



# Lab Manual

## Practical and Skills Development

---

# CERTIFICATE

---

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN  
SATISFACTORILY PERFORMED BY

**Registration No** : 25BCE11365  
**Name of Student** : Divyanshu 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:

### Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	factorial(n)	27-09-25	
2	is_palindrome(n)	27-09-25	
3	mean_of_digits(n)	27-09-25	
4	digital_root(n)	27-09-25	
5	is_abundant(n)	27-09-25	
6	is_deficient(n)	27-09-25	
7	is_harshad(n)	27-09-25	
8	is_automorphic(n)	04-10-25	
9	is_pronic(n)	04-10-25	
10	prime_factors(n)	04-10-25	
11	count_distinct_prime_factors(n)	04-10-25	
12	is_prime_power(n)	09-11-25	
13	is_mersenne_prime(p)	09-11-25	
14	twin_primes(limit)	09-11-25	
15	count_divisors(n)	09-11-25	

### Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
16	aliquot_sum(n)	16-11-25	
17	are_amicable(a, b)	16-11-25	
18	multiplicative_persistence(n)	16-11-25	
19	is_highly_composite(n)	16-11-25	
20	mod_exp(base, exponent, modulus)	16-11-25	
21	mod_inverse(a, m)	16-11-25	
22	crt(remainders, moduli)	16-11-25	
23	is_quadratic_residue(a, p)	16-11-25	
24	order_mod(a, n)	16-11-25	
25	is_fibonacci_prime(n)	16-11-25	
26	Lucas Sequence Generator	16-11-25	
27	Algorithmic Determination of Perfect Powers	16-11-25	
28	Analysis of Collatz Sequence Length	16-11-25	
29	Polygonal Number Calculation	16-11-25	
30	Carmichael Number Verification System	16-11-25	

**Practical Index**

<b>S. No.</b>	<b>Title of Practical</b>	<b>Date of Submission</b>	<b>Signature of Faculty</b>
<b>31</b>	Miller-Rabin test	16-11-25	
<b>32</b>	Pollard's rho algorithm	16-11-25	
<b>33</b>	Riemann zeta function	16-11-25	
<b>34</b>	Partition Function	16-11-25	

**Practical No: 01**

**Date:** 27-09-2025

**TITLE:** Factorial of a Number

**AIM/OBJECTIVE(s):** The aim is to implement the function factorial n that computes n factorial which is the product of all positive integers up to n.

**METHODOLOGY & TOOL USED:** An iterative approach was used where the product is built step by step from one to n. Python was used for its straightforward loop structure and reliable handling of large integers.

**BRIEF DESCRIPTION:** The factorial of n increases rapidly and is useful in combinatorics and probability. The function multiplies sequential integers and returns the final product for any non-negative input.

**RESULTS ACHIEVED:** The implementation produced correct values for all tested inputs including large values that exceed standard data types.

**DIFFICULTY FACED BY STUDENT:** Students occasionally struggled with understanding large integer growth and ensuring the loop included all required values.

**SKILLS ACHIEVED:** Students learned iterative computation, precise control of loops and handling of high value outputs.

**Practical No: 02**

**Date:** 27-09-2025

**TITLE:** Palindrome Check

**AIM/OBJECTIVE(s):** The aim is to implement the function is palindrome n which determines whether a given integer reads the same forward and backward. The objective is to understand simple numerical string manipulation and logical checks used in algorithmic verification.

**METHODOLOGY & TOOL USED:** The method converts the integer into a string and compares it with its reversed form. If both match, the number is identified as a palindrome. Python was used because it allows easy string conversion and provides direct slicing operations for reversal.

**BRIEF DESCRIPTION:** A palindromic number maintains the same sequence of digits when read in both directions. The implemented function performs a direct comparison between the original representation and its reversed version. This gives a quick and reliable test for palindromicity.

**RESULTS ACHIEVED:** The function correctly identified palindromic and non palindromic numbers across various test inputs. It performed efficiently even for large integers due to the simplicity of the operations involved.

**DIFFICULTY FACED BY STUDENT:** Students sometimes struggled with converting between integer and string types and with understanding how slicing operations work. Handling negative numbers and ensuring that only the absolute digit pattern was examined also required some care.

**SKILLS ACHIEVED:** Students strengthened their understanding of string processing, conditional logic, and simple numerical validation. They also gained practice in designing clear and concise functions that perform a single well defined task.

**Practical No: 03****Date:** 27-09-2025**TITLE:** Average of all digits of a number

**AIM/OBJECTIVE(s):** The aim is to implement the function mean of digits n which calculates the average value of all individual digits in the given number. The objective is to practice numerical decomposition and basic arithmetic operations within a programming context.

**METHODOLOGY & TOOL USED:** : The method converts the number into a string so that each digit can be accessed easily. Each character is then converted back to an integer and added to a running total. The sum is divided by the count of digits to obtain the mean. Python was used because it supports simple type conversion and straightforward iteration over characters.

**BRIEF DESCRIPTION:** The function extracts every digit from the input number and computes their average. This is done by collecting the digits in sequence and applying the arithmetic mean formula. The result gives a measure of the typical value of the digits present in the number.

**RESULTS ACHIEVED:** The implementation produced correct averages for a wide range of test values including single digit and multi digit numbers. The function executed quickly due to the low computational complexity of the operations involved.

**DIFFICULTY FACED BY STUDENT:** Students occasionally experienced issues with converting between string and integer types and with understanding that each character in the string corresponds to one digit.



Handling negative values also required careful consideration to avoid including the sign in the digit count.

**SKILLS ACHIEVED:** Students strengthened their understanding of type conversion, string manipulation, digit extraction, and basic arithmetic computation. They also improved their ability to write clean modular functions that perform simple numerical processing tasks.

**Practical No: 04****Date:** 27-09-2025**TITLE:** Repeated Sum

**AIM/OBJECTIVE(s):** The aim is to implement the function digital root n which repeatedly sums the digits of a number until only one digit remains. The objective is to understand iterative digit processing and the idea of reducing a number to its fundamental single digit value.

**METHODOLOGY & TOOL USED:** The method converts the number into a string, extracts each digit, and sums them. If the result has more than one digit the process repeats until a single digit is produced. Python was used because it makes string manipulation and repeated loops simple and reliable.

**BRIEF DESCRIPTION:** The digital root is obtained by adding all digits of a number and continuing the summation on the resulting values until a single digit is reached. This function automates that reduction process using repeated digit extraction and summation.

**RESULTS ACHIEVED:** The implementation correctly produced the digital root for all tested inputs including large values. The iterative process completed quickly because each repeated step reduces the number of digits significantly.

**DIFFICULTY FACED BY STUDENT:** Students sometimes found it challenging to manage the loop that continues until one digit remains. Ensuring that the sign of negative numbers is ignored and handling input conversion without errors also required attention

**SKILLS ACHIEVED:** Students improved their understanding of loops, digit processing, recursive style logic, and number reduction techniques. They also gained confidence in writing functions that apply repeated transformations to reach a stable final value.

**Practical No: 05**

**Date:** 27-09-2025

**TITLE:** Function to Identify Abundant Numbers

**AIM/OBJECTIVE(s):** The aim is to implement the function is abundant n which determines whether the sum of all proper divisors of n is greater than the number itself. The objective is to understand divisor analysis and the classification of integers based on their divisor sums.

**METHODOLOGY & TOOL USED:** The method finds all proper divisors by iterating from one up to the square root of n and collecting divisors in pairs. These values are summed and compared with n to check if the number qualifies as abundant. Python was used because it offers simple looping structures and efficient integer operations.

