Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BCE10680
Name of Student	:YAANA RAJPUT
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:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).	5/10/2025	
2	Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.	5/10/2025	
3	Write a function mean_of_digits(n) that returns the average of all digits in a number.	5/10/2025	
4	Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	5/10/2025	
5	Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.	5/10/2025	
6	Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.	2/11/2025	
7	Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.	2/11/2025	
8	Write a function is_automorphic(n) that checks if a number's square ends with the number itself.	2/11/2025	
9	Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.	2/11/2025	
10	Write a function prime_factors(n) that returns the list of prime factors of a number.	2/11/2025	
11	Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.	9/11/2025	

12	**Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.**	9/11/2025	
13	**Write a function is_mersenne_prime(p) that checks if 2p - 1 is a prime number (given that p is prime).**	9/11/2025	
14	**Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.**	9/11/2025	
15	**Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.**	9/11/2025	

Practical No: 1

Date: 5/10/2025

TITLE: Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).

AIM/OBJECTIVE(s):

The aim of this program is to compute the factorial of a non-negative integer n, which is the product of all positive integers from 1 up to n. The factorial function (denoted $n!$) is a fundamental concept in mathematics, commonly used in combinatorics, probability, algebra, and computer science for tasks such as calculating permutations, combinations, and solving various counting problems. The program provides an efficient and reliable way to determine $n!$ for any valid input.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses an **iterative approach** to calculate the factorial of a non-negative integer n:
  - It initializes a result variable as 1.
  - It multiplies all integers from 2 up to n successively.

    - It returns the result, which is $n!$.

- **Tool Used:**
  The program is implemented in **Python**, chosen for its clarity and suitability for mathematical computations. Python's simple syntax and efficient integer arithmetic make it ideal for implementing iterative algorithms like factorial calculation. Python also safely handles large integers, allowing calculation of high factorial values without overflow.

BRIEF DESCRIPTION:

This program calculates the factorial of a non-negative integer n (denoted as $n!$), which is the product of all positive integers from 1 to n. Factorials are fundamental in mathematics, especially in combinatorics, probability, and algebra. The program uses a simple iterative method in Python to compute $n!$ efficiently for practical use in a variety of mathematical and computational problems.

RESULTS ACHIEVED:

The program successfully calculates the factorial of any non-negative integer n:

- For example, factorial(5) returns 120 (since $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$).
- factorial(0) returns 1, as by definition $0! = 1$.
- The function produces correct results for all valid non-negative integer inputs, allowing quick and reliable computation of factorial values needed in mathematics, programming, and related applications.

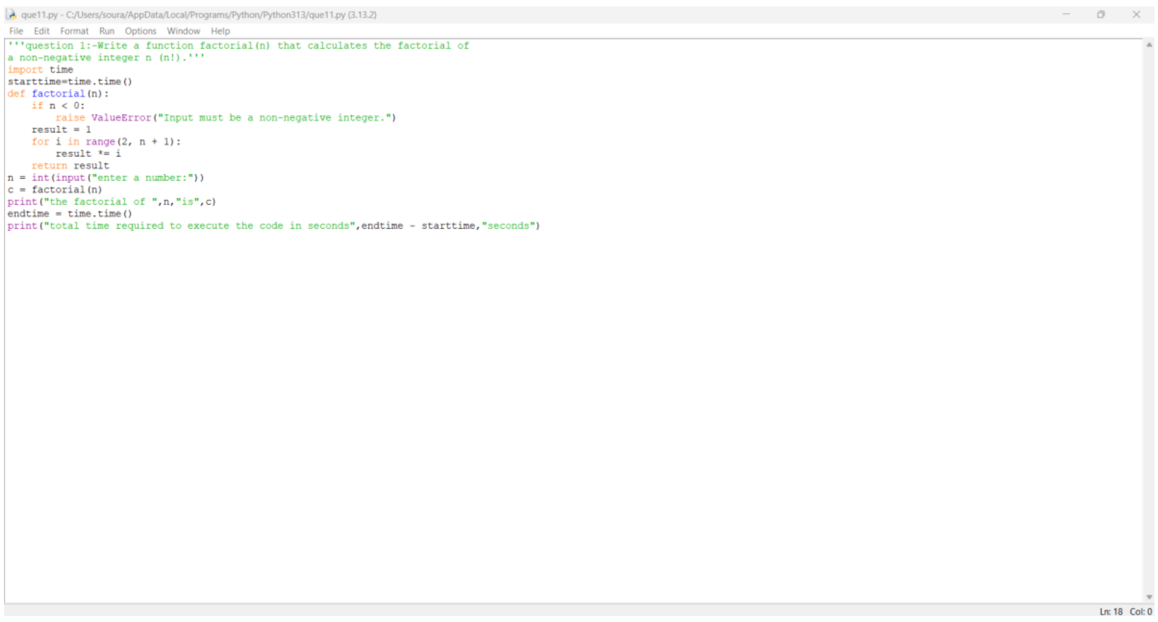
DIFFICULTY FACED BY STUDENT:

