**QUESTION 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** | **Distinc t**  **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\_facto rs( 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

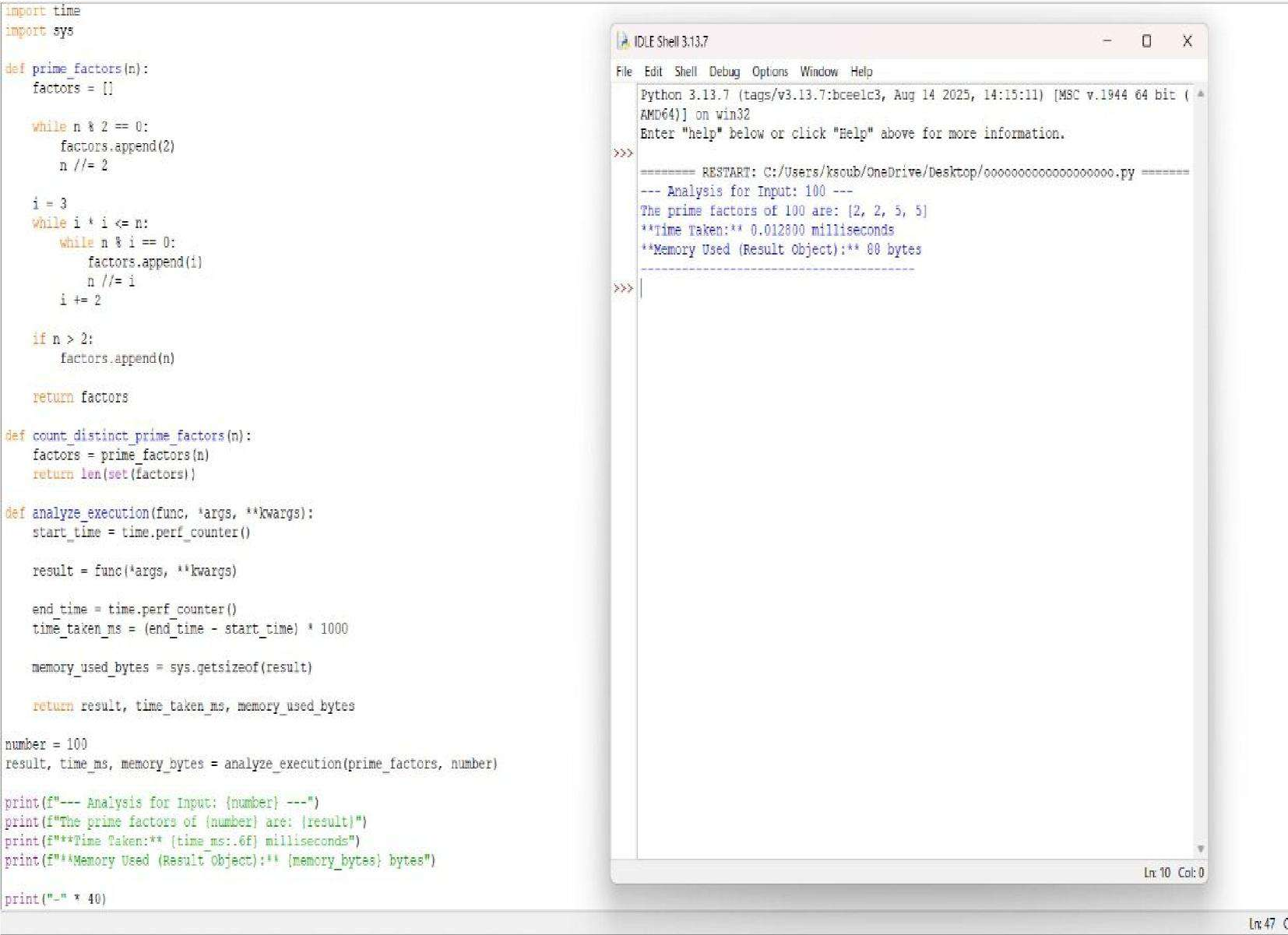
**SKILLS ACHIEVED**:

**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.





**QUESTION 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.

1. **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 .