**BRIEF DESCRIPTION:** A number is considered abundant when the total of its proper divisors exceeds its own value. The implemented function evaluates the divisor set of n, computes their sum, and determines whether the excess condition is satisfied.

**RESULTS ACHIEVED:** The function successfully identified abundant numbers and non abundant numbers across a variety of inputs. It produced results aligned with known examples such as twelve and twenty four. The approach worked efficiently even for moderately large values.

**DIFFICULTY FACED BY STUDENT:** Students sometimes had difficulty identifying proper divisors and ensuring that n itself was not included in the sum. Handling divisor pairs correctly and avoiding repeated counting also required careful implementation.

**SKILLS ACHIEVED:** Students gained experience in divisor computation, conditional evaluation, and mathematical classification of integers. They also strengthened their ability to write structured functions that analyse numerical properties systematically.

**Practical No: 06****Date:** 27-09-2025**TITLE:** Function to Identify Deficient Numbers

**AIM/OBJECTIVE(s):** The aim is to implement the function is deficient n which determines whether the sum of all proper divisors of n is less than the number itself. The objective is to understand divisor based classification of integers and the concept of deficiency.

**METHODOLOGY & TOOL USED:** The method collects proper divisors by checking every integer up to the square root of n and adding both divisors in each valid pair. These divisors are summed and the result is compared with n. Python was used because it provides simple looping structures and efficient integer operations for divisor detection.

**BRIEF DESCRIPTION:** A number is considered deficient when the sum of its proper divisors is smaller than the number. The implemented function calculates this sum and checks the deficiency condition. This allows quick classification of any input value.

**RESULTS ACHIEVED:** The function correctly identified deficient numbers and matched known values such as eight and fourteen. It worked efficiently on a wide range of inputs including larger integers.

**DIFFICULTY FACED BY STUDENT:** Students sometimes struggled with identifying proper divisors accurately and ensuring that the number n itself was excluded from the divisor list. Careful handling of divisor pairs and avoiding double addition required attention.

**SKILLS ACHIEVED:** Students developed skills in divisor analysis, conditional numerical evaluation, and careful implementation of mathematical definitions in code. They also improved their understanding of number classes based on divisor sums.

**Practical No: 07****Date:** 04-10-2025**TITLE:** Harshad Numbers

**AIM/OBJECTIVE(s):** The aim is to implement the function is harshad n which checks whether a number is divisible by the sum of its digits. The objective is to understand digit manipulation and divisibility rules in programming.

**METHODOLOGY & TOOL USED: :** The method converts the number into its digits, sums them, and checks whether the original number is divisible by this sum. Python was used because it allows easy conversion of integers to strings and simple arithmetic operations on digits.

**BRIEF DESCRIPTION:** A Harshad number is one that is divisible by the sum of its digits. The function extracts each digit, computes the total sum, and tests divisibility. It returns True if the number meets the Harshad condition and False otherwise.

**RESULTS ACHIEVED:** The implementation successfully identified Harshad numbers across a variety of test inputs. It produced results quickly and accurately even for larger numbers.

**DIFFICULTY FACED BY STUDENT:** Students sometimes found it challenging to extract digits correctly and ensure that the sum computation excluded errors. Handling zero digits and avoiding division errors required attention.



**SKILLS ACHIEVED:** Students gained experience in digit extraction, arithmetic computation, and applying divisibility rules in programming. They also strengthened their ability to implement clear and efficient logical checks on numerical properties.

**Practical No: 08****Date:** 04-10-2025**TITLE:** Automorphic Numbers

**AIM/OBJECTIVE(s):** The aim is to implement the function is automorphic n which checks whether the square of a number ends with the number itself. The objective is to understand numeric pattern recognition and string or digit manipulation in programming.

**METHODOLOGY & TOOL USED:** The method computes the square of the number, converts both the number and its square to strings, and checks whether the square ends with the original number. Python was used because string slicing and conversion operations are straightforward and efficient.

**BRIEF DESCRIPTION:** An automorphic number has the property that its square ends in the same digits as the number itself. The function performs this check by comparing the last digits of the square with the original number and returns True if they match and False otherwise.

**RESULTS ACHIEVED:** The implementation successfully identified automorphic numbers such as five and twenty five. It worked correctly for a variety of test inputs and completed the check quickly due to simple arithmetic and string operations.

**DIFFICULTY FACED BY STUDENT:** Students sometimes struggled with converting between numeric and string forms and with performing the correct slicing of the square to compare with the original number. Understanding the automorphic property conceptually also required careful attention.

**SKILLS ACHIEVED:** Students gained experience in pattern recognition, string manipulation, and numeric checks. They also improved their ability to translate mathematical properties into efficient computational tests.

**Practical No: 09****Date:** 04-09-2025**TITLE:** Pronic Numbers

**AIM/OBJECTIVE(s):** The aim is to implement the function is pronic which checks whether a number can be expressed as the product of two consecutive integers. The objective is to understand number properties and develop efficient algorithms to identify them.

**METHODOLOGY & TOOL USED::** The method iterates through integers starting from one, checking if the product of a number and its consecutive integer equals n. Python was used because it provides simple looping structures and arithmetic operations suitable for this check.

**BRIEF DESCRIPTION:** A pronic number is defined as the product of two consecutive integers. The function tests integers sequentially until the product either matches n or exceeds it. If a match is found, the function returns True; otherwise, it returns False.

**RESULTS ACHIEVED:** The implementation correctly identified pronic numbers such as two, six, and twelve. It worked efficiently for a variety of inputs and returned results quickly because the product grows rapidly, limiting the number of iterations required.

**DIFFICULTY FACED BY STUDENT:** Students sometimes struggled to determine the stopping condition for the iteration and to ensure that only consecutive integers were considered. Understanding the mathematical definition of pronic numbers and translating it into code also required attention.

**SKILLS ACHIEVED:** Students gained experience in iterative computation, number property recognition, and translating mathematical definitions into efficient algorithms. They also improved their ability to implement logical conditions effectively.

**Practical No: 10****Date:** 04-09-2025

**TITLE:** Implementation of a Function to Compute Prime Factors of a Number

**AIM/OBJECTIVE(s):** The aim is to implement the function prime factors n which returns all prime factors of a given integer. The objective is to understand integer factorization and develop algorithms to decompose numbers into their prime components.

**METHODOLOGY & TOOL USED:** The method iterates through possible divisors starting from two, repeatedly dividing n by each prime divisor until it no longer divides evenly. Python was used for its ability to handle loops, conditional checks, and integer arithmetic efficiently.

**BRIEF DESCRIPTION:** Prime factorization is the process of expressing a number as a product of prime numbers. The implemented function finds each prime divisor in sequence and collects them into a list. The process continues until the remaining value is one, ensuring all prime factors are included.

**RESULTS ACHIEVED:** The function correctly returned prime factors for a variety of test inputs, including small numbers, large numbers, and semiprimes. It performed efficiently due to the sequential division approach and handled multiple occurrences of the same prime factor accurately.

**Difficulties Faced by Students:** Students sometimes had difficulty understanding repeated division to handle powers of the same prime and ensuring that only prime factors were included. Managing edge cases

such as  $n$  equals one or prime numbers themselves required careful attention.

**SKILLS ACHIEVED:** Students gained experience in integer factorization, loop control, and algorithmic problem solving. They also strengthened their ability to translate mathematical concepts into practical programming solutions.

