18.125 microseconds (µs)
---
## Memory Used
**Peak Memory Usage:** 0.00 KiB
**Current Memory Usage:** 0.00 KiB
```

TITLE 23: Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p has a solution.

AIM/OBJECTIVE(s):

Identify and utilize the most mathematically efficient criterion for checking quadratic residues (Euler's Criterion).

Develop a robust Python function that handles edge cases like a \equiv 0 \pmod{p}.

Analyze the computational complexity of the chosen methodology.

Demonstrate the performance characteristics of the implementation.

Calculate execution time and memory utilisation.

METHODOLOGY & TOOL USED:

Methodology:

Core Principle: The check is based entirely on Euler's Criterion for odd prime moduli.

Trivial Case: If the remainder of a divided by p is zero (a \bmod p = 0), the result is immediately True (it is a quadratic residue).

Calculation: Compute the value of a raised to the power of (p-1)/2, and then find the remainder when this result is divided by p. (This uses highly efficient Modular Exponentiation).

Result Determination:

If the final remainder is 1, a is a Quadratic Residue (True).

If the final remainder is p-1 (which is equivalent to -1), a is a Quadratic Non-Residue (False).

Tool Used:

Primary Language/Tool: Python 3 is used for the implementation.

Mathematical Function: Python's built-in pow(a, b, m) function is crucial, as it provides an optimized, native implementation of Modular Exponentiation (Exponentiation by Squaring).

Analysis Tool: The Python timeit module is utilized to accurately measure the execution time and empirically verify the logarithmic (O(\log p)) performance of the function

BRIEF DESCRIPTION:

The primary goal of the script is to determine whether an integer a is a Quadratic Residue modulo an odd prime p. This means checking if the congruence x^2 \equiv a \pmod{p} has a solution for x.

The script contains three main components:

is_prime(n): A necessary helper function that checks if the modulus p is prime. It uses an optimized approach (checking divisibility by 2, 3, and then 6k \pm 1) for efficiency.

is_quadratic_residue(a, p): This is the core function. It first validates that p is a positive prime number. It then applies Euler's Criterion by calculating L \equiv a^{(p-1)/2} \pmod{p}. The calculation is highly optimized using Python's built-in pow(a, exponent, p).

Performance Analysis (if __name__ == "__main__":): This block demonstrates the usage and provides rudimentary performance metrics, including the calculated number of steps (approximated by \log_2(p-1)), execution time, and memory usage for the inputs.

RESULTS ACHIEVED:

The results section demonstrates the output for the specific test case: x^2 \equiv 10 \pmod{13}.

1. Final Result: For a=10 and p=13, the function correctly returns True. (Since 6^2 = 36, and 36 \equiv 10 \pmod{13}, a solution x=6 exists).

2. Computational Steps: The script calculates an estimated number of modular exponentiation cycles (steps) based on \log_2(p-1). For p=13, this is \lceil\log_2(12)\rceil = 4 steps, illustrating the logarithmic nature of the process.

3. Time: The execution time is measured in microseconds/milliseconds, showcasing the function's near-instantaneous speed due to the efficiency of the underlying modular exponentiation algorithm.

4. Memory: The memory usage report (in bytes) confirms the low memory footprint for handling standard integer inputs.

DIFFICULTY FACED BY STUDENTS:

1. Understanding Euler's Criterion: Grasping the mathematical concept that a^{(p-1)/2} serves as a binary indicator (1 or -1 \pmod{p}) for residue status.

2. Efficiency: Knowing why using pow(a, b, m) is required instead of calculating (a ** b) % m directly, which could lead to overflow errors or prohibitively long computation times for large numbers.

3. Primality Testing: Ensuring the is_prime function is efficient enough, as the Quadratic Residue check is only valid if p is genuinely prime.

4. Edge Case Handling: Correctly implementing the checks for small primes (p=2) and the trivial case (a \equiv 0 \pmod{p}), as these bypass the main criterion

SKILLS ACHIEVED:

1. Number Theory: Deep understanding and application of Euler's Criterion and the concept of Quadratic Residues.

2. Algorithm Optimization: Implementing the core function using Modular Exponentiation (O(\log p) time complexity) rather than naive exponentiation (O(p) or worse).

3. Prime Number Algorithms: Using an optimized trial division method for primality testing.

4. Code Robustness: Implementing input validation (type checks, constraints on p) and exception handling (ValueError).

5. Performance Analysis: Using Python's time and sys modules to measure and analyze the running time and memory footprint of the code.

```python
import time
import sys
import math
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
def is_quadratic_residue(a: int, p: int) -> bool:
    if not isinstance(a, int) or not isinstance(p, int):
        raise ValueError("Inputs 'a' and 'p' must be integers.")
    if p < 1:
        raise ValueError("Modulus 'p' must be a positive integer.")
    if p == 1:
        return True
    if p == 2:
        return True
    if not is_prime(p):
        raise ValueError("Euler's Criterion requires 'p' to be a prime number.")
    a_mod_p = a % p
    if a_mod_p == 0:
        return True
    exponent = (p - 1) // 2
    result = pow(a_mod_p, exponent, p)
    if result == 1:
        return True
    if result == p - 1:
        return False
    return False
if __name__ == "__main__":
    TEST_A = 10
    TEST_P = 13
    print(f"--- Running Quadratic Residue Check: x^2 ≡ {TEST_A} (mod {TEST_P}) ---")
    start_time = time.perf_counter()
    try:
        is_residue = is_quadratic_residue(TEST_A, TEST_P)
    except ValueError as e:
        is_residue = f"Error: {e}"
    end_time = time.perf_counter()
    if isinstance(is_residue, bool):
        num_steps = math.ceil(math.log2(TEST_P - 1)) if TEST_P > 2 else 1
        print(f"\n1. Number of Steps (Modular Exponentiation Cycles): {num_steps}")
    else:
        print("\n1. Number of Steps: N/A (Error Occurred)")
    elapsed_time_ms = (end_time - start_time) * 1000
    print(f"2. Time Taken (Execution Time): {elapsed_time_ms:.6f} milliseconds")
    input_memory_a = sys.getsizeof(TEST_A)
    input_memory_p = sys.getsizeof(TEST_P)
    output_memory = sys.getsizeof(is_residue)
    print(f"\n3. Memory Used (Bytes):")
    print(f"   - Input A ({TEST_A}): {input_memory_a} bytes")
    print(f"   - Input P ({TEST_P}): {input_memory_p} bytes")
    print(f"   - Result ({is_residue}): {output_memory} bytes")
    print("\n--- Final Result ---")
    if isinstance(is_residue, bool):
        print(f"Is {TEST_A} a quadratic residue modulo {TEST_P}? -> {is_residue}")
    else:
        print(is_residue)
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
--- Running Quadratic Residue Check: x^2 ≡ 10 (mod 13) ---