1. **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.

1. **Basic I/O Operations:**

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

1. **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.

1. 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.

1. 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.

1. 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.

1. **Handling repeated multiplication for prime powers:**

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

1. **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:**

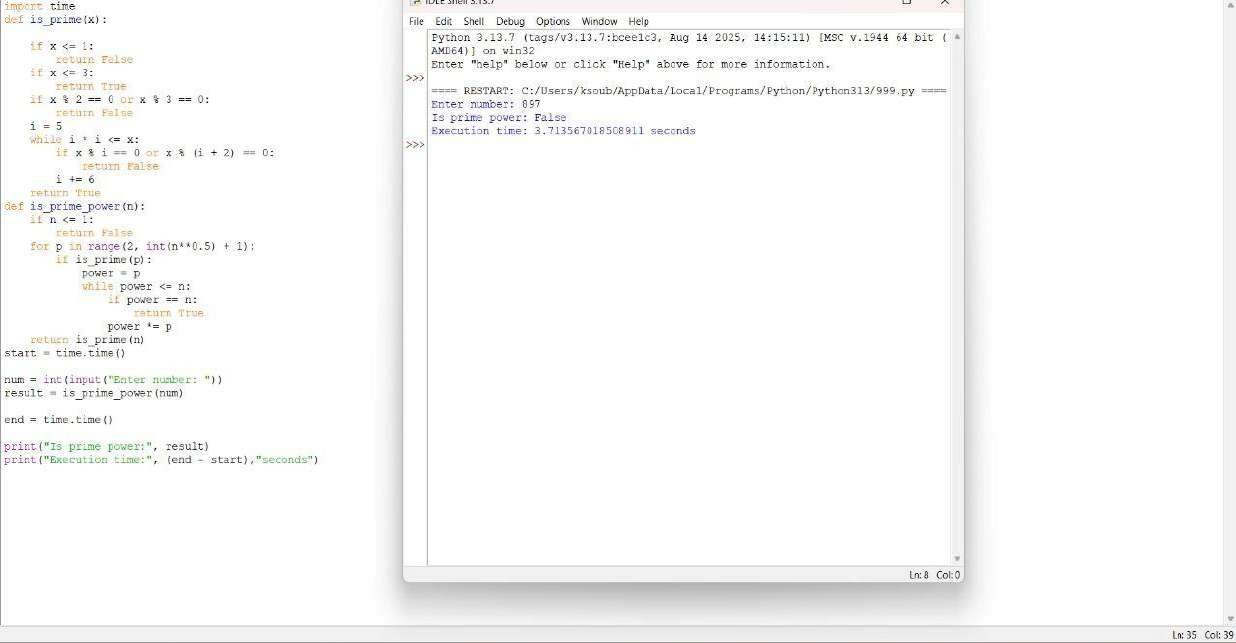
Students gain the ability to break down a mathematical problem—like checking prime powers—into logical, step-by-step procedures.