**Practical No: 11****Date:** 11-09-2025**TITLE:** Function to Count Distinct Prime Factors of a Number

**Aim/Objective:** The aim is to implement the function count distinct prime factors n which calculates the number of unique prime factors of a given number. The objective is to understand prime factorization and develop the ability to identify distinct factors efficiently.

**Methodology and Tools Used:** The method iterates through possible divisors starting from two, checking divisibility and collecting each prime factor into a set to ensure uniqueness. Python was used because it provides efficient set operations, loops, and integer arithmetic for implementing this logic.

**Brief Description:** The function identifies all prime factors of a number and counts only the distinct ones. By using a set to store factors, duplicates are automatically excluded. The final size of the set gives the number of unique prime factors.

**Results Achieved:** The implementation successfully returned the count of distinct prime factors for a variety of inputs, including numbers with repeated prime factors and semiprimes. It worked efficiently and accurately for moderate to large numbers.

**Difficulties Faced by Students:** Students sometimes struggled with ensuring that repeated prime factors were not counted multiple times and with correctly handling edge cases such as prime numbers or one. Implementing efficient factor collection required careful use of data structures.



**Skills Achieved:** Students developed skills in prime factorization, set operations, and efficient counting of unique elements. They also strengthened their ability to translate mathematical definitions into practical, optimized code.

**Practical No: 12****Date:** 09-11-2025**TITLE:** Program to Check for Prime Power

**AIM/OBJECTIVE(s):** To write a function `is_prime_power(n)` that verifies if a number  $n$  can be expressed in the form  $p^k$ , where  $p$  is a prime number and  $k \geq 1$ .

**METHODOLOGY & TOOL USED:**

- Tool: Python 3, time, tracemalloc, and the math module (for sqrt).
- Methodology: The program first implements a helper function `is_prime`. The main `is_prime_power(n)` function then iterates from 2 up to the square root of  $n$ . It looks for the first prime factor  $p$  of  $n$ . Once  $p$  is found, the function repeatedly divides  $n$  by  $p$  until it is no longer divisible. If the remaining value of  $n$  is 1, it means  $n$  was composed only of that single prime factor  $p$ , and the function returns True. If a factor is found but isn't prime, or if after dividing,  $n$  is not 1, it returns False.

**BRIEF DESCRIPTION:** This program checks if a number is a power of a single prime number.

**RESULTS ACHIEVED:** The function correctly returns True or False based on the prime power property. The performance metrics for time and memory are successfully recorded.

**DIFFICULTY FACED BY STUDENT:** The main difficulty was conceptual. It's inefficient to check every possible prime  $p$  and every power  $k$ . The key insight was that if  $n = p^k$ , then  $p$  must be the smallest prime factor of  $n$ . The algorithm was built around finding this smallest prime factor and then verifying that no other prime factors exist.

**SKILLS ACHIEVED:** Developed algorithmic thinking by breaking down a complex mathematical property pk into a simpler, verifiable process.

**Practical No: 13****Date:** 09-11-25**TITLE:** Program to Check for Mersenne Prime.

**AIM/OBJECTIVE(s):** To write a function `is_mersenne_prime(p)` that, given a prime  $p$ , checks if the corresponding Mersenne number  $2p - 1$  is also a prime number.

**METHODOLOGY & TOOL USED:** The function first calculates the Mersenne number,  $M = 2p - 1$ , using Python's efficient exponentiation. It then re-uses the same `is_prime()` helper function from the previous question to perform a standard primality test on  $M$ .

**BRIEF DESCRIPTION:** The program takes a prime number  $p$  and checks if  $2p - 1$  is also prime.

**RESULTS ACHIEVED:** The function correctly identifies whether  $2p - 1$  is prime for a given  $p$ .

**DIFFICULTY FACED BY STUDENT:** The logic itself was simple. The main difficulty is one of performance and scale. The value of  $2p - 1$  grows extremely quickly. While Python handles arbitrarily large integers, the `is_prime()` function becomes very slow for large  $p$ . For this assignment, the standard primality test is sufficient, but it was clear that this method is not feasible for finding large Mersenne primes.

**SKILLS ACHIEVED:** Gained experience with Python's ability to handle arbitrarily large integers.

**Practical No: 14**

**Date:** 09-11-25

**TITLE:** Program to Generate Twin Primes .

**AIM/OBJECTIVE(s):** To write a function `twin_primes(limit)` that generates and returns a list of all twin prime pairs up to a given limit.

**METHODOLOGY & TOOL USED:**

- Tool: Python 3, time, tracemalloc.
- Methodology: The program iterates through all numbers  $i$  from 2 up to  $\text{limit} - 2$ . In each iteration, it uses the `is_prime()` helper function to check if both  $i$  and  $i + 2$  are prime. If this condition is met, the tuple  $(i, i + 2)$  is appended to a results list. Finally, the complete list of pairs is returned.

**BRIEF DESCRIPTION:** This program finds all pairs of prime numbers that have a difference of 2. Examples include (3, 5), (5, 7), (11, 13), and (17, 19). The program finds all such pairs below the user's input limit.

**RESULTS ACHIEVED:** The program successfully generates a list of all twin prime pairs within the specified range. The time and memory metrics reflect the total cost of finding all pairs.

**DIFFICULTY FACED BY STUDENT:** The implementation was straightforward. The main difficulty is the inefficiency of the algorithm. It calls `is_prime()` twice for every number, which is highly redundant. A much faster (but more complex) solution would be to generate all primes up to limit once using a Sieve of Eratosthenes, and then iterate through that list to find pairs. For the scope of this assignment, the simpler, less efficient method was chosen.

**SKILLS ACHIEVED:** Learned to implement a search algorithm by iterating through a range and checking a property.

**Practical No: 15**

**Date:** 09-11-25

**TITLE:** Program to Count Number of Divisors.

**AIM/OBJECTIVE(s):** To write a function `count_divisors(n)` that returns how many positive divisors a given number `n` has.

**METHODOLOGY & TOOL USED:**

- Tool: Python 3, time, tracemalloc, and math(for sqrt).
- Methodology: The program uses an optimized  $O(\sqrt{n})$  algorithm. It iterates `i` from 1 up to the square root of `n`. If `i` divides `n`, it means we have found a pair of divisors: `i` and `n/i`. Therefore, we add 2 to the count. A special case is when `n` is a perfect square. In this case, `i` and `n/i` are the same, so we should only add 1 to the count. This is handled by checking if `i * i == n`.

**BRIEF DESCRIPTION:** This program counts the total number of positive integers that divide a given number evenly.

**RESULTS ACHIEVED:** The function correctly and efficiently counts the divisors for any positive integer. The performance metrics are also captured.

**DIFFICULTY FACED BY STUDENT:** The initial brute-force idea is to check every number from 1 to `n`, which is very slow ( $O(n)$ ). The difficulty was in understanding and correctly implementing the  $O(\sqrt{n})$  optimization. The trickiest part was handling the perfect square case, as failing to do so would result in an off-by-one error.

**SKILLS ACHIEVED:** Learned to analyze and implement an optimized algorithm, improving from a brute-force  $O(n)$  solution to an efficient  $O(\sqrt{n})$  one.

- Gained skill in handling specific mathematical edge cases (perfect squares) to prevent off-by-one errors.
- Practiced using the `math.sqrt()` function to set an efficient loop boundary.