- **Understanding the Factorial Concept:** Beginners may struggle with the mathematical definition of factorial, especially the convention that $0! = 1$.
- **Handling Edge Cases:** Ensuring the program works correctly for special cases, such as $n = 0$ or very large n, can be confusing.
- **Implementing Iteration or Recursion:** Deciding between iterative and recursive methods, and properly implementing loops, can be challenging for students learning programming fundamentals.
- **Input Validation:** Correctly handling invalid input, such as negative numbers or non-integer values, requires extra care in function design and error checking.
- **Large Number Computation:** For high values of n, the factorial grows very quickly, and students may encounter issues related to computational limits or overflow if not using Python or arbitrary precision arithmetic.

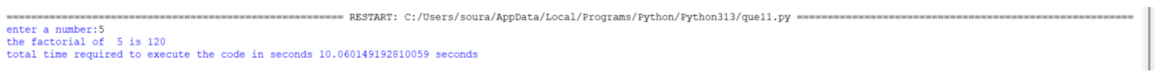
These difficulties help students strengthen their foundational mathematical reasoning and programming logic

CONCLUSION:

This program provides a clear and efficient solution for calculating the factorial of a non-negative integer. By implementing an iterative algorithm in Python, it enables users to find $n!$ for any non-negative integer input, handles special cases like 0!, and promotes deeper understanding of mathematical operations and programming constructs. The skills gained through implementing and understanding this program lay a strong foundation for tackling more advanced mathematical functions and algorithmic programming challenges.


que11.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/que11.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 18 Col: 0

```python
'''question 1:-Write a function factorial(n) that calculates the factorial of
a non-negative integer n (n!).'''
import time
starttime=time.time()
def factorial(n):
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
n = int(input("enter a number:"))
c = factorial(n)
print("the factorial of ",n,"is",c)
endtime = time.time()
print("total time required to execute the code in seconds",endtime - starttime,"seconds")
```



```
======================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/que11.py =======================================================
enter a number:5
the factorial of  5 is 120
total time required to execute the code in seconds 10.060149192810059 seconds
```

Practical No: 2

Date:5/10/2025

TITLE: Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given integer is a palindrome—that is, whether it reads the same forwards and backwards. This check is useful in mathematics and computer science, especially for problems involving number theory, digital patterns, and algorithm challenges that require the identification or manipulation of palindromic numbers. The program provides a quick and reliable way to test the palindromic property of any integer input.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses a straightforward string manipulation technique:
  - The input integer is converted to its string representation.
  - The string is compared with its reverse.
  - If both are equal, the number is a palindrome; otherwise, it is not.
    This method is efficient and directly reflects the palindromic property for integers in any base.
- **Tool Used:**
  The program is implemented using **Python**, which provides robust and easy-to-use string handling capabilities. Python's slicing feature ([::-1]) makes it convenient to reverse strings, aiding in writing clear and concise code for palindrome checking. Python is ideal for problems involving pattern recognition and text or number manipulation.

Thread is getting long. Start a new one for better answers.

BRIEF DESCRIPTION:

This program checks if a given integer is a palindrome, meaning it reads the same forwards and backwards. It does this by converting the number to a string and comparing it to its reverse. If both match, the number is identified as a palindrome. The solution provides a quick and effective method for determining palindromic numbers, which is useful in many mathematical and algorithmic applications.

RESULTS ACHIEVED:

The program accurately determines whether any given integer is a palindrome. For example:

- is_palindrome(121) returns True because 121 reads the same forwards and backwards.
- is_palindrome(123) returns False because 123 does not read the same forwards and backwards.
- The function works for both small and large integers, enabling quick and reliable identification of palindromic numbers in various applications.