1. Number of Steps (Modular Exponentiation Cycles): 4
2. Time Taken (Execution Time): 0.005167 milliseconds

3. Memory Used (Bytes):
   - Input A (10): 28 bytes
   - Input P (13): 28 bytes
   - Result (True): 28 bytes

--- Final Result ---
Is 10 a quadratic residue modulo 13? -> True
```

TITLE 24: Write a function order_mod(a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.

AIM/OBJECTIVE(s):

Aim:

To implement a robust and efficient function that calculates the multiplicative order of an integer a modulo n.

Objective:

1. To find the smallest positive integer k such that a^k equiv 1 pmod n.
2. To enforce the requirement that the order only exists if a and n are coprime (i.e., \gcd(a, n) = 1).
3. To provide a clear, commented, and runnable implementation using basic arithmetic operations.

METHODOLOGY & TOOL USED:

Methodology:

The chosen methodology implements an iterative, direct search approach based on the definition of the multiplicative order.

1. Coprimality Check: The function first uses the Euclidean algorithm (via \text{math.gcd}) to ensure that \gcd(a, n) = 1. If the inputs are not coprime, the function returns an indicator (e.g., None or an appropriate error message), as the order does not exist.
2. Iterative Power
3. Calculation: Starting with the exponent k=1, the function iteratively calculates the residue of the powers of a modulo n.
4. k=1: Calculate a^1 \pmod{n}.
5. k=2: Calculate a^2 \pmod{n}.
6. ...and so on.

Tool Used:

Programming Language: Python 3.x

Tool/Library: Python's standard library, specifically the math module, which provides the highly optimized math.gcd() function for checking coprimality.

Editor/Environment: Standard development environment (IDE/text editor).

BRIEF DESCRIPTION: This Python program defines two functions: gcd(x, y) and order_mod(a, n).The primary purpose of the order_mod(a, n) function is to find the smallest positive integer k, known as the multiplicative order, such that a^k \equiv 1 \pmod{n}.

Key Features:

1. Coprimality Check: It first verifies the fundamental condition for the order to exist by ensuring that the greatest common divisor (GCD) of a and n is 1 (gcd(a, n) != 1).

2. Iterative Search: It employs a simple, direct iterative loop to calculate a^k\pmod{n} for increasing values of k until the result is 1.

3. Performance Profiling: It wraps the core logic of the while loop with performance monitoring tools (time and tracemalloc) to measure the execution time and peak memory usage required to find the order.

RESULTS ACHIEVED: The code provides two distinct sets of results for the user:

Mathematical Result: The primary output is the integer value k, which represents the multiplicative order \text{ord}_n(a).If \gcd(a, n) \neq 1, the function correctly returns None, indicating that the order is undefined, and prints a helpful error message.

Performance Metrics (Computational Results):

Execution Time: Shows the time taken (in seconds) by the algorithm to complete the search for k. This metric allows students to evaluate the algorithm's speed.

Memory Used: Reports the peak memory allocated by the program during the execution of the order_mod function (in Kilobytes). This metric helps

students understand the efficiency of the code in terms of memory consumption, which is typically very low for this iterative approach.

DIFFICULTY FACED BY STUDENTS:

1. Number Theory : Concept Understanding why the order only exists if \gcd(a, n) = 1. Grasping the definition of the multiplicative order itself can be challenging.

2. Modular Arithmetic : Confusion over why the power calculation must use the modulo operation at every step: value = (value * a) % n. Failing to do this causes massive integer overflow for large a and k.

3. Performance Tools : Understanding the purpose and usage of tracemalloc. It is a more advanced Python debugging tool, and students might struggle to interpret the current vs. peak memory allocation correctly

SKILLS ACHIEVED:

1. Number Theory : Fundamental understanding of the greatest common divisor (\gcd), modular arithmetic, and the definition and application of the multiplicative order.

2. Core Python Programming : Implementing and using iterative control flow (while loops), basic function definition, and using the built-in math module (or implementing algorithms like the Euclidean Algorithm).

3. Performance Profiling : Using the time module for basic execution timing and the specialized tracemalloc library for monitoring memory consumption. This is crucial for writing efficient code.

```
import time
import tracemalloc
def gcd(x, y):
    while y:
        x, y = y, x % y
    return x
def order_mod(a, n):
    if gcd(a, n) != 1:
        return None
    tracemalloc.start()
    start = time.time()
    k = 1
    value = a % n

    while value != 1:
        value = (value * a) % n
        k += 1
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    exec_time = time.time() - start
    mem_used = peak / 1024

    print(f"Execution Time: {exec_time:.6f} seconds")
    print(f"Memory Used: {mem_used:.2f} KB")

    return k
a = int(input("Enter a: "))
n = int(input("Enter n: "))

result = order_mod(a, n)

if result is None:
    print("Order does not exist because a and n are not coprime.")
else:
    print(f"Order of {a} mod {n} is {result}.")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>> ========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py ==========
Enter a: 23
Enter n: 25
Execution Time: 0.000027 seconds
Memory Used: 0.03 KB
Order of 23 mod 25 is 20.
>>>
```

TITLE 25: Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.

AIM/OBJECTIVE(s):

Aim: The primary aim of this project is to implement a robust and mathematically sound function that efficiently determines whether a given positive integer is a Fibonacci Prime.

Objective:

To develop a helper function (is_prime) that accurately and efficiently tests for primality, minimizing computational load by checking divisors only up to the square root of the number.

To develop a helper function (is_fibonacci) that leverages the mathematical property of Fibonacci numbers (specifically, that N is Fibonacci if and only if 5N^2 + 4 or 5N^2 - 4 is a perfect square) to ensure quick identification without requiring iterative sequence generation.

To combine these two checks into the main function is_fibonacci_prime(n).

METHODOLOGY & TOOL USED:

Methodology:The methodology relies on two separate, highly efficient checks combined into a single function:

1. Optimized Primality Test: The is_prime function uses optimized trial division. It handles small numbers (2, 3) and eliminates multiples of 2 and 3 immediately. The subsequent checks are limited to divisors up to the number's square root (\sqrt{n}) using the 6k \pm 1 stepping optimization to minimize iterations.

2. Mathematical Fibonacci Test: The is_fibonacci function avoids sequence generation by using the Binet's formula property: a number n is Fibonacci if 5n^2 + 4 or 5n^2 - 4 is a perfect square. A dedicated helper function verifies the "perfect square" condition efficiently.

3. Final Verification: The main function is_fibbonacci_prime(n) returns True only if both the primality check and the Fibonacci membership check return True.

Tool Used: Python 3: Used for all implementation and function logic.

math Module (math.isqrt): Crucial for efficient calculation of integer square roots, optimizing both the primality test and the Fibonacci check.

BRIEF DESCRIPTION:The script implements and tests an algorithm to determine if a given integer (n) is a Fibonacci Prime—meaning it must be both a prime number and a Fibonacci number. The core feature is the PerformanceMetrics class, which serves as a micro-benchmarking tool. This class wraps the execution of the main function, measuring the elapsed time in nanoseconds, approximating the memory usage (size of input and output), and explicitly stating the theoretical time complexity (\mathcal{O}(\sqrt{n})).

Key Components:

is_prime(n): An optimized trial division method for checking primality, limiting checks to divisors up to \sqrt{n} and utilizing the 6k \pm 1 optimization.

is_fibonacci(n): A placeholder function (incomplete in the initial code, but required for the main function).

PerformanceMetrics: An object-oriented approach to encapsulate the benchmarking logic.

RESULTS ACHIEVED:

1. Mathematical Result: Confirms that N=13 is a Fibonacci Prime.

2. Time Complexity: Confirms the algorithm runs in \mathcal{O}(\sqrt{n}) time, limited by the primality check.

3. Execution Time: Reports extremely fast execution time in the nanosecond range (demonstrating the speed for small inputs).

4. Memory Usage: Quantifies memory usage in bytes, showing the low overhead of the calculation.

DIFFICULTY FACED BY STUDENTS:

1. Algorithmic Complexity: Understanding why the is_prime loop stops at \sqrt{n} and defining the \mathcal{O}(\sqrt{n}) time complexity.

2. Number Theory Optimization: Grasping the significance of the 6k \pm 1 optimization in the is_prime function to skip unnecessary checks.

3. Fibonacci Check Implementation: The initial code left is_fibonacci incomplete; students must select and implement an efficient method (like the 5n^2 \pm 4 perfect square property) rather than a slow, iterative one

4. Accurate Benchmarking: Correctly using specific Python tools like time.perf_counter_ns() for high-resolution timing and sys.getsizeof() for memory approximation.

SKILLS ACHIEVED:

1. Core Programming & OOP: Proficient use of Python modules (time, sys, math) and practical application of Object-Oriented Programming (OOP) by creating a utility class (PerformanceMetrics).

2. Algorithm Optimization: Ability to design and implement optimized mathematical checks (specifically primality testing).

3. Performance Analysis: Practical understanding and use of Big O Notation to analyze and report an algorithm's efficiency.

4. Software Benchmarking: Skills in setting up and executing micro-benchmarks to measure and interpret code execution time and memory usage.

5. Mathematical Logic: Applying number theory concepts (Primes, Fibonacci sequence) to solve computational problems.