**Practical No: 16****Date:** 16-11-25**TITLE:** Efficient Calculation of the Sum of Proper Divisors (Aliquot Sum).**AIM/OBJECTIVE(s):** Develop an efficient Python function, aliquot\_sum(n), to calculate the sum of all proper divisors of a positive integer n.**METHODOLOGY & TOOL USED:** The solution was implemented in Python using an optimized  $O(\sqrt{n})$  iteration algorithm. Instead of checking every number up to n, the code checks only up to  $\sqrt{n}$  to find divisor pairs, which significantly improves speed. Execution time was measured using the built-in time.time() function.**BRIEF DESCRIPTION:** The Python code calculates the sum by starting with 1 (the first proper divisor). It then iterates from 2 upwards. For every number i that divides n, both i and its pair  $n/i$  are added to the total. A check is included to prevent the square root of n from being double-counted when n is a perfect square.**RESULTS ACHIEVED:** The algorithm successfully calculates the aliquot sum. For example, the proper divisors of  $n = 12$  are 1, 2, 3, 4, 6, resulting in an Aliquot Sum of 16. Performance profiling confirmed the high efficiency of the  $O(\sqrt{n})$  approach, showing execution time.**DIFFICULTY FACED BY STUDENT:** The primary challenge was the system-dependent nature of performance measurement. Both the resource module and the cross-platform psutil library presented obstacles, highlighting the need to understand deployment environments when measuring system resources.



**SKILLS ACHIEVED:** Algorithmic Thinking (developing the optimized  $O(\sqrt{n})$  solution), Basic Performance Profiling (using the time module), Error Handling and Debugging (troubleshooting operating system-specific module errors), and Input/Output Handling for creating a complete, user-friendly command-line application.

**Practical No: 17****Date:** 16-11-25**TITLE:** Amicable Number Identification.

**AIM/OBJECTIVE(s):** The primary objective of this project was to create a Python function, `are_amicable(a, b)`, that can accurately determine if two positive integers are an amicable pair.

**METHODOLOGY & TOOL USED:** The methodology for this project was a direct computational approach. The program takes two integer inputs, calculates their respective divisor sums, and then applies a conditional check to confirm the amicable relationship. To accomplish this, the Python programming language was the primary tool used. For performance analysis, the built-in time module was used to measure execution speed, and the tracemalloc module was used to track memory utilization.

**BRIEF DESCRIPTION:** The solution involves calculating the sum of the proper divisors for each of the two input numbers, 'a' and 'b'. The program then checks if the sum of 'a's divisors equals 'b' and if the sum of 'b's divisors equals 'a'. If this condition is true, the pair is amicable. The core of the program is an efficient divisor summation function that checks divisors only up to the square root of the number to optimize performance.

**RESULTS ACHIEVED:** A functional, self-contained Python script was successfully created. The program accurately determines if a pair of numbers is amicable and outputs a simple True or False result. In addition to this core function, the script also provides quantitative data on its own efficiency, reporting the total execution time and the memory utilization.

**DIFFICULTY FACED BY STUDENT:** A common difficulty faced by students is designing an efficient algorithm to calculate the sum of proper divisors. Many first attempts involve a simple scan from 1 up to the number itself, which is very slow for large numbers. The more optimized approach, checking only up to the square root, is a key concept to learn. Additionally, the requirement to measure performance introduces a learning curve for using specialized libraries like `tracemalloc`.

**SKILLS ACHIEVED:** Through this project, students gain proficiency in several key areas. They learn about algorithmic efficiency by implementing an optimized divisor sum function. They practice function decomposition by breaking the problem into smaller, manageable functions like `sumdiv` and `are_amicable`. Students also gain practical experience in performance measurement by using Python's `time` and `tracemalloc` modules. Finally, the project reinforces the ability to translate a formal mathematical definition into concrete programming logic.

**Practical No: 18****Date: 16-11-25****TITLE:** Multiplicative Persistence Analysis.

**AIM/OBJECTIVE(s):** The primary objective of this project was to develop a Python program to calculate the "multiplicative persistence" of a given positive integer. This involves repeatedly multiplying the digits of a number until a single-digit number is obtained and counting the number of steps required for this process.

**METHODOLOGY & TOOL USED:** The project was implemented using the Python programming language. The methodology involved creating a main function that uses a while loop to check if the number is still greater than 9. Inside this loop, a separate helper function is called, which converts the number to a string, iterates through its individual digits, multiplies them together, and returns the new product. A counter variable tracks the number of steps. For performance measurement, the standard time module was used to benchmark execution time, and the tracemalloc module was employed to track peak memory allocation.

**BRIEF DESCRIPTION:** The final script first prompts the user to enter a positive integer. Once the input is received, it starts the performance timers and memory tracker. The program then computes the multiplicative persistence by repeatedly calculating the product of the number's digits until the result is a single-digit number. After the calculation is complete, the script stops the trackers and prints three pieces of information: the total persistence count, the execution time and the peak memory used.

**RESULTS ACHIEVED:** The program successfully meets all stated objectives. It correctly computes the multiplicative persistence for any positive integer input, from simple cases to more complex ones. Furthermore, the script provides clear and useful metrics for both

execution time and memory usage, which helps in understanding the program's efficiency.

**DIFFICULTY FACED BY STUDENT:** A common difficulty faced was managing the conversion between data types. The logic requires treating the number as an integer for mathematical comparisons but as a string to iterate through its individual digits. Another challenge was structuring the while loop condition correctly to ensure it terminates at the right time (when the number is less than 10) and doesn't run an extra, unnecessary step. Finally, correctly implementing the tracemalloc library required understanding how to start the trace and how to get the peak memory usage.

**SKILLS ACHIEVED:** Through this project, several key skills were developed. Students gained proficiency in fundamental Python programming, including defining functions, using while loops for iterative processes, and performing data type conversions between integers and strings. More importantly, students learned foundational code profiling techniques by using standard libraries to measure and report on a program's time and space complexity, which is a crucial skill for writing efficient and optimized code.

**Practical No: 19**

**Date: 16-11-25**

**TITLE:** Highly Composite Number Verification.

**AIM/OBJECTIVE(s):** The aim of this project was to successfully implement a Python function capable of verifying whether a given positive integer,  $N$ , possesses the property of being a Highly Composite Number, meaning it has more divisors than any smaller positive integer.

**METHODOLOGY & TOOL USED:** The chosen methodology leveraged Python's standard capabilities, utilizing the time module for precise execution timing and the tracemalloc module to monitor and report on peak memory consumption during the program's run. The primary tool employed was a simple function that efficiently calculates the number of divisors for any integer by iterating only up to its square root. This core function was then used within the main logic to perform the comparative check against all integers smaller than the input number  $N$ .

**BRIEF DESCRIPTION:** The program first prompts the user to enter an integer. It uses a dedicated function to determine the divisor count for the input  $N$ . It then employs a loop that sequentially checks every positive integer from 1 up to  $N-1$ . In each step of the loop, the program compares the divisor count of the current smaller number to the divisor count of  $N$ . If any smaller number is found to have an equal or greater number of divisors, the input number  $N$  is immediately identified as not highly composite. The entire process of input, calculation, and verification is contained within the boundaries of the performance tracking functions.

**RESULTS ACHIEVED:** The results confirmed the functional correctness of the implementation, successfully returning a Boolean value (True or False) that accurately reflects the highly composite status of the input number. Crucially, the program provides quantitative performance

metrics, specifically outputting the execution time and the peak memory usage. This allows for an objective assessment of the code's efficiency for the given input size.