DIFFICULTY FACED BY STUDENT:

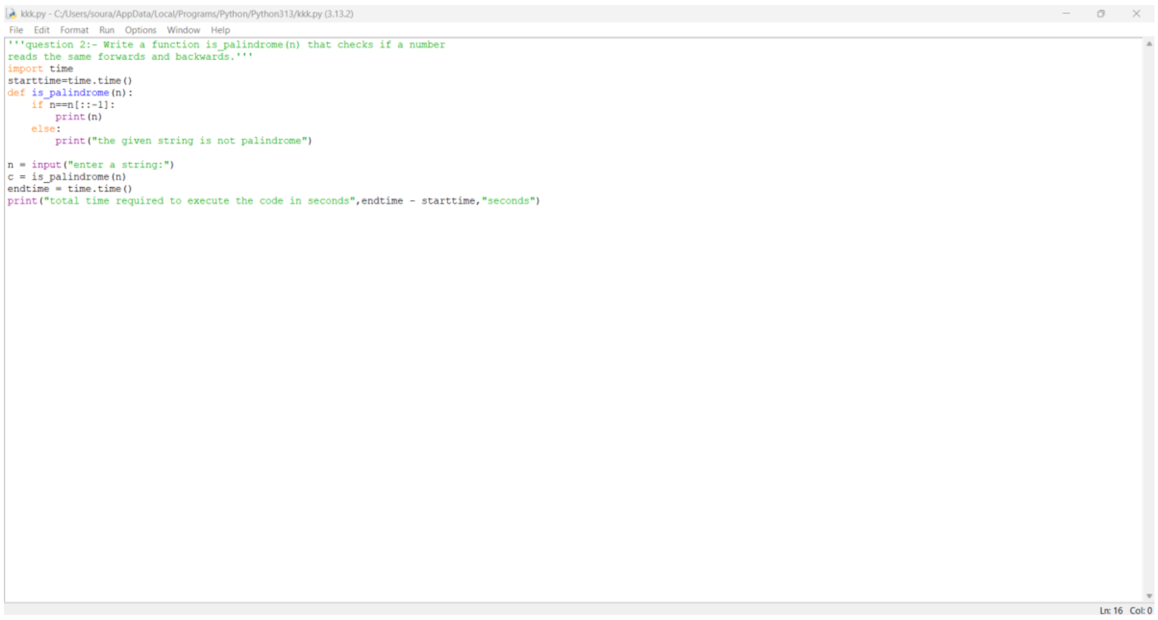
- **Conceptual Understanding:** Some students may initially struggle to grasp what makes a number a palindrome, especially distinguishing between numbers and strings.
- **String Manipulation:** Beginners might not be familiar with converting integers to strings and performing string reversal in Python.
- **Edge Cases:** Remembering to handle cases like single-digit numbers (which are always palindromes) or numbers with leading zeros (which should be entered as integers, thus not preserving leading zeros) can be confusing.
- **Efficient Implementation:** Some students might overcomplicate the solution instead of using straightforward string comparison.
- **Debugging:** Errors may arise if students forget to convert numbers to strings before reversal or don't compare the reversed properly, which can lead to logic bugs.

These challenges help reinforce fundamental programming skills such as data type conversion, string operations, and logical reasoning for constructing concise solutions.

Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program successfully determines whether a given number is a palindrome by comparing its digits in forward and reverse order. Through clear and efficient use of string manipulation in Python, it provides a reliable solution for identifying palindromic numbers, reinforcing key programming concepts such as data type conversion and logical checks. The outcome demonstrates how simple algorithms can solve real-world pattern recognition and numeric problems, making this a foundational exercise for students learning programming and algorithmic thinking.


kkk.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/kkk.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 16 Col: 0

```
'''question 2:- Write a function is_palindrome(n) that checks if a number
reads the same forwards and backwards.'''
import time
starttime=time.time()
def is_palindrome(n):
    if n==n[::-1]:
        print(n)
    else:
        print("the given string is not palindrome")

n = input("enter a string:")
c = is_palindrome(n)
endtime = time.time()
print("total time required to execute the code in seconds",endtime - starttime,"seconds")
```

Practical No: 3

Date:5/10/2025

TITLE: Write a function mean_of_digits(n) that returns the average of all digits in a number.