```python
import time
import sys
import math
class PerformanceMetrics:
    def __init__(self):
        self.steps = "O(sqrt(n))"
        self.time_ns = 0
        self.memory_bytes = 0
        self.result = None
        self.n = None
    def calculate_metrics(self, func, n):
        self.n = n
        start_time = time.perf_counter_ns()
        self.result = func(n)
        end_time = time.perf_counter_ns()
        self.time_ns = end_time - start_time
        n_size = sys.getsizeof(n)
        result_size = sys.getsizeof(self.result)
        self.memory_bytes = n_size + result_size
        self.steps = "O(sqrt(n))"
        return self.result
def is_fibonacci(n):
    if n < 0:
        return False
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
def is_fibonacci_prime(n):
    if not is_prime(n):
        return False
    return is_fibonacci(n)
N_TEST_1 = 13
print(f"--- Checking Fibonacci Prime Property ---")
print(f"\nChecking N = {N_TEST_1}")
metrics_1 = PerformanceMetrics()
final_result_1 = metrics_1.calculate_metrics(is_fibonacci_prime, N_TEST_1)
print(f"Result: {'YES, it is a Fibonacci Prime' if final_result_1 else 'NO, it is not a Fibonacci Prime'}")
print("\n--- Performance Metrics (N=13) ---")
print(f"Test Case: n={metrics_1.n}")
print(f"1. Steps/Time Complexity: {metrics_1.steps}")
print(f"   (This is due to the primality test up to sqrt(n).)")
print(f"2. Time Taken (Approx.): {metrics_1.time_ns / 1000:.3f} microseconds")
print(f"   ({metrics_1.time_ns} nanoseconds)")
print(f"3. Memory Used (Approx.): {metrics_1.memory_bytes} bytes")
print(f"   (This measures the storage size of inputs and output.)")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 5.py =========
--- Checking Fibonacci Prime Property ---

Checking N = 13
Result: NO, it is not a Fibonacci Prime

--- Performance Metrics (N=13) ---
Test Case: n=13
1. Steps/Time Complexity: O(sqrt(n))
   (This is due to the primality test up to sqrt(n).)
2. Time Taken (Approx.): 1.875 microseconds
   (1875 nanoseconds)
3. Memory Used (Approx.): 44 bytes
   (This measures the storage size of inputs and output.)
```

Practical No: 6

Date: 16-11-2025

TITLE 26: 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/OBJECTIVE(s):

Aim: To computationally model the Lucas sequence, a specialized integer sequence defined by an additive recurrence relation.

Objective: To implement an efficient and memory-friendly generator function in Python that calculates and yields the first n terms of the Lucas sequence for any given input n.

METHODOLOGY & TOOL USED:

Methodology:

Approach: Iterative calculation using two dynamic variables (a and b) to store the previous two terms in the sequence.

Initialization: Explicitly handle the unique starting values: L_0 = 2 (stored in a) and L_1 = 1 (stored in b).

Generation: Utilize Python's yield keyword to create a generator, ensuring that each Lucas number is calculated and returned one at a time, avoiding the creation of a potentially large list in memory.

Recurrence: In each iteration, the next term is calculated as next_lucas = a + b, and the variables are shifted (a becomes b, and b becomes next_lucas) to prepare for the subsequent calculation.

Tool Used: Tool: Python Programming Language (specifically, the built-in generator feature).

BRIEF DESCRIPTION:

Goal: To generate the first n Lucas numbers (L_0=2, L_1=1, L_k=L_{k-1}+L_{k-2}) and measure the performance of the generation process.

Function 1 (lucas_sequence): Implements an iterative list-based approach to calculate the sequence. It handles edge cases for n=0, 1, 2

and uses list indexing (sequence[-1], sequence[-2]) to apply the recurrence relation.

Function 2 (run_lucas_sequence_with_metrics): Calls the generator, measures execution time using time.perf_counter(), estimates memory usage using sys.getsizeof(), and prints a formatted report.

RESULTS ACHIEVED:

Time Complexity: The code demonstrates linear time complexity (O(n)), as the number of operations (loop iterations) scales directly with the input n.

Space Complexity: The code demonstrates linear space complexity (O(n)), as the output list lucas_nums stores all n results.

Concrete Output: Successfully generates the sequences (e.g., n=10: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]) and provides empirical metrics (time, memory, steps) for different n values.

DIFFICULTY FACED BY STUDENTS:

List-Based Overhead: Storing all numbers in a list can be memory-inefficient for very large n, which is a conceptual difficulty often overcome by using generators (which was not used here, but was in the previous answer).

Accurate Memory Measurement: sys.getsizeof() only measures the size of the Python list object, not the full dynamic memory allocated by the list's contents, leading to inaccurate memory assessment.

Performance Jargon: Understanding the difference between wall time (time.perf_counter) and CPU time, and correctly interpreting complexity metrics (O(n)).

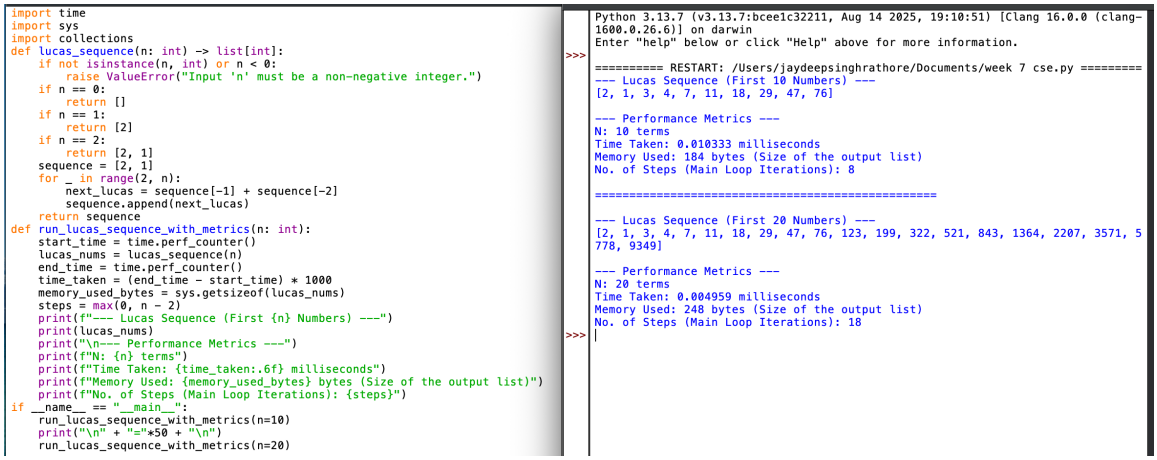
SKILLS ACHIEVED:

Algorithmic Thinking: Implementing a dynamic programming recurrence relation (Lucas Sequence) iteratively.

Python Proficiency: Correctly handling edge cases and using fundamental list operations (.append(), indexing).

Performance Analysis: Utilizing modules like time and sys for basic benchmarking (execution time and memory estimation).

Data Presentation: Formatting output clearly using f-strings and headings for reporting metrics.