**DIFFICULTY FACED BY STUDENT:** A potential difficulty students face with this problem is the significant computational complexity of the direct method, which is approximately  $O(N \cdot \sqrt{N})$ . For large input numbers, the brute-force comparison loop and repeated square-root calculations can lead to excessively long execution times. Students must learn to anticipate and manage such performance bottlenecks, recognizing that finding highly composite numbers quickly requires mathematical optimization beyond simple looping structures.

**SKILLS ACHIEVED:** The exercise fostered several valuable skills, including proficient use of core Python functions, disciplined implementation of conditional logic, and function-based code structuring. Most importantly, students achieved a foundational skill in applied performance analysis by learning how to instrument code to measure time and memory usage using built-in tools. This skill is essential for developing robust and resource-efficient software in real-world applications.

**Practical No: 20****Date:** 16-11-25**TITLE:** : Implementation of Modular Exponentiation.

**AIM/OBJECTIVE(s):** The primary aim of this project was to develop and implement an efficient algorithm for modular exponentiation. The objective was to create a Python function that correctly calculates  $(\text{base}^{\text{exponent}}) \% \text{modulus}$ .

**METHODOLOGY & TOOL USED:** The core methodology involved implementing the 'binary exponentiation' or 'exponentiation by squaring' algorithm. This iterative approach is significantly more efficient than naive multiplication by handling the exponent in its binary form. The primary tool used was the Python programming language. For performance analysis, the time module was used to measure execution speed, and the tracemalloc module was employed to track peak memory usage.

**BRIEF DESCRIPTION:** The Python script first prompts the user to enter three integers: the base, the exponent, and the modulus. It then calls a dedicated function named `mod_exp` that implements the binary exponentiation logic. This function uses a while loop to process the exponent, applying modulo operations at each step to keep the numbers manageable. Before calling this function, the script starts the tracemalloc tracker and records a start time. After the calculation is complete, it stops the timer and the memory tracker, then prints the final computed result, the total execution time, and the peak memory consumed during the process.

**RESULTS ACHIEVED:** A functional and efficient Python script was successfully created. The program accurately computes the result of modular exponentiation for a wide range of integer inputs, handling large exponents without overflowing. Furthermore, the script successfully



provides clear and useful performance metrics, outputting the precise execution time and the peak memory utilization.

**DIFFICULTY FACED BY STUDENT:** A primary difficulty students often face is grasping the core logic of the binary exponentiation algorithm, as it is less intuitive than a simple loop. Translating this logic into bug-free code can be challenging, particularly ensuring the modulo operation is applied at each intermediate step. Another potential hurdle is learning to correctly use the `time` and `tracemalloc` modules to start and stop the tracking at the appropriate times to isolate the function's specific performance.

**SKILLS ACHIEVED:** Upon completing this project, students achieved several key skills. They gained a practical understanding of algorithmic optimization by implementing an efficient alternative to a naive approach. They solidified their Python programming abilities, including function definition and handling user input. Most importantly, they acquired foundational knowledge in code profiling and performance analysis, learning how to benchmark a function's speed and memory footprint, which is a valuable skill in software development.

## Practical No: 21

**Date:** 16-11-25

**TITLE:** Modular Multiplicative Inverse in Python

**AIM/OBJECTIVE(s):** To write a Python program that computes the Modular Multiplicative Inverse of a number  $a$  under modulo  $m$ , i.e., to find an integer  $x$ .

- To understand the concept of modular arithmetic and its applications in number theory and cryptography.
- To implement the Extended Euclidean Algorithm in Python for finding modular inverses.
- To check the condition  $\gcd(a, m) = 1$  for the existence of the inverse.
- To write a clear, beginner-friendly Python function `mod_inverse(a, m)` that returns the inverse if it exists, otherwise `None`.
- To measure and display time and memory utilization of the program for performance awareness.

### METHODOLOGY & TOOL USED:

1. Problem Understanding
  - Define the mathematical problem: find  $x$  such that .
  - Ensure the condition  $\gcd(a, m) = 1$  holds, otherwise the inverse does not exist.
2. Algorithm Selection
  - Use the Extended Euclidean Algorithm to compute.
  - If  $\gcd(a, m) = 1$ , then  $x$  is the modular inverse of  $a$  modulo  $m$ .
3. Implementation in Python
  - Write a function `mod_inverse(a, m)` that:
    - Applies the Extended Euclidean Algorithm.
    - Returns the modular inverse if it exists, otherwise returns `None`.
  - Include time measurement (using `time` module) and memory tracking (using `tracemalloc`) to evaluate performance.
4. Testing and Validation
  - Test the function with sample inputs where the inverse exists.

- Test with inputs where the inverse does not exist .
- Compare results with theoretical expectations to ensure correctness.

**BRIEF DESCRIPTION:** This inverse exists only when  $\gcd(a, m) = 1$ . The program uses the Extended Euclidean Algorithm to compute the inverse efficiently. The algorithm finds integers  $x$  and  $y$  such that:

$$a \cdot x + m \cdot y = \gcd(a, m)$$

If the gcd is 1, then  $x$  is the modular inverse of  $a$  modulo  $m$ . The Python implementation is kept simple and beginner-friendly, with additional features to measure execution time and memory usage for performance analysis.

#### **RESULTS ACHIEVED:**

Successfully implemented a Python function `mod_inverse(a, m)` that computes the modular multiplicative inverse using the Extended Euclidean Algorithm.

#### **DIFFICULTY FACED BY STUDENT:**

- Understanding modular arithmetic concepts.
- Linking mathematical theory with Python implementation.
- Handling recursion in the Extended Euclidean Algorithm.
- Managing edge cases when inverse/order does not exist.
- Avoiding common coding errors in modulo operations.
- Interpreting time and memory utilization results.

**SKILLS ACHIEVED:** Modular Arithmetic: Understanding congruence and inverse under modulo. Extended Euclidean Algorithm: Applying it to find modular inverses.

- Recursion
- Input Validation: Checking if inverse exists.
- Mathematical Reasoning: Translating number theory into working code.

- Code Clarity: Writing readable, well-commented functions for beginners.

**Practical No: 22**

**Date:** 16-11-25

**TITLE::** Implementation of Chinese Remainder Theorem (CRT) Solver in Python

**AIM/OBJECTIVE(s):**

To implement a Python program that solves a system of linear congruences using the Chinese Remainder Theorem (CRT).

- Apply the Chinese Remainder Theorem for finding unique solutions modulo product of coprime moduli.
- Implement a Python function `crt(remainders, moduli)` to compute the solution.
- Verify correctness with sample inputs and outputs.
- Analyze efficiency through time and memory usage.

**METHODOLOGY & TOOL USED:**

- Apply the Chinese Remainder Theorem to solve simultaneous congruences.
- Compute product of moduli, partial products, and modular inverses.
- Implement logic in Python and validate with test cases.

Tools used:

- Python language for coding.
- Extended Euclidean Algorithm for modular inverse.
- `time` and `tracemalloc` modules for performance analysis.
- IDE/Editor ( Jupyter Notebook) for program execution.

**BRIEF DESCRIPTION:** The program applies the Chinese Remainder Theorem in Python to solve simultaneous congruences efficiently.

**RESULTS ACHIEVED:** The program correctly solved the given system of congruences using CRT, producing accurate and efficient results.

**DIFFICULTY FACED BY STUDENT:** Students faced challenges in understanding modular arithmetic concepts, selecting suitable algorithms, handling recursion/iteration, managing edge cases where solutions do not exist, and avoiding common coding errors during Python implementation.