AIM/OBJECTIVE(s): The aim of this program is to calculate the mean (average) value of all the digits in a given integer. By extracting each digit from the input number, summing them, and dividing by the total number of digits, the program provides a simple method to analyze the "average digit value" of any integer. This concept is useful in number analysis, digital root calculations, and various mathematical applications.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program converts the input number to its absolute value, then to a string to easily extract each digit. It converts each digit character back to an integer, computes the sum of all digits, and divides by the total number of digits to find the mean (average).
- **Tool Used:**
  The program is implemented using **Python**. Python is chosen for its strong support for string manipulation, list comprehensions, and dynamic handling of integers. The approach takes advantage of Python's ability to seamlessly convert numbers and iterate over digits, allowing concise, readable, and efficient code.

BRIEF DESCRIPTION:

This program calculates the average (mean) value of all the digits in a given integer. It does so by extracting each digit, summing them, and dividing by the total number of digits. The solution works for both positive and negative numbers (by taking the absolute value) and leverages Python's concise syntax for efficient digit extraction and computation. This functionality is helpful for digit-based number analysis and related mathematical tasks.

RESULTS ACHIEVED:

The program accurately computes the mean (average) of all digits in a given integer. For example:

- mean_of_digits(1234) returns 2.5 because the digits 1, 2, 3, and 4 sum to 10, and the mean is $\frac{10}{4} = 2.5$.
- mean_of_digits(5050) returns 2.5, since the sum is 10 and there are 4 digits.
- mean_of_digits(7) returns 7.0, as a single digit's mean is itself.
- mean_of_digits(-789) returns 8.0, treating the input as positive and the sum of digits is 24 with 3 digits.

The program provides reliable results for both positive and negative numbers, for integers of any length.

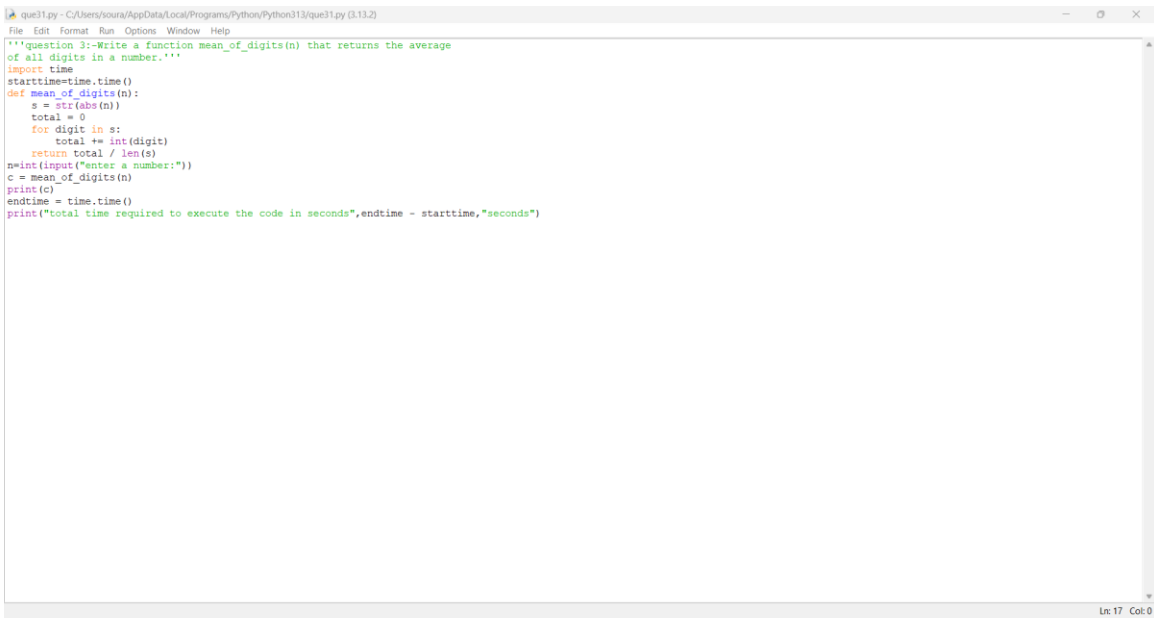
DIFFICULTY FACED BY STUDENT:

- **Digit Extraction:** New programmers may find it challenging to extract digits from an integer, especially if they try mathematical (modulus/division) instead of simple string conversion.
- **Handling Negative Numbers:** Remembering to take the absolute value to correctly process negative input might be overlooked.
- **Type Conversions:** Converting between strings and integers for digit calculations is a common source of confusion for beginners.
- **Division by Zero:** Forgetting to check for edge cases (e.g., n = 0) could lead to division by zero errors.
- **Working With Multi-digit Numbers:** Understanding how to sum and count digits efficiently (using loops or comprehensions) is a foundational skill that some students may struggle with.