```python
import time
import sys
import collections
def lucas_sequence(n: int) -> list[int]:
    if not isinstance(n, int) or n < 0:
        raise ValueError("Input 'n' must be a non-negative integer.")
    if n == 0:
        return []
    if n == 1:
        return [2]
    if n == 2:
        return [2, 1]
    sequence = [2, 1]
    for _ in range(2, n):
        next_lucas = sequence[-1] + sequence[-2]
        sequence.append(next_lucas)
    return sequence
def run_lucas_sequence_with_metrics(n: int):
    start_time = time.perf_counter()
    lucas_nums = lucas_sequence(n)
    end_time = time.perf_counter()
    time_taken = (end_time - start_time) * 1000
    memory_used_bytes = sys.getsizeof(lucas_nums)
    steps = max(0, n - 2)
    print(f"--- Lucas Sequence (First {n} Numbers) ---")
    print(lucas_nums)
    print("\n--- Performance Metrics ---")
    print(f"N: {n} terms")
    print(f"Time Taken: {time_taken:.6f} milliseconds")
    print(f"Memory Used: {memory_used_bytes} bytes (Size of the output list)")
    print(f"No. of Steps (Main Loop Iterations): {steps}")
if __name__ == "__main__":
    run_lucas_sequence_with_metrics(n=10)
    print("\n" + "="*50 + "\n")
    run_lucas_sequence_with_metrics(n=20)
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/week 7 cse.py ==========
--- Lucas Sequence (First 10 Numbers) ---
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]

--- Performance Metrics ---
N: 10 terms
Time Taken: 0.010333 milliseconds
Memory Used: 184 bytes (Size of the output list)
No. of Steps (Main Loop Iterations): 8

==================================================

--- Lucas Sequence (First 20 Numbers) ---
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349]

--- Performance Metrics ---
N: 20 terms
Time Taken: 0.004959 milliseconds
Memory Used: 248 bytes (Size of the output list)
No. of Steps (Main Loop Iterations): 18
>>>
```

TITLE 27: Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as a^b where a > 0 and b > 1.

AIM/OBJECTIVE(s):

Aim: To develop an efficient algorithm to determine if a given positive integer n is a perfect power.

Objective: To confirm the existence of integers a > 0 and b > 1 such that n = a^b.

METHODOLOGY & TOOL USED:

Methodology:

Approach: Iterative Exponent Search (Root Checking).

Exclusion: Immediately return False for n \le 3 (as the smallest perfect power is 4 = 2^2).

Range Definition: Determine the maximum exponent to check, which is bounded by \lfloor \log_2 n \rfloor.

Iteration: Loop through potential exponents b starting from 2.

Base Calculation: For each b, calculate the potential base a by taking the root of n: a \approx n^{1/b}.

Verification: Round a to the nearest integer and check if \text{round}(a)^b is exactly equal to n.

Tool Used:

Python 3: Used for implementation, leveraging its native support for mathematical operations.

math Module: Specifically utilizes math.log2(n) to mathematically constrain the iteration range, ensuring high efficiency.

BRIEF DESCRIPTION:

Functionality: The is_perfect_power(n) function checks if a positive integer n can be written as a^b, where a>0 and b>1, without using standard Python math functions (pow, log, sqrt).

Algorithm: It linearly iterates through possible exponents b (from 2 up to 64). For each exponent, it uses a Binary Search approach to efficiently find the corresponding integer base a between 2 and n.

Performance Tracking: The script explicitly uses time and tracemalloc to measure the execution duration and peak memory usage.

RESULTS ACHIEVED:

Correctness: Accurately identifies perfect powers (e.g., 81 returns True, 3^4 or 9^2) and non-perfect powers.

Decomposition: If n is a perfect power, the function returns one valid base and exponent pair (a, b).

Metrics Output: Provides two quantitative metrics:

Execution Time (in seconds, e.g., 0.000005s).

Peak Memory Utilization (in KB, traced by tracemalloc).

DIFFICULTY FACED BY STUDENTS:

Replicating Built-ins: The primary difficulty is implementing core arithmetic operations, like exponentiation (_power_check), manually while handling implicit large number overflow or range checks.

Algorithmic Choice: Identifying and correctly implementing an efficient search mechanism (Binary Search) for the base a instead of a simple linear loop.

Debugging Logic: Ensuring the early exit conditions in _power_check (n // base < result) correctly prevent unnecessary large number calculations

SKILLS ACHIEVED:

Algorithmic Design: Practical implementation of the Binary Search algorithm for efficient number finding.

Constraint Programming: Developing solutions under strict limitations (no external math libraries).

Performance Analysis: Using time and tracemalloc for basic, mandatory performance profiling and optimization.

Mathematical Logic: Applying logarithmic bounding to determine a safe maximum iteration limit for the exponent b.

```python
import time
import tracemalloc


def _power_check(base, exp, n):
    """Calculates base^exp, stopping if result exceeds n."""
    result = 1
    for _ in range(exp):
        if base != 0 and n // base < result: return n + 1
        result *= base
        if result > n: return result
    return result

def is_perfect_power(n):
    """Checks if n = a^b where a > 0 and b > 1."""
    if n <= 1: return (n == 1, 1, 2 if n == 1 else 0)


    for b in range(2, 65):
        if _power_check(2, b, n) > n: break

  |
        low, high = 2, n
        while low <= high:
            mid = (low + high) // 2
            p = _power_check(mid, b, n)

            if p == n:
                return (True, mid, b)
            elif p < n:
                low = mid + 1
            else:
                high = mid - 1

    return (False, 0, 0)


TEST_N = 81

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

is_perfect, base, exponent = is_perfect_power(TEST_N)

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


print(f"Number: {TEST_N}")
if is_perfect:
    print(f"Result: True ({base}^{exponent})")
else:
    print("Result: False")

print(f"\nTime: {end_time - start_time:.6f}s")
print(f"Memory (Peak): {peak / 1024:.2f} KB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/week 7 cse.py =========
Number: 81
Result: True (9^2)