**SKILLS ACHIEVED:** Modular arithmetic basics Extended Euclidean Algorithm

- Recursive problem-solving
- Input validation (checking coprimality)
- Translating math into Python code
- Clear, beginner-friendly documentation

**Practical No: 23****Date: 16-11-25****TITLE:**Implementation of Quadratic Residue Checker.

**AIM/OBJECTIVE(s):** To implement a Python function `is_quadratic_residue(a, p)` that checks whether the congruence has a solution.

- Understand the concept of quadratic residues in modular arithmetic.
- Apply Euler's Criterion to test if  $a$  is a quadratic residue modulo prime  $p$ .
- Implement the check in Python using efficient modular exponentiation.
- Validate correctness with sample inputs where solutions exist and do not exist.
- Analyze performance using time and memory measurement tools.

**METHODOLOGY AND TOOL USED:**

- Use Euler's Criterion:  $\rightarrow$  quadratic residue, else non-residue.
- Implement efficient modular exponentiation in Python.
- Validate results with test cases where solutions exist and do not exist.

**Tools Used**

- Python language for implementation.
- `math` / `pow` function for modular exponentiation.
- `time` and `tracemalloc` modules for performance analysis.
- IDE/Editor (VS Code, Jupyter Notebook, etc.) for coding and testing.

**BRIEF DESCRIPTION:** The program checks if has a solution using Euler's Criterion in Python.

**RESULTS ACHIEVED:** The function correctly identified quadratic residues and non-residues, with efficient execution and minimal resources.

**DIFFICULTY FACED:** Students struggled with modular exponentiation, understanding Euler's Criterion, and handling prime modulus conditions.

**SKILLS ACHIEVED:** Modular arithmetic with prime modulus Euler's Criterion for quadratic residues

- Efficient exponentiation using `pow()` with modulus
- Input validation for mathematical constraints
- Translating number theory concepts into Python code



**Practical No: 24**

**Date: 16-11-25**

**TITLE:** Order Mod Function in Modular Arithmetic.

**AIM/OBJECTIVE(s):** To implement a Python function `order_mod(a, n)` that finds the smallest positive integer  $k$  such that .

- Understand the concept of order of an element in modular arithmetic.
- Implement the logic in Python to compute the order.
- Verify correctness with sample inputs.
- Measure execution time and memory usage for performance analysis.

**METHODOLOGY AND TOOL USED:**

- Check if  $\gcd(a, n) = 1$ ; otherwise, order does not exist.
- Compute successive powers of  $a$  modulo  $n$  until the result is 1.
- Return the smallest exponent  $k$ .
- Validate with test cases and track performance.

**BRIEF DESCRIPTION:** The program finds the order of an integer modulo  $n$ , showing the smallest exponent.

**RESULTS ACHIEVED:** The function correctly computed orders for valid inputs and identified cases where order does not exist.

**DIFFICULTY FACED BY STUDENT:** Students struggled with understanding the mathematical definition of order, handling gcd conditions, and avoiding infinite loops in code.

**SKILLS ACHIEVED:** The practical successfully demonstrated the concept of Modular Arithmetic and their efficient verification using Python.

**Practical No: 25**

**Date:** 16-11- 2025

**TITLE:** Implementation of Fibonacci Prime Checker in Python.

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

- Understand the concept of Fibonacci primes.
- Apply mathematical tests for Fibonacci numbers and primality.
- Implement the combined check in Python.
- Validate correctness with sample inputs.
- Measure execution time and memory usage.

**METHODOLOGY & TOOL USED:**

Methodology

- Use the property:  $n$  is Fibonacci .
- Check primality using trial division up to  $\sqrt{n}$ .
- Combine both checks in a single function.
- Test with known Fibonacci primes and non-primes. Tools used
- Python language for implementation.
- `time` and `tracemalloc` modules for performance analysis.
- IDE/Editor ( Jupyter Notebook) for coding and testing.

**BRIEF DESCRIPTION:** The program verifies whether a number is both Fibonacci and prime, identifying Fibonacci primes efficiently.

**RESULTS ACHIEVED:** The function correctly identified Fibonacci primes (e.g., 2, 3, 5, 13) and excluded non-prime Fibonacci numbers.

**DIFFICULTY FACED BY STUDENT:** Students struggled with applying the Fibonacci test, optimizing prime checks, and combining both conditions correctly.

**SKILLS ACHIEVED:** The practical successfully demonstrated the concept of Fibonacci primes and their efficient verification using Python.

**Practical No: 26****Date:** 16-11-2025**TITLE:** Lucas Sequence Generator Performance Analysis

**AIM/OBJECTIVE(s):** The primary aim of this project was to develop a Python program that generates the first n numbers of the Lucas sequence based on user input. The objective extended to analyzing the efficiency of the code by calculating and displaying the specific memory utilization in bytes and the execution time in nanoseconds required to perform the generation.

**METHODOLOGY & TOOL USED:** The project was implemented using the Python programming language, chosen for its readability and robust standard libraries. The methodology involved defining a specific function to handle the logic of the Lucas sequence, which initializes with 2 and 1. To fulfill the performance tracking requirements, the built-in time module was utilized to capture high-precision time in nanoseconds, and the tracemalloc library was employed to trace memory allocation blocks. The input was processed using standard input functions, and the logic relied on iterative addition.

**BRIEF DESCRIPTION:** The Lucas sequence is an integer sequence similar to the Fibonacci numbers, differing only in its starting values, which are 2 and 1. Each subsequent number is the sum of the previous two. The program accepts an integer n from the user and constructs a list starting with the base cases. A loop iteratively calculates the next value by summing the last two entries in the list and appends it. While this calculation occurs, the program runs background processes to monitor the peak memory used and the exact time elapsed between the start and end of the function call.

**RESULTS ACHIEVED:** The developed code successfully takes an integer input and outputs the correct sequence of Lucas numbers for that length. For instance, an input of 5 correctly yields the sequence [2, 1, 3, 4, 7]. Furthermore, the program provides accurate metrics for the

computational cost, printing the peak memory usage in bytes and the execution duration in nanoseconds immediately after the sequence is displayed. This confirms both the logical correctness of the generator and the successful integration of performance monitoring tools.

**DIFFICULTY FACED BY STUDENT:** A common difficulty faced during this task was correctly handling the base cases, specifically when the user requests only 1 or 0 terms, which often led to index errors if not explicitly managed. Additionally, students struggled with the placement of the performance counters; placing the start timer before the user input resulted in inaccurate execution times that included the time the user took to type. Understanding how to interpret and print the output of the memory tracer also proved challenging for those new to system-level libraries.

**SKILLS ACHIEVED:** By completing this report and coding task, skills were achieved in algorithmic thinking, specifically regarding iterative sequence generation. Students learned the importance of performance profiling, gaining the ability to measure code efficiency concretely using time and memory metrics. The task also reinforced best practices in Python programming, including function modularity, library utilization, and robust input handling to prevent runtime errors during execution.

**Practical No: 27****Date:** 16-11-2025**TITLE:** Algorithmic Determination of Perfect Powers.

**AIM/OBJECTIVE(s):** The primary objective of this task was to develop a Python function named `is_perfect_power(n)` that determines whether a given integer can be expressed as a base raised to an exponent  $b$  (where  $a > 0$  and  $b > 1$ ). A secondary aim was to implement performance monitoring to measure the exact execution time in nanoseconds and the peak memory utilisation in bytes during the process.