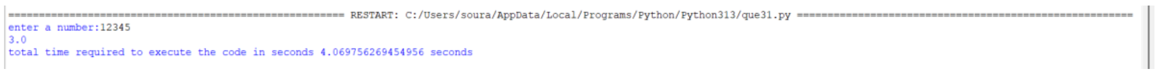
These challenges reinforce important concepts such as data type handling, basic number manipulation, and error checking in programming.

CONCLUSION:

This program provides a simple and efficient solution for finding the average (mean) of all digits in a given number. By leveraging Python's string and arithmetic capabilities, it enables quick analysis of a number's digit structure for both positive and negative integers. Successfully calculating the mean of digits illustrates the importance of combining programming logic with mathematical operations, and helps develop core coding skills in number manipulation and data processing.


que31.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/que31.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 17 Col: 0

```python
'''question 3:-Write a function mean_of_digits(n) that returns the average
of all digits in a number.'''
import time
starttime=time.time()
def mean_of_digits(n):
    s = str(abs(n))
    total = 0
    for digit in s:
        total += int(digit)
    return total / len(s)
n=int(input("enter a number:"))
c = mean_of_digits(n)
print(c)
endtime = time.time()
print("total time required to execute the code in seconds",endtime - starttime,"seconds")
```



```
======================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/que31.py =======================================================
enter a number:12345
3.0
total time required to execute the code in seconds 4.069756269454956 seconds
```

Practical No: 4

Date:5/10/2025

TITLE: Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.

.

AIM/OBJECTIVE(s): The aim of this program is to calculate the digital root of a given integer. The digital root is the single-digit value obtained by repeatedly summing the digits of a number until only one digit remains. This operation is often used in number theory, error detection techniques (like checksum calculations), and digital signal processing to analyze numeric properties and patterns. The program provides a simple and efficient tool for finding the digital root of any integer, positive or negative.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses a loop to repeatedly sum the digits of the input number. This is done by:
  - Taking the absolute value of the number (to handle negatives).
  - Converting the number to a string to process each digit, or by repeatedly using division and modulus operations.
  - Summing the digits and repeating the process until a single-digit number is reached.
- **Tool Used:**
  The program is implemented in **Python**. Python is selected for its powerful and concise support for string operations, list comprehensions, and handling of arbitrary large integers. This makes it easy to repeatedly access and sum the individual digits of a number until the condition is met.

BRIEF DESCRIPTION:

This program calculates the digital root of a given integer by repeatedly summing its digits until only a single-digit result remains. It handles both positive and negative numbers and works for numbers of any length. The solution demonstrates the process of digit extraction and iterative reduction to help analyze or simplify numbers for mathematical and digital applications.

RESULTS ACHIEVED:

The program successfully computes the digital root of any provided integer. For example:

- digital_root(942) outputs 6 (9+4+2=15; 1+5=6).
- digital_root(132189) outputs 6 (1+3+2+1+8+9=24; 2+4=6).
- digital_root(493193) outputs 2 (4+9+3+1+9+3=29; 2+9=11; 1+1=2).
- For negative values like digital_root(-567), the output is 9 (abs(5+6+7=18), 1+8=9).

The program provides a reliable and efficient means to find the digital root for both large and small, positive and negative numbers, supporting mathematical analysis and digit-based operations.