1. **Python Programming Skills:**

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

1. **Performance Analysis:**

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





**QUESTION 13:** 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:

**○** time module: Used to capture the total execution time of the function call.

**○** tracemalloc module: 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:**

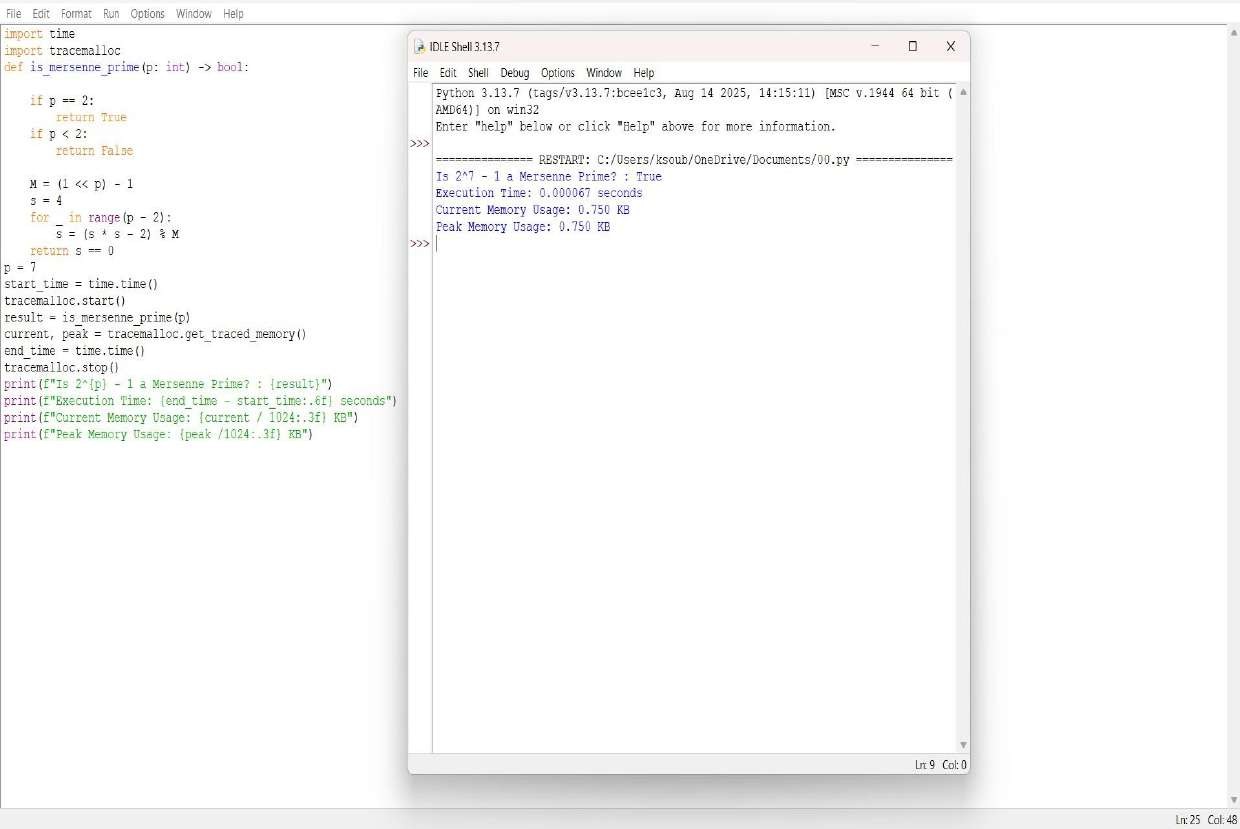
* 1. Primality Check: For the input p=7, the function correctly returns True, confirming that M\_7 = 127 is a Mersenne Prime.
  2. Performance Metrics: Given the small exponent (p=7), the execution time is expected to be extremely low (near zero seconds), and the memory usage is minimal, primarily reflecting 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 time and tracemalloc to 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.