**METHODOLOGY & TOOL USED:** The tool selected for this implementation was the Python programming language, utilizing its standard library modules. The methodology involved using the `math` module to calculate logarithmic bounds and roots, the `time` module to capture high-precision timestamps, and the `tracemalloc` module to trace memory allocation blocks. The algorithmic approach relied on a brute-force check of possible exponents ranging from 2 up to the binary logarithm of the input number, calculating the corresponding root for each to verify if it forms an integer solution.

**BRIEF DESCRIPTION:** The program begins by accepting an integer input from the user. The core function handles edge cases, such as returning `true` for the number 1, before entering a loop that tests potential exponents. For every valid exponent  $b$ , the code calculates the  $b$ -th root of the number and checks if raising the rounded root back to the power of  $b$  yields the original number. While this logic executes, the system tracks memory usage and records the start and end times to compute the duration.

**RESULTS ACHIEVED:** The resulting program successfully identifies perfect powers, correctly validating inputs such as 8 (which is  $2^3$ ) or 16 (which is  $4^2$  or  $2^4$ ), while rejecting non-perfect powers like 10. It outputs a boolean result indicating the nature of the number. Additionally, the console displays the precise execution time in nanoseconds and the peak memory consumed in bytes, providing a clear view of the code's efficiency.

**DIFFICULTY FACED BY STUDENT:** A common difficulty faced during this task is determining the efficient upper limit for the loop to prevent unnecessary iterations, as iterating up to the number itself causes performance issues. Students also frequently struggle with floating-point precision errors when calculating roots, where a mathematically integer result might appear as a float like 2.99999, leading to incorrect validation if not properly rounded or cast to an integer.

**SKILLS ACHIEVED:** Through this exercise, students gained proficiency in mathematical algorithm design and numerical constraints handling. They developed the ability to perform code instrumentation and profiling, learning how to quantify software efficiency through time and space complexity metrics. This task also reinforced the importance of handling edge cases and input validation in computational logic.



**Practical No: 28****Date:** 16-11-2025**TITLE:** Analysis of Collatz Sequence Length.

**AIM/OBJECTIVE(s):** The primary objective of this experiment is to develop a computational solution to calculate the length of the Collatz sequence for any given positive integer. A secondary aim is to evaluate the efficiency of the solution by measuring the execution time in nanoseconds and memory utilization in bytes during the runtime.

**METHODOLOGY & TOOL USED:** The methodology involves implementing an iterative algorithm using the Python programming language. The standard input-output functions are used to interact with the user. To measure performance, specific Python libraries are employed: the time module is used to capture high-precision timing data, and the tracemalloc library is utilized to monitor memory allocation blocks. The logic follows the standard Collatz rules where even numbers are halved and odd numbers are tripled and incremented by one until the number one is reached.

**BRIEF DESCRIPTION:** The project focuses on the Collatz Conjecture, a famous unsolved problem in mathematics which states that any positive integer will eventually reach the number one. The program accepts a user-defined integer and enters a loop that applies the specific arithmetic operations. While the calculation proceeds, the system tracks the computational resources consumed. This dual approach allows for both verifying the mathematical property and understanding the computational cost associated with potentially long sequences.

**RESULTS ACHIEVED:** The program successfully computes the number of steps required for various integers to converge to one. It provides immediate feedback to the user, displaying not only the sequence length but also the precise time taken for execution and the peak memory usage. These results verify that even small numbers can generate long sequences, illustrating the unpredictable nature of the conjecture while

confirming that the algorithm runs within acceptable time and memory limits for standard inputs.

**DIFFICULTY FACED BY STUDENT:** Students often encounter challenges in understanding the setup of performance monitoring tools, specifically the correct placement of start and stop triggers for memory tracking. Additionally, grasping the concept of a while loop with a condition that isn't a simple counter can be initially confusing. There is also the potential difficulty of handling invalid inputs, such as non-integers or negative numbers, which requires implementing robust error handling to prevent the program from crashing.

**SKILLS ACHIEVED:** Through this exercise, learners acquire essential skills in algorithmic design and control flow implementation using loops and conditional statements. They gain practical experience in software profiling by learning how to instrument code to measure execution time and memory usage. Furthermore, the task enhances problem-solving abilities by requiring the translation of a mathematical definition into executable code, alongside improved competency in Python syntax and standard library usage.

**Practical No: 29****Date:** 16-11-2025**TITLE:** Polygonal Number Calculation.

**AIM/OBJECTIVE(s):** The primary objective of this activity was to develop a Python program capable of calculating the n-th s-gonal number based on user-provided integers. A secondary aim was to integrate performance analysis tools to measure the exact execution time in nanoseconds and the peak memory utilisation in bytes during the computation.

**METHODOLOGY & TOOL USED:** The project was implemented using the Python programming language, chosen for its simplicity and robust standard libraries. The methodology involved translating the mathematical formula for polygonal numbers into a Python function. To achieve the performance monitoring goals, the tracemalloc library was utilized for tracking memory allocation, and the time module was employed to capture high-resolution timestamps. The tools allowed for precise measurement of system resources without requiring external software.

**BRIEF DESCRIPTION:** The developed script functions by first prompting the user to input the number of sides (s) and the specific position (n) in the sequence. It then processes these inputs using the algebraic formula  $((s - 2) * n^2 - (s - 4) * n) / 2$  to determine the result. Surrounding this calculation, the code initiates a memory trace and records the system time, stopping these monitors immediately after the result is found. The final output presents the calculated number alongside the specific resource usage metrics.

**RESULTS ACHIEVED:** The program successfully calculated the correct s-gonal number for various test inputs, demonstrating the accuracy of the implemented logic. In addition to the mathematical result, the console output provided the peak memory usage in bytes and the execution duration in nanoseconds, effectively showing the efficiency of the simple mathematical operations used in the solution.

**DIFFICULTY FACED BY STUDENT:** A common difficulty faced was correctly deriving and implementing the general algebraic formula, particularly ensuring the order of operations was correct in Python syntax. Additionally, students struggled with the placement of the timing and memory tracking functions, as placing them incorrectly around the user input prompts would result in inaccurate performance data that included the user's reaction time.

**SKILLS ACHIEVED:** Through this task, students achieved a stronger grasp of translating mathematical algorithms into functional programming code. They also developed essential skills in software profiling, specifically learning how to use internal libraries to benchmark code performance. This exercise enhanced their understanding of time and space complexity in a practical, hands-on manner.

**Practical No: 30****Date:** 16-11-2025**TITLE:** Carmichael Number Verification System.

**AIM/OBJECTIVE(s):** The primary objective of this project is to develop a Python function named `is_carmichael` that verifies whether a given composite number  $n$  acts as a Carmichael number. This involves checking if the number satisfies the modular arithmetic condition where  $a$  to the power of  $n$  minus 1 is congruent to 1 modulo  $n$  for all integers  $a$  that are coprime to  $n$ .

**METHODOLOGY & TOOL USED:** The solution was implemented using the Python programming language, leveraging standard libraries to handle mathematical operations and system metrics. The methodology involved creating helper functions to first establish if a number is composite and then iterating through potential bases to verify the Carmichael condition. The `tracemalloc` library was utilized to trace memory blocks and calculate peak usage in bytes, while the `time` module was employed to capture the start and end timestamps for determining execution speed in nanoseconds.

**BRIEF DESCRIPTION:** A Carmichael number is a specific type of composite positive integer that passes the Fermat primality test for all bases relatively prime to it. The developed program automates the identification process by first filtering out prime numbers. For the remaining composite candidates, the code iterates through integers smaller than the input. It uses the greatest common divisor function to find coprimes and applies modular exponentiation to strictly validate the mathematical definition provided in the problem statement.