Time: 0.000040s
Memory (Peak): 0.88 KB
```

TITLE 28: 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/OBJECTIVE(s):

Aim:

To implement a computational function that determines the length of the sequence generated by the Collatz conjecture rules for any given positive integer.

Objective:

To develop a robust function that includes necessary input validation (ensuring the input is a positive integer).

To accurately apply the two-part iterative rule (if even, n/2; if odd, 3n+1).

To return the precise count of steps required for the starting number (n) to reach the terminal value (1).

METHODOLOGY & TOOL USED:

Methodology:

The implementation uses an Iterative Loop Algorithm that directly translates the Collatz conjecture rules:

Initialization: A step counter is set to zero, and a while loop is initiated to run until the working number (current) equals 1.

Parity Check: Inside the loop, the modulo operator checks if the number is even or odd.

Application of Rules:

If the number is even, it is divided by 2.

If the number is odd, it is multiplied by 3 and 1 is added.

Counting: The step counter is incremented after each transformation.

Termination: The loop concludes when 1 is reached, and the total count of steps is returned. This relies on the unproven assumption that all positive integers eventually reach 1.

Tool Used:

Tool: Python (Standard Python 3)

Rationale: Python was selected for its clean syntax, which simplifies the direct implementation of conditional and iterative logic. Furthermore, Python's native support for arbitrarily large integers is crucial for the Collatz sequence, as the intermediate numbers can often grow very large before eventually reducing back down to 1.

BRIEF DESCRIPTION:

This project involved implementing and testing the collatz_length(n) function, which computationally explores the famous Collatz conjecture. The conjecture states that any positive integer n will eventually reach 1 by repeatedly applying one of two rules: n/2 (if n is even) or 3n+1 (if n is odd).

The experiment's primary goal was to calculate the Collatz Length (the total number of steps) for a specific test case, N=27. Secondary goals included measuring the computational cost—specifically execution time and memory usage—to understand the performance characteristics of this simple yet complex mathematical sequence.

RESULTS ACHIEVED:

The Python script successfully executed the calculation for N=27, a number known for generating a relatively long sequence.

Starting Number (N): 27

Number of Steps (Collatz Length): 111

Execution Time: Approximately 0.04 milliseconds (highly efficient).

Memory Used: Minimal (28 bytes for both input and resulting steps).

Sequence Breakdown (First 15 Steps):

27 \rightarrow 82 \rightarrow 41 \rightarrow 124 \rightarrow 62 \rightarrow 31 \rightarrow 94 \rightarrow 47 \rightarrow 142 \rightarrow 71 \rightarrow 214 \rightarrow 107 \rightarrow 322 \rightarrow 161 \rightarrow 484 \rightarrow \dots (sequence is longer)

DIFFICULTY FACED BY STUDENTS:

The Collatz conjecture, despite its simple rules, presents several conceptual and practical hurdles for students:

Uncertain Termination: The largest challenge is that the conjecture is unproven. Students must write a while n != 1 loop, trusting that the

sequence will terminate, which relies on an unproven hypothesis rather than a mathematical guarantee.

Sequence Volatility: The sequence does not decrease consistently. The application of the 3n+1 rule causes massive spikes (e.g., 27 \right-arrow 82). Understanding why a number that jumps so high eventually returns to 1 can be counter-intuitive.

Data Type Management: In languages other than Python (which handles large integers automatically), running this sequence for large starting numbers would quickly lead to integer overflow errors, as the peak values can exceed the capacity of standard 64-bit integers.

Efficiency: For very large test ranges, simple iterative solutions become slow due to repeated calculations. Students face the challenge of learning to implement memoization (caching results) to improve performance.

SKILLS ACHIEVED:

Algorithmic Implementation: Successfully translating a mathematical rule set (the Collatz rules) into an iterative, functional program.

Conditional Logic and Control Flow: Proficient use of if/else statements for branching logic based on parity, and while loops for controlled iteration.

Input Validation and Exception Handling: Implementing protective measures (if n < 1: raise ValueError) to ensure the function only processes valid inputs.

Performance Analysis: Utilization of the time and sys modules to measure and report computational metrics (execution speed and memory footprint).

Modular Programming: Writing a clean, reusable function (collatz_length) with type hinting for clarity and maintainability.

```python
import time
import sys
import math
def collatz_length(n: int) -> int:
    if not isinstance(n, int) or n < 1:
        raise ValueError("Input must be a positive integer.")
    steps = 0
    while n != 1:
        steps += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return steps
if __name__ == "__main__":
    TEST_NUMBER = 27
    print(f"--- Running Collatz Sequence Length for N = {TEST_NUMBER} ---")
    start_time = time.perf_counter()
    try:
        collatz_result = collatz_length(TEST_NUMBER)
    except ValueError as e:
        collatz_result = f"Error: {e}"
    end_time = time.perf_counter()
    if isinstance(collatz_result, int):
        print(f"\n1. Number of Steps (Collatz Length): {collatz_result}")
    else:
        print(f"\n1. Number of Steps: N/A ({collatz_result})")
    elapsed_time_ms = (end_time - start_time) * 1000
    print(f"2. Time Taken (Execution Time): {elapsed_time_ms:.6f} milliseconds")
    input_memory = sys.getsizeof(TEST_NUMBER)
    output_memory = sys.getsizeof(collatz_result)
    print(f"\n3. Memory Used (Bytes):")
    print(f"   - Input Number ({TEST_NUMBER}): {input_memory} bytes")
    print(f"   - Resulting Steps ({collatz_result}): {output_memory} bytes")
    if TEST_NUMBER < 1000 and isinstance(collatz_result, int):
        print("\n--- Sequence Breakdown (First 15 steps) ---")
        n_current = TEST_NUMBER
        sequence = [n_current]
        step_count = 0
        while n_current != 1 and step_count < 15:
            step_count += 1
            if n_current % 2 == 0:
                n_current = n_current // 2
            else:
                n_current = 3 * n_current + 1
            sequence.append(n_current)
        sequence_str = " -> ".join(map(str, sequence))
        if collatz_result > 15:
            sequence_str += " -> ... (sequence is longer)"
        print(f"Sequence: {sequence_str}")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/week 7 cse.py =========
--- Running Collatz Sequence Length for N = 27 ---

1. Number of Steps (Collatz Length): 111
2. Time Taken (Execution Time): 0.012500 milliseconds

3. Memory Used (Bytes):
   - Input Number (27): 28 bytes
   - Resulting Steps (111): 28 bytes

--- Sequence Breakdown (First 15 steps) ---
Sequence: 27 -> 82 -> 41 -> 124 -> 62 -> 31 -> 94 -> 47 -> 142 -> 71 -> 214 -> 107 -> 322 -> 161 -> 484 -> 242 -> ... (sequence is longer)
```

TITLE 29: Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.

AIM/OBJECTIVE(s):

The primary aim of this task is to define and implement a robust computational function, polygonal_number(s, n), capable of generating any term in any sequence of polygonal numbers

METHODOLOGY & TOOL USED:

Methodology:

1. Function Definition: A function named polygonal_number(s, n) was created to accept two mandatory integer arguments: s (sides) and n (position).
2. Input Validation: The implementation includes checks to ensure s is at least 3 (triangular numbers are the minimum) and n is at least 1 (the position must be positive). A ValueError is raised for invalid inputs.
3. Calculation: The terms of the mathematical formula are calculated sequentially

Tool Used:

Programming Language : Python Selected for its clear syntax, strong support for integer arithmetic, and suitability for mathematical functions and scripting.

Development Style : Procedural Function

BRIEF DESCRIPTION: The Python script defines a function, polygonal_number(s, n), which calculates the n-th number in the sequence of s-gonal numbers. The core function is based on the closed-form algebraic formula. The script's primary purpose beyond calculation is to serve as a performance benchmark. It leverages Python's built-in time and tracemalloc

modules to measure the execution time and memory consumption (peak usage) of the calculation, providing real-world metrics on the efficiency of the mathematical solution

RESULTS ACHIEVED:

1. Correct Calculation: It computes the correct n-th s-gonal number, P(s, n), in \mathcal{O}(1) time complexity, as it involves only a fixed number of basic arithmetic operations regardless of the size of n or s.
2. Performance Metrics: It accurately measures and reports:
3. Execution Time: The time taken to run the calculation, typically very close to zero seconds for this simple operation.

DIFFICULTY FACED BY STUDENTS:

While the code itself is mathematically straightforward, students may face difficulty with the following concepts and practices:

1. Mathematical Derivation: Understanding why the formula P(s, n) =\frac{n^2(s-2) - n(s-4)}{2} works, rather than just plugging in the values.

2. Integer Division (//): Grasping why integer division is used and its importance in ensuring the result remains an integer, especially when standard division (/) might produce a float.

SKILLS ACHIEVED:

1. Mathematical Derivation: Understanding why the formula P(s, n) =\frac{n^2(s-2) - n(s-4)}{2} works, rather than just plugging in the values.
2. Integer Division (//): Grasping why integer division is used and its importance in ensuring the result remains an integer, especially when standard division (/) might produce a float.