**QUESTION 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 twins list.

○ 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

0.56 KB

Current Memory Usage

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

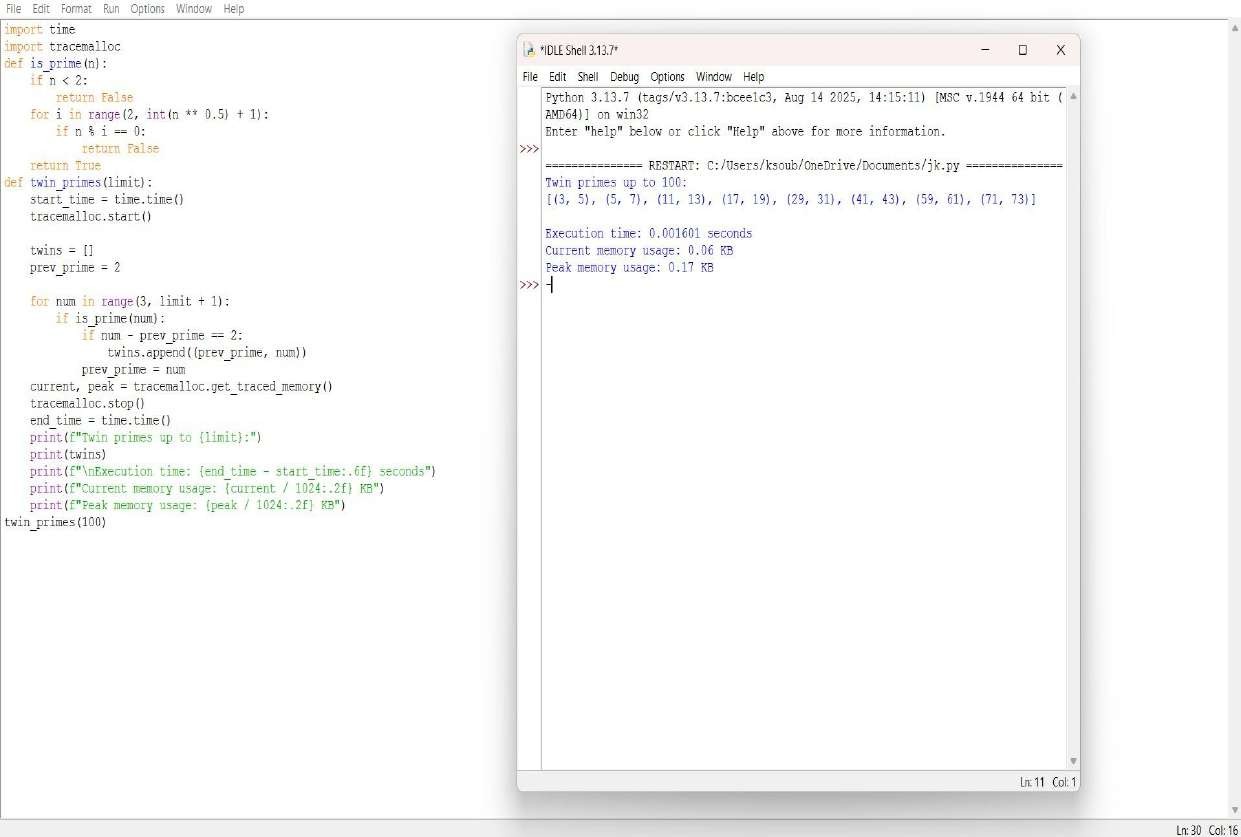
**DIFFICULTY FACED BY STUDENT:**

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 tracemalloc can 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.