**RESULTS ACHIEVED:** The resulting program successfully identifies whether a user-inputted integer is a Carmichael number. Upon execution, it provides a boolean confirmation of the number's status. Additionally, the system outputs precise performance metrics, displaying the exact peak memory utilisation in bytes and the total time taken for

the calculation in nanoseconds, allowing for an analysis of the algorithm's efficiency.

**DIFFICULTY FACED BY STUDENT:** Students frequently encounter difficulties in distinguishing between standard composite numbers and pseudoprimes. A common challenge lies in understanding the specific requirement that the modular condition must hold for all coprime bases, rather than just a single instance. Furthermore, grasping the concept of efficient modular exponentiation to prevent overflow errors when dealing with large exponents often proves to be a complex hurdle for beginners.

**SKILLS ACHIEVED:** By completing this task, students gain a deeper understanding of number theory, specifically Fermat's Little Theorem and primality testing. Practically, they acquire skills in algorithmic logic and performance profiling. They learn how to instrument code to measure resource consumption, a critical skill in optimizing software for speed and memory efficiency.

**Practical No: 31****Date:** 16-11-2025**TITLE:** Implementation of the Miller Rabin Probabilistic Primality Test

**AIM/OBJECTIVE(s):** The aim is to implement the function is prime miller rabin  $n$   $k$  to determine whether a number is probably prime by using a probabilistic approach that increases accuracy through repeated testing.

**METHODOLOGY & TOOL USED:** The method rewrites  $n$  minus 1 in the form  $2$  to the power  $r$  multiplied by  $d$ , selects random bases, and performs modular exponentiation checks. Python was used because it supports large integers and provides simple tools for random and arithmetic operations.

**BRIEF DESCRIPTION:** The Miller Rabin test evaluates whether a number behaves like a prime when tested with several random bases. If any base exposes a contradiction the number is composite. If all tests pass for  $k$  rounds the number is considered probably prime with very low error probability.

**RESULTS ACHIEVED:** The implemented function correctly identified composite and prime numbers and showed strong efficiency even for large inputs. Increasing  $k$  improved confidence in the output without significant performance loss.

**DIFFICULTY FACED BY STUDENT:** Students found the decomposition of  $n$  minus 1 challenging along with modular exponentiation and the idea of probabilistic correctness. Working with very large integers also required careful handling.

**SKILLS ACHIEVED:** Students developed understanding of number theory, modular arithmetic, randomized algorithms, and efficient implementation techniques relevant to modern computational mathematics and cryptography.



**Practical No: 32****Date:** 16-11-2025**TITLE:** Implementation of Pollard's Rho Integer Factorization Algorithm.

**AIM/OBJECTIVE(s):** The objective of this work is to implement the `pollard_rho(n)` function for efficient integer factorization using Pollard's rho algorithm. The aim is to understand a fast, probabilistic method for finding non-trivial factors of large composite numbers and to observe its performance in practical scenarios.

**METHODOLOGY & TOOL USED:** The implementation follows the classic Pollard's rho approach by using a pseudo-random polynomial function and Floyd's cycle-finding technique to detect repeated values. Python was used due to its simplicity and ability to handle large integers. The algorithm repeatedly computes values modulo  $n$ , checks the greatest common divisor between iterates, and identifies non-trivial factors when the gcd becomes greater than one and less than  $n$ . A Miller-Rabin test supports primality checks during the factorization process.

**BRIEF DESCRIPTION:** Pollard's rho is a probabilistic factorization algorithm designed to quickly find factors of composite integers. It relies on generating a sequence that eventually falls into a cycle, and the difference between values in this cycle helps reveal a hidden factor. The function efficiently handles large semiprimes where trial division becomes slow, making it suitable for cryptographic or computational number-theory tasks.

**RESULTS ACHIEVED:** The implemented function successfully identified prime factors of multiple test numbers, including large semiprimes. It demonstrated high speed, returning factors significantly faster than classical deterministic techniques. The supporting recursive factorization routine produced complete decompositions for all evaluated inputs.

**DIFFICULTY FACED BY STUDENT:** Students often struggled with understanding the cycle-finding logic and why gcd operations reveal factors. Handling failures where the algorithm returns  $n$  and ensuring retries with new parameters also posed challenges. Integrating probabilistic primality testing added complexity for those unfamiliar with modular arithmetic.

**SKILLS ACHIEVED:** Through this implementation, students gained stronger intuition about probabilistic number-theory methods, modular computations, and algorithmic optimization. They also improved their coding precision, debugging ability, and understanding of how modern factorization algorithms operate behind the scenes.

**Practical No: 33****Date:** 16-11-2025

**TITLE:** Implementation of a Function to Approximate the Riemann Zeta Function

**AIM/OBJECTIVE(s):** The aim is to create the function `zeta approx s` terms that computes an approximation of the Riemann zeta function by summing the first given number of terms in its infinite series. The objective is to understand how numerical approximations work for complex mathematical functions.

**METHODOLOGY & TOOL USED:** The approximation uses the basic series definition of the zeta function which sums one divided by  $n$  raised to the power  $s$  for  $n$  ranging from one to the number of terms specified. Python was used because it handles floating point operations easily and allows straightforward implementation of loops and power computations.

**BRIEF DESCRIPTION:** The zeta function is an infinite series but in practical computation it is approximated by adding only a finite number of terms. The function `zeta approx s` calculates the partial sum and returns a numerical value that approaches the true value of  $\zeta(s)$  as more terms are added.

**RESULTS ACHIEVED:** The implementation produced accurate approximations for values of  $s$  greater than one. Increasing the number of terms improved the precision of the result at the cost of slightly more computation time.

**DIFFICULTY FACED BY STUDENT:** Students sometimes struggled to understand the convergence of the series and the effect of selecting too few terms. Handling floating point precision and ensuring the function ran efficiently were also common issues.

**SKILLS ACHIEVED:** Students gained experience in series approximation, numerical computation, and practical implementation of mathematical functions using programming. They also strengthened their understanding of convergence and computational accuracy.

**Practical No: 34****Date:** 16-11-2025**TITLE:** Implementation of a Function to Compute the Partition Function

**AIM/OBJECTIVE(s):** The aim is to implement the function partition function  $n$  which computes  $p_n$ , the number of distinct ways to express  $n$  as a sum of positive integers without considering order. The goal is to understand integer partitions and the use of dynamic methods to evaluate them.

**METHODOLOGY & TOOL USED:** The implementation uses a dynamic programming approach. An array is created where each entry stores the number of partitions for that index. The value for each number is built by iterating through all smaller positive integers and updating the counts accordingly. Python was used for its simplicity in handling lists and arithmetic operations.

**BRIEF DESCRIPTION:** The partition function  $p_n$  counts how many different combinations of positive integers can add up to  $n$ . Order does not matter which means that for example three plus two is the same as two plus three. The implemented function constructs the values of  $p_n$  step by step by accumulating contributions from smaller partitions.

**RESULTS ACHIEVED:** The function successfully computed the partition values for a wide range of inputs. For small inputs the results matched known values from mathematical references. The dynamic programming approach provided efficient computation even for moderately large  $n$ .

**DIFFICULTY FACED BY STUDENT:** Students often found it challenging to understand why order must be ignored and how to enforce this in computation. Constructing the dynamic list correctly and avoiding repeated counting was another area of difficulty.

**SKILLS ACHIEVED:** Students gained experience in dynamic programming, integer combinatorics, and recursive structure analysis. They also improved their ability to translate mathematical definitions into efficient computational procedures.