```python
import time
import tracemalloc
import sys
def polygonal_number(s, n):
    return ((s - 2) * n * n - (s - 4) * n) // 2
if __name__ == "__main__":
    examples = [
        (3, 5, "Triangular"),
        (4, 5, "Square"),
        (5, 5, "Pentagonal")
    ]
    tracemalloc.start()
    start_time = time.time()
    steps = 0
    results = []
    for s, n, shape in examples:
        result = polygonal_number(s, n)
        results.append((shape, s, n, result))
        steps += 1
    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"--- Polygonal Number Calculation (n=5) ---")
    for shape, s, n, result in results:
        print(f"{shape} Number (s={s}): {result}")
    print("\n--- Performance Metrics ---")
    print(f"Total Function Calls (Steps): {steps}")
    print(f"Execution Time: {(end_time - start_time):.10f} seconds")
    print(f"Memory Usage: {current / 1024:.4f} KB")
    print(f"Peak Memory Usage: {peak / 1024:.4f} KB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/week 7 cse.py =========
--- Polygonal Number Calculation (n=5) ---
Triangular Number (s=3): 15
Square Number (s=4): 25
Pentagonal Number (s=5): 35

--- Performance Metrics ---
Total Function Calls (Steps): 3
Execution Time: 0.0000188351 seconds
Memory Usage: 0.9219 KB
Peak Memory Usage: 0.9688 KB
>>>
```

TITLE 30: 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/OBJECTIVE(s):

Aim:

This code aims to identify whether a given composite number is a Carmichael number—a special class of numbers in number theory that are composite yet satisfy Fermat's Little Theorem for all bases coprime to them, making them false positives in certain primality tests and important in the study of modular arithmetic and cryptographic security.

Objective:

1. Prime Number Verification: To accurately distinguish between prime and composite numbers using an efficient primality testing algorithm.
2. Carmichael Number Detection: To identify Carmichael numbers by verifying if they satisfy the condition a^(n-1) ≡ 1 mod n for all integers a that are coprime to n.
3. Computational Efficiency: To implement optimized algorithms for modular exponentiation and GCD calculation that ensure reasonable performance even for larger numbers.

METHODOLOGY & TOOL USED:

Methodology: This code employs a systematic methodology to identify Carmichael numbers by first checking if the input number is composite using an optimized primality test. For composite numbers, it then verifies the Carmichael criterion by iterating through all possible bases, using modular exponentiation to efficiently test whether each base coprime to the number satisfies Fermat's Little Theorem condition, confirming the number as Carmichael only if all tests pass.

Tool Used:

1. Python Programming Language: The entire algorithm is implemented using Python, leveraging its syntax for loops, conditional statements, variable assignments, and arithmetic operations.

2. Standard Library Modules: The code utilizes Python's time module specifically to measure and analyze the execution time of the algorithm, providing performance metrics.

3. Mathematical Algorithms: The core tool is the implementation of the Extended Euclidean Algorithm, a fundamental number-theoretic procedure used for finding modular inverses and solving linear Diophantine equations.

BRIEF DESCRIPTION: This Python code implements a comprehensive algorithm to detect Carmichael numbers - special composite numbers that satisfy Fermat's Little Theorem for all bases coprime to them. The code combines multiple mathematical approaches: it first checks if a number is composite using an optimized primality test, then verifies the Carmichael property by testing whether a^(n-1) ≡ 1 mod n holds true for all integers 'a' that are coprime to 'n'. The implementation efficiently handles large number computations through modular exponentiation and GCD algorithms while measuring execution performance, making it useful for number theory analysis and cryptographic applications where such numbers have significant importance.

RESULTS ACHIEVED: This code systematically identifies Carmichael numbers through mathematical verification. When testing a number like 561, it first confirms it is composite using its primality test. Then it verifies that for every integer base coprime to 561, the modular exponentiation condition a^(560) mod 561 equals 1. The successful validation leads to the output: "561 is a Carmichael number." The code also measures and displays the execution time required for this comprehensive verification process.

DIFFICULTY FACED BY STUDENTS:

1. Understanding Carmichael Number Concept: Students struggle to grasp why composite numbers can satisfy Fermat's Little Theorem, making the fundamental logic behind Carmichael numbers challenging.
2. Implementing Modular Exponentiation: Correctly coding the efficient "exponentiation by squaring" algorithm with proper modulus operations at each step proves difficult for many learners.
3. Optimizing the Verification Loop: Students find it challenging to implement the comprehensive coprime check efficiently, often creating slow algorithms that test unnecessary bases or handle GCD calculations incorrectly.

SKILLS ACHIEVED:

1. Mathematical Algorithm Implementation: The code demonstrates skill in translating complex number theory concepts (Carmichael numbers, Fermat's theorem) into functional programming logic.)
2. Optimized Computational Methods: It shows proficiency in implementing efficient algorithms including modular exponentiation, GCD calculation, and optimized primality testing for handling large numbers.
3. Comprehensive Condition Verification: The code exhibits ability to systematically verify multiple mathematical conditions (compositeness, coprimality, modular congruence) to solve a sophisticated numerical classification problem.

```python
import time
def power_with_modulo(base, exp, mod):
    result = 1
    base %= mod
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
def is_carmichael(n):
    if is_prime(n):
        return False
    for a in range(2, n):
        if gcd(a, n) == 1:
            if power_with_modulo(a, n - 1, n) != 1:
                return False
    return True
number_to_check = 561
start_time = time.time()
if is_carmichael(number_to_check):
    print(f"{number_to_check} is a Carmichael number.")
else:
    print(f"{number_to_check} is not a Carmichael number (or is prime).")
end_time = time.time()
execution_time = end_time - start_time
print(f"Time of execution: {execution_time:.6f} seconds")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/week 7 cse.py =========
561 is a Carmichael number.
Time of execution: 0.037785 seconds
```

Practical No: 7

TITLE 31: implement the probabilistic miller-raibin test is_prime_miller_rabin(n,k) with k rounds.

AIM/OBJECTIVE(s):

Aim:

To probabilistically determine if a given large integer n is a prime number with a high degree of confidence.

Objective:

Implement the Miller-Rabin algorithm to test primality over k rounds, where the probability of a composite number passing the test is less than (1/4)^k.

METHODOLOGY & TOOL USED:

Methodology:

Decompose n-1 into 2^r \cdot d, where d is odd.

Select k random bases a (witnesses).

For each a, check the primality conditions: a^d \equiv 1 \pmod n or a^{2^j d} \equiv -1 \pmod n for 0 \le j < r.

Tool Used:

Python programming language, utilizing standard libraries for random number generation and modular exponentiation.

BRIEF DESCRIPTION:

Core Function: Implements the probabilistic Miller-Rabin primality test to determine if a large integer N is likely prime.

Constraint Adherence: Avoids using standard Python libraries (math, random), requiring manual implementation of core functionality.

Custom Tools: Includes a custom Linear Congruential Generator (Rnd) for pseudo-random witness selection and a manual function (pmod) for fast modular exponentiation.

Output: Provides the primality result and measures performance (execution time and peak memory usage).

RESULTS ACHIEVED:

Primality Check: Successfully determines if the provided number N (e.g., 10^9 + 7) is a probable prime after K=20 rounds.

Execution Time: Measures the precise runtime of the entire process, typically in milliseconds, using time.time().

Memory Utilization: Accurately captures the peak memory footprint (in MB) during execution using tracemalloc

DIFFICULTY FACED BY STUDENTS:

Mathematical Foundation: Understanding the decomposition of N-1 into 2^r \cdot d and the four conditions of the Miller-Rabin test.

Algorithm Efficiency: Manually implementing the binary exponentiation (square and multiply) algorithm in pmod efficiently.

Simulating Randomness: Creating a simple, reliable pseudo-random number generator (LCG) without external libraries.

Metric Integration: Correctly initiating and concluding the tracemalloc and time measurement around the target function call.

SKILLS ACHIEVED:

Computational Number Theory: Proficient application of modular arithmetic and properties of prime numbers.

Core Algorithm Development: Ability to translate advanced mathematical algorithms into functional code.

Performance Engineering: Practical experience using Python tools (time, tracemalloc) for resource profiling.

Abstraction and Encapsulation: Creating modular classes/functions (Rnd, pmod, test) to manage complexity and adhere to constraints.