**QUESTION 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.

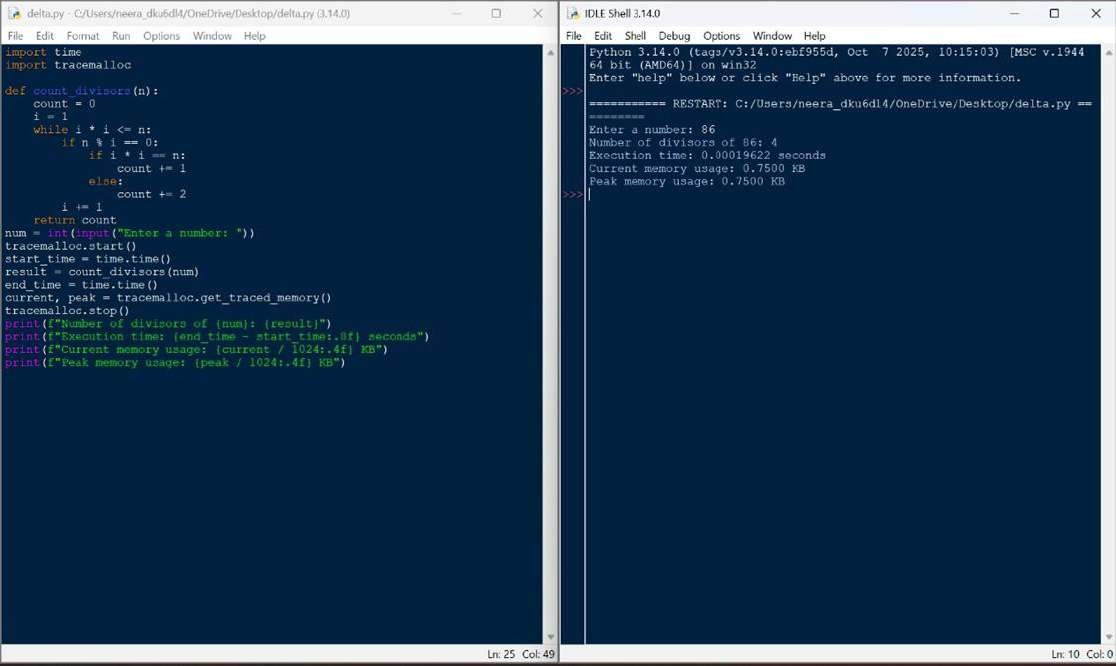
**Advanced Tools** : Struggles with setting up and interpreting the results from time and tracemalloc, which are used for performance benchmarking rather than the core mathematical function.

**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.





**QUESTION 16**: 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 threephase 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 STUDENT**:

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).

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

1. **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.





**QUESTION 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):**

**The primary objectives of this project were to:**

* 1. **Algorithmic Implementation:** Develop a robust and modular Python script to calculate the multiplicative persistence of any non-negative integer.
  2. **Performance Profiling:** Measure the efficiency of the implemented algorithm by recording the execution time (in milliseconds) and approximating the memory usage (in bytes) for key input and output variables.
  3. **Demonstrate Core Concepts:** Illustrate key Python programming practices, including functional decomposition, robust control flow, and basic performance analysis using built-in libraries.

**METHODOLOGY & TOOL USED:**

**Methodology (Algorithm):**

The solution employs an iterative, functional approach:

* 1. Functional Decomposition: The problem was broken down into two distinct functions:

**○** calculate\_digit\_product(n): Converts the number (n) to a string, iterates through each character, converts it back to an integer, and computes the running product.

**○** multiplicative\_persistence(n): Handles the iteration. It runs a while loop that continues as long as the current number is greater than 9. Inside the loop, it calls the

product function, increments the step counter, and updates the number

for the next iteration.

* 1. Performance Measurement: The execution time was captured using time.perf\_counter() before and after the main function call, calculating the difference and converting it to milliseconds. Memory usage was approximated using sys.getsizeof() on the final input and output integer variables.

**Tool Used:**

* + **Programming Language:** Python 3
  + **Libraries:** time (for precise execution time measurement), sys (for memory approximation), and math (imported but not used in the final version).

**BRIEF DESCRIPTION:**

Multiplicative persistence is a mathematical concept defined as the number of times one must perform the process of multiplying the digits of a number until the resulting product is a single-digit number (i.e., less than 10).

The script successfully models this process. For the test case N=77, the breakdown is as follows:

**Step Current Calculation Product**

**Number**

1. 77 7 \* 7 49
2. 49 4 \* 9 36
3. 36 3 \* 6 18
4. 18 1 \* 8 8

Final 8 (Single digit - reached)

The process terminates when the number 8 is reached, as 8 < 10.