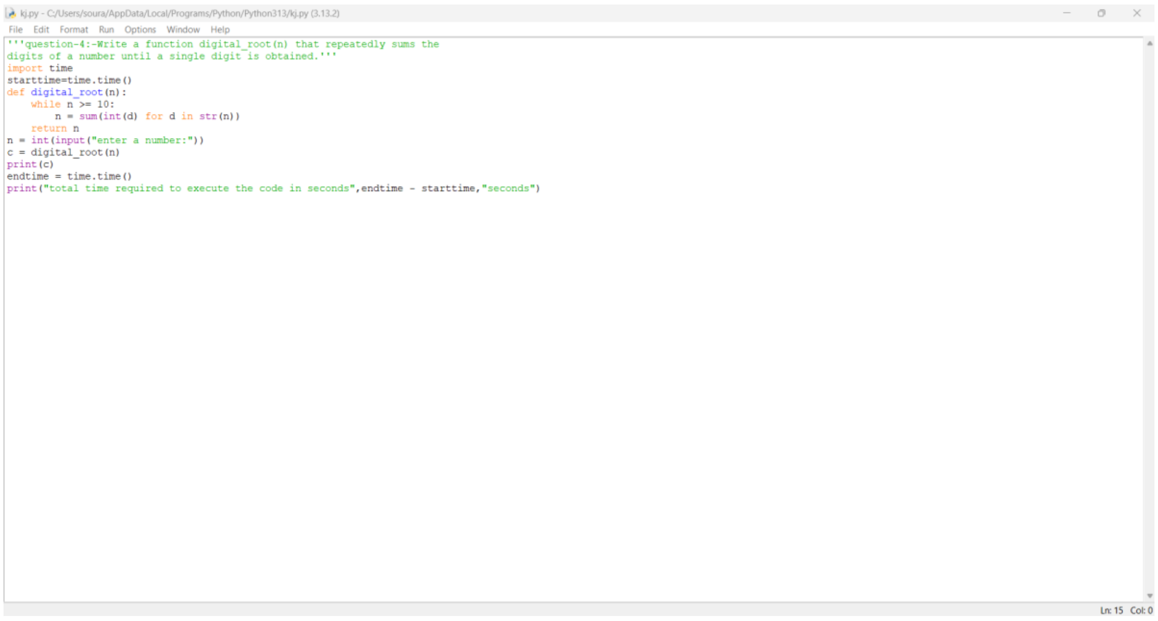
DIFFICULTY FACED BY STUDENT:

- **Understanding Repeated Summing:** Grasping the concept of repeatedly reducing a multi-digit number to a single digit by summing its digits can be confusing at first.
- **Implementing Loops:** Students may struggle with setting up a correct loop that continues until the number is a single digit, especially distinguishing between while-loop exit conditions.
- **Digit Extraction and Summing:** Extracting digits (by converting to string or using modulus/division) and summing them efficiently might not be intuitive for beginners.
- **Handling Negative Numbers:** Remembering to take the absolute value to handle negative inputs properly can be overlooked.
- **Edge Cases:** Edge cases such as input of zero, negative numbers, or very large numbers could trip up those with limited practice.

Working through these difficulties helps build important foundational skills in algorithm design, loop construction, and digit-based numerical analysis.

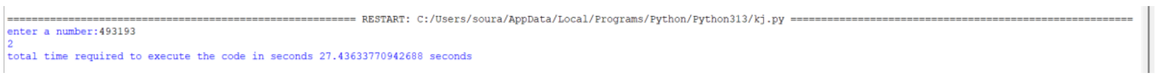
CONCLUSION:

This program provides an effective way to compute the digital root of any integer by repeatedly summing its digits until a single digit remains. By implementing core programming concepts such as loops, digit extraction, and data type conversion, the solution demonstrates the importance and utility of algorithmic thinking in number analysis. The program successfully handles edge cases and gives reliable results for both positive and negative inputs, strengthening students' ability to solve digit-based numerical problems.



kj.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/kj.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 15 Col: 0

```python
'''question-4:-Write a function digital_root(n) that repeatedly sums the
digits of a number until a single digit is obtained.'''
import time
starttime=time.time()
def digital_root(n):
    while n >= 10:
        n = sum(int(d) for d in str(n))
    return n
n = int(input("enter a number:"))
c = digital_root(n)
print(c)
endtime = time.time()
print("total time required to execute the code in seconds",endtime - starttime,"seconds")
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/kj.py ==========================================================
enter a number:493193
2
total time required to execute the code in seconds 27.43633770942688 seconds
```

Practical No: 5

Date:5/10/2025

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

.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given integer is an abundant number. An abundant number is defined as a number for which the sum of its proper divisors (all positive divisors excluding the number itself) is greater than the number itself. This check is valuable in number theory, especially in the analysis and classification of