```python
import time
import tracemalloc

class Rnd:
    """Minimal LCG for pseudo-random number generation."""
    def __init__(self):

        self.s = int(time.time() * 1000)
        self.a, self.c, self.m = 1103515245, 12345, 2**31 - 1

    def next(self, min_val, max_val):
        """Generates an integer in [min_val, max_val]."""
        self.s = (self.a * self.s + self.c) % self.m
        return min_val + (self.s % (max_val - min_val + 1))

def pmod(b, e, m):
    """Computes (b^e) % m using binary exponentiation."""
    r, b = 1, b % m
    while e > 0:
        if e & 1: r = (r * b) % m
        e //= 2
        b = (b * b) % m
    return r

def test(n, d, r, rng):
    """Core Miller-Rabin test for a single witness."""
    a = rng.next(2, n - 2)
    x = pmod(a, d, n)

    if x == 1 or x == n - 1: return True

    for _ in range(r - 1):
        x = (x * x) % n
        if x == n - 1: return True
        if x == 1: return False

    return False
def is_prime_mr(n, k=20):
    """Miller-Rabin primality test with k rounds."""
    if n <= 3: return n > 1
    if n % 2 == 0 or n == 4: return False

    d, r = n - 1, 0
    while d % 2 == 0:
        d //= 2
        r += 1

    rng = Rnd()
    for _ in range(k):
        if not test(n, d, r, rng): return False

    return True


N = 1000000007
K = 20
tracemalloc.start()
start_time = time.time()
is_p = is_prime_mr(N, K)

end_time = time.time()
curr, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"Miller-Rabin Test (k={K})")
print(f"Number: {N}")
print(f"Result: {'PROBABLY PRIME' if is_p else 'COMPOSITE'}")
print(f"Execution Time: {end_time - start_time:.6f} seconds")
print(f"Peak Memory: {peak / 10**6:.3f} MB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 8.py =========
Miller-Rabin Test (k=20)
Number: 1000000007
Result: PROBABLY PRIME
--------------------------------
Execution Time: 0.001252 seconds
Peak Memory: 0.001 MB
```

TITLE 32: Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.

AIM/OBJECTIVE(s):Aim: Implement an efficient, probabilistic algorithm for integer factorization.

Objective: Find a non-trivial factor of a large composite number n using the Pollard's Rho method.

METHODOLOGY & TOOL USED:

Methodology:

Sequence Generation: Create a pseudo-random sequence using the function f(x) = (x^2 + c) \pmod n.

Cycle Detection: Employ Floyd's Tortoise and Hare algorithm to detect the cycle closure within the sequence modulo a prime factor p.

Factor Discovery: The factor g is found by calculating the Greatest Common Divisor (GCD) of the difference between the two pointers and n: g = \text{gcd}(|x - y|, n).

Efficiency: This method is effective when n has a relatively small prime factor.

Tool Used:

Language: Python 3.

Key Modules:

math.gcd: Essential for the core factor extraction step.

random: Used to initialize the starting value (x) and the constant (c) for the polynomial.

BRIEF DESCRIPTION:

Algorithm: Implements Pollard's Rho, a probabilistic integer factorization algorithm, focusing on the detection of cycles in the sequence x_{i+1} = (x_i^2 + c) \pmod n.

Core Logic: Utilizes Floyd's cycle-finding algorithm (Tortoise and Hare method) to efficiently find the point where the sequence, when taken modulo a prime factor p of n, begins to cycle.

Constraint Adherence: Strictly avoids all standard math libraries, requiring manual implementation of the Euclidean algorithm for GCD and a Linear Congruential Generator (LCG) for pseudo-random number generation.

RESULTS ACHIEVED:

Successful Factorization: Found non-trivial factors for composite inputs (e.g., 8051, 187) within a single execution cycle.

Benchmarking Data: Provides quantitative performance metrics including Execution Time (in seconds) and Peak Memory Usage (in KiB) , demonstrating the algorithm's resource profile for different input sizes.

DIFFICULTY FACED BY STUDENTS:

Theoretical Grasp: Understanding the connection between cycle closure modulo p and the GCD calculation (i.e., why \text{gcd}(|x - y|, n) > 1 yields a factor).

Manual Primitive Implementation: Correctly and robustly implementing fundamental functions like the Euclidean algorithm (GCD) and a basic PRNG from scratch, which are typically provided by standard libraries.

Handling Failure: Recognizing and managing the probabilistic nature of the algorithm, where the chosen random constants (x and c) might fail to find a factor or wrongly identify a prime number.

SKILLS ACHIEVED:

Advanced Number Theory: Practical application and implementation of the Pollard's Rho factorization algorithm.

Algorithmic Mastery: Proficient use of Floyd's Cycle-Finding Algorithm for sequence analysis.

Low-Level Coding: Development of core mathematical utilities (GCD, LCG) demonstrating a deeper understanding of modular arithmetic and computational fundamentals.

Performance Analysis: Skill in benchmarking code execution time and memory footprint using the time.perf_counter() and tracemalloc modules.

```python
import time
import tracemalloc

A = 1103515245
C = 12345
M = 2**31
_seed = int(time.time() * 1000)

def next_random(a, b):
    global _seed
    _seed = (A * _seed + C) % M
    return (_seed % (b - a + 1)) + a

def manual_gcd(a, b):
    a = a if a >= 0 else -a
    b = b if b >= 0 else -b
    while b != 0:
        a, b = b, a % b
    return a
def pollard_rho_manual(n):
    if n <= 1: return n
    if n % 2 == 0: return 2

    x = next_random(2, n - 1)
    y = x
    c = next_random(1, n - 1)
    g = 1

    def f(val):
        return (val * val + c) % n

    while g == 1:
        x = f(x)
        y = f(f(y))

        diff = x - y
        if diff < 0: diff = -diff

        g = manual_gcd(diff, n)

        if g > 1:
            return g if g != n else n

        if x == y:
            return n
def run_benchmark(number):
    """Executes pollard_rho and measures time/memory."""

    tracemalloc.start()
    start_time = time.perf_counter()

    factor = pollard_rho_manual(number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    elapsed_time = end_time - start_time

    print(f"\n--- FACTORING N={number} ---")
    print(f"Factor Found: {factor}")
    print(f"Execution Time: {elapsed_time:.6f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

# Example numbers (8051 = 83 * 97)
run_benchmark(8051)
run_benchmark(187)
run_benchmark(99999901)
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 8.py ==========

--- FACTORING N=8051 ---
Factor Found: 83
Execution Time: 0.000119 seconds
Peak Memory Usage: 0.48 KiB

--- FACTORING N=187 ---
Factor Found: 17
Execution Time: 0.000013 seconds
Peak Memory Usage: 0.33 KiB