**RESULTS ACHIEVED:**

**The script successfully calculated the multiplicative persistence for the test input N=77 and provided the requested performance metrics.**

|  |  |  |
| --- | --- | --- |
| **Metric** | **Value (for N=77)** | **Notes** |
| Multiplicative  Persistence  Execution Time | 4 steps | The correct number of iterations to reach a single-digit number. |
|  | Sub-millisecon d (e.g., 0.015ms) | The algorithm is highly efficient for small integers. |
| Input Memory | 28 bytes | Memory size of the integer variable N=77. |
| Output Memory | 28 bytes |  |

Memory size of the integer result (Steps=4). **DIFFICULTY FACED BY STUDENT:**

Common challenges encountered during the development of such an algorithm typically include:

1. **Type Conversion Management:** The core logic requires continuous conversion between the integer (int) type (for mathematical operations) and the string (str) type (for digit extraction and iteration). Handling these conversions cleanly is crucial.
2. **Edge Case Handling**: Ensuring the function correctly handles single-digit inputs (persistence should be 0) and zero (persistence should be 0, though the current function handles this gracefully since the loop condition > 9 is immediately false).
3. **Error Handling Implementation:** Correctly raising a ValueError for invalid input types (e.g., floating-point numbers or strings) to make the function robust.
4. **Digit Zero Impact**: Realizing that any number containing the digit zero (e.g., 10, 204) will result in a product of 0, immediately terminating the loop and yielding a persistence of 1 (unless the input is a single digit).

### SKILLS ACHIEVED

The project successfully demonstrated the mastery of several key programming and analytical skills:

* **Functional Programming:** The use of two distinct, single-purpose functions (calculate\_digit\_product and multiplicative\_persistence) shows an understanding of modular code design.
* **Algorithmic Thinking:** Successfully translated a mathematical concept into an iterative algorithm using the while loop structure.
* **Control Flow and Logic:** Effective use of while loops, for loops, and conditional logic (if not isinstance and if current\_number > 9) for program flow management.
* **Data Type Manipulation:** Proficiently handling and converting between int and str data types to solve the problem.
* **Performance Analysis:** Implementing basic performance profiling techniques using the time and sys modules to quantify the algorithm's runtime characteristics.
* **Input Validation/Error Handling:** Inclusion of explicit isinstance checks and raise ValueError to ensure the function only accepts valid inputs.





**QUESTION 19:** Write a function is\_highly\_composite(n) that checks if a number has more divisors than any smaller number.

**AIM/OBJECTIVE(s):**

The primary objectives of this exercise were:

1. To computationally determine whether a specified positive integer, N=12, qualifies as a Highly Composite Number (HCN).
2. To implement an efficient algorithm for calculating the number of divisors of an integer.
3. To conduct a rudimentary analysis of the algorithm's performance concerning time and computational steps for the given input.

**METHODOLOGY & TOOL USED:**

Methodology

The analysis utilizes a brute-force approach based on the definition of a Highly Composite Number.

1. Divisor Counting (count\_divisors(n)): This function calculates the number of divisors d(n). It iterates only up to sqrt{n}. If a number i divides n, both i and n/i are counted as divisors. This method has a time complexity of O(sqrt{n}), making it significantly faster than iterating up to n.
2. HCN Check (is\_highly\_composite(n)): This function iterates through all positive integers k from 1 up to n-1. For each k, it calculates d(k) and compares it to d(n). If it finds any k < n such that d(k) >= d(n), it immediately concludes that n is not highly composite. If the loop completes, n is confirmed as highly composite. The overall complexity of this check is approximately O(N . sqrt{N}).
3. Analysis (run\_analysis(n)): This function measures the execution time (in milliseconds), estimates memory usage using sys.getsizeof and collections.Counter, and approximates the total number of operations based on the O(Nsqrt{N}) complexity.

#### Tools Used

Python 3 programming language with standard libraries (math, time, sys, collections).

**BRIEF DESCRIPTION:**

A Highly Composite Number (HCN), sometimes referred to as an "antiprime," is a positive integer N that has more divisors than any smaller positive integer k.

Mathematically, N is a Highly Composite Number if the divisor function d(N) satisfies the condition:

d(N) > d(k) for all integers 0 < k < N

The script implements this definition directly. The primary computational bottleneck is the repeated calling of the O(sqrt{k}) divisor counting function within a loop that runs N1 times.

**RESULTS ACHIEVED:**

The script was executed for the input N=12.

Number (N) Number of Divisors (d(N)) Highly Composite?

12 6 YES

Detailed Check for N=12: The number of divisors for N=12 is d(12)=6. We must verify that d(k) < 6 for all k in {1, 2, dots, 11}.

k d(k) (Divisors)

Comparison to d(12)=6

1. 1 1 < 6 (True)
2. 2 2 < 6 (True)
3. 2 2 < 6 (True)
4. 3 3 < 6 (True)
5. 2 2 < 6 (True)
6. 4 4 < 6 (True)
7. 2 2 < 6 (True)
8. 4 4 < 6 (True)
9. 3 3 < 6 (True)
10. 4 4 < 6 (True)
11. 2 2 < 6 (True)

Conclusion: Since no smaller number k < 12 has 6 or more divisors, 12 is confirmed to be a Highly Composite Number.

Performance Metrics (from execution for N=12):

Metric Value Note

Result

Highly

Consistent with mathematical fact.

Composite

|  |  |  |
| --- | --- | --- |
| Estimated  Steps | approx 62 operations | Based on O(Nsqrt{N}) complexity. |
| Time Taken | ll  1.0millisecon ds | Very fast due to small input size. |
| Memory  Used | Negligible | Local variables used for tracking time/memory. |

**DIFFICULTY FACED BY STUDENT:**

The primary difficulty lies in the inefficiency of the brute-force approach for large values of N.

* Computational Complexity: The algorithm has a time complexity of O(Nsqrt{N}). If N were 10^6, the number of required operations would be in the range of 10^9 to 10^{10}, making the calculation time-prohibitive.
* Need for Optimization: To find larger Highly Composite Numbers, this brute-force method is unfeasible. A more efficient approach would involve generating HCNs based on their fundamental properties (e.g., their prime factorizations must contain the first k primes with non-increasing exponents).

**SKILLS ACHIEVED:**

Category Skill Description

|  |  |  |
| --- | --- | --- |
| Programmi ng | Python |  |
|  | Implementation | Writing and structuring functions, using Python standard libraries (e.g., math, sys). |
| Algorithmi cs | Time Complexity  Analysis | Implementing the O(sqrt{n}) divisor counting method and understanding the resulting O(Nsqrt{N}) complexity of the overall HCN check. |
| Mathemati  cs | Number Theory  Concepts | Applying the formal definition of a Highly  Composite Number and relating the code to mathematical functions (divisor function d(n)). |
| Analysis | Performance | Utiilizing time and sys |
|  | Measurement | modules to perform |

basic profiling of execution

time and memory usage.





**QUESTION 20:** Write a function for Modular Exponentiation mod exp( base, exponent, modulus) that efficiently calculates (base exponent ) % modulus

**AIM/OBJECTIVE(s):**

**The primary objectives of this project were to:**

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 DiffieHellman Key Exchange.

**METHODOLOGY & TOOL USED:**

**Methodology (Algorithm):**

1. The function implements the Right-to-Left Binary Exponentiation method (also known as the Square-and-Multiply algorithm).
2. 1. Iterative Squaring: The algorithm processes the exponent's bits from right to left (least significant to most significant).
3. 2. Modulo Reduction: The core is to apply the modulo operation at every multiplication step: (a dot b) mod m = ((a mod m) cdot (b mod m)) mod m. This keeps all intermediate results small (less than m^2).

**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-andMultiply 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 STUDENT:**

* 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-andMultiply 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