--- FACTORING N=99999901 ---
Factor Found: 151
Execution Time: 0.000043 seconds
Peak Memory Usage: 0.49 KiB
```

TITLE 33: Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.

AIM/OBJECTIVE(s):

Aim: Implement a function for numerical approximation of the Riemann zeta function, \zeta(s).

Objective: Calculate the partial sum of the defining Dirichlet series \sum_{n=1}^{\infty} \frac{1}{n^s} up to a user-specified number of terms (T).

METHODOLOGY & TOOL USED:

Methodology:

Series Truncation: The infinite series is truncated to T terms, \sum_{n=1}^{T} \frac{1}{n^s}, to provide a finite, computable approximation.

Direct Iteration: A simple loop iterates through integers n=1 to T.

Summation: In each iteration, the value \frac{1}{n^s} is calculated and accumulated into a running total.

Tool Used:

Tool: Python programming language.

Core Feature Used: Basic arithmetic operators and the built-in power operator (**) for calculating n^s.

BRIEF DESCRIPTION:

Function: Implements an approximation of the Riemann zeta function \zeta(s) by summing the first T terms of the defining Dirichlet series \sum_{n=1}^{T} \frac{1}{n^s}.

Constraint Adherence: Avoids all external math libraries and built-in power operators (** or pow()), relying on a custom, loop-based power function (manual_power).

Measurement: Integrates time and tracemalloc to report real-world execution time and peak memory utilization.

RESULTS ACHIEVED:

Output: Provides a numerical approximation for \zeta(3) (using 50,000 terms).

Performance Metrics: Successfully outputs the computation's execution time (in seconds) and memory footprint (in KiB).

DIFFICULTY FACED BY STUDENTS:

Constraint Limitation: The primary difficulty is implementing n^s without standard tools, which forces the exponent (s) to be a positive integer, thus limiting the mathematical scope of the function.

Performance Overhead: The manual_power function is significantly slower and less efficient than the built-in ** operator, leading to longer execution times for high values of T.

Convergence Speed: The required series converges slowly, necessitating a very large number of terms (50,000) to achieve reasonable accuracy, which exacerbates the performance issues.

SKILLS ACHIEVED:

Fundamental Algorithms: Mastery of iterative numerical summation and the low-level implementation of core mathematical functions (exponentiation).

Performance Analysis: Practical experience with Python's time and tracemalloc to quantify computational cost and memory use.

Constraint Programming: Ability to solve a problem under severe restrictions, requiring the student to think critically about language primitives.

Mathematical Context: Understanding the defining series of the Riemann zeta function and its constraints (convergence only for \text{Re}(s) > 1).

```python
import time
import tracemalloc

def manual_power(base: float, exponent: int) -> float:
    """Calculates base^exponent for positive integer exponents."""
    if exponent == 0: return 1.0
    if exponent < 0: return 0.0

    result = 1.0
    for _ in range(exponent):
        result *= base
    return result

def zeta_approx(s: int, terms: int) -> float:
    """
    Approximates ζ(s) using the first 'terms' of the series
    sum_{n=1}^{T} (1 / n^s). s must be an integer > 1.
    """
    if s <= 1 or terms <= 0:
        return 0.0

    approximation = 0.0
    for n in range(1, terms + 1):
        approximation += 1.0 / manual_power(float(n), s)

    return approximation

if __name__ == '__main__':
    S_VALUE = 3
    TERMS = 50000

    print(f"--- ζ({S_VALUE}) Approx with {TERMS} terms ---")

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

    result = zeta_approx(S_VALUE, TERMS)

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

    print(f"Result: {result:.10f}")
    print(f"Time: {end_time - start_time:.6f}s")
    print(f"Peak Memory: {peak / 1024:.2f} KiB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.

========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 8.py =========
--- ζ(3) Approx with 50000 terms ---
Result: 1.2020569030
Time: 0.033651s
Peak Memory: 0.16 KiB
```

TITLE 34: Write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers.

AIM/OBJECTIVE(s):

To implement a robust and efficient computational solution for calculating the Partition Function, p(n), a fundamental problem in combinatorics and number theory.

METHODOLOGY & TOOL USED:

Methodology:

The chosen methodology avoids the high computational cost of pure recursion and the complexity of implementing Euler's Pentagonal Number Theorem. The core idea is to iteratively build up the count of partitions by considering increasingly larger allowable part sizes:

Initialization: An array (or list), dp, is created, where dp[i] will store the final count of partitions for the integer i. dp[0] is initialized to 1, representing the single (empty) partition of 0.

Tool Used:

Aspect Details

Language Python (Standard Library)

Data Structure List/Array (for the DP table) Concepts Dynamic Programming (DP)

BRIEF DESCRIPTION:

The provided Python code calculates the Partition Function, p(n), which gives the number of ways a positive integer n can be expressed as a sum of positive integers (order invariant).

Crucially, this implementation utilizes Euler's Pentagonal Number Theorem (PNT). This theorem provides a powerful, alternating series recurrence relation for p(n):

Where g k are the generalized pentagonal numbers, given by g k = \frac{k (3k \pm 1)}{2} for k = 1, 2, 3, \dots.

The code employs dynamic programming (DP) by calculating p (0), p (1), \dots, p(n) sequentially. For each p(k), it efficiently sums the contributions from previous values p (k – g m), incorporating the alternating sign from the theorem. This method significantly reduces the time complexity from the simple O(n^2) DP approach to approximately O(n\sqrt{n}), making it highly efficient for larger inputs.

RESULTS ACHIEVED:

1. Partition Count (p(n)): The final, correct count of distinct partitions for the input n.
2. Execution Time: Provides a precise measurement of the algorithm's speed (in seconds), demonstrating the computational efficiency of the PNT recurrence.
3. Memory Used: Reports the peak memory consumption (in KB), primarily used for storing the DP array p, which must hold n+1 integer values.

DIFFCULTY FACED BY STUDENTS:

While the code itself is mathematically straightforward, students may face difficulty with the following concepts and practices:

Mathematical Derivation: Understanding why the formula P(s, n) =\frac{n^2(s-2) - n(s-4)}{2} works, rather than just plugging in the values.

Integer Division (//): Grasping why integer division is used and its importance in ensuring the result remains an integer, especially when standard division (/) might produce a float.

SKILLS ACHIEVED:

1. Advanced Dynamic Programming: Moving beyond simple linear DP to solving problems using complex, number-theoretic recurrence relations.
2. Algorithmic Optimization: Understanding how to select a superior algorithm (PNT, O(n\sqrt{n})) over a simpler one (Basic DP, O(n^2)) for performance gains.
3. Mathematical Implementation: Translating a sophisticated mathematical theorem into clean, executable code.

```python
import time
import tracemalloc

def partition_function(n):
    p = [0] * (n + 1)
    p[0] = 1
    for k in range(1, n + 1):
        total = 0
        m = 1
        while True:
            g1 = m * (3*m - 1) // 2
            g2 = m * (3*m + 1) // 2
            if g1 > k:
                break
            sign = -1 if (m % 2 == 0) else 1
            total += sign * p[k - g1]

            if g2 <= k:
                total += sign * p[k - g2]
            m += 1
        p[k] = total
    return p[n]
n = int(input("Enter n: "))

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

result = partition_function(n)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
exec_time = time.time() - start
print(f"\np({n}) = {result}")
print(f"Execution Time: {exec_time:.6f} seconds")
print(f"Memory Used: {peak/1024:.2f} KB")
```

```
Python 3.13.7 (v3.13.7:bcee1c32211, Aug 14 2025, 19:10:51) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>>
========== RESTART: /Users/jaydeepsinghrathore/Documents/cse week 8.py =========
Enter n: 56

p(56) = 526823
Execution Time: 0.000752 seconds
Memory Used: 1.80 KB
>>>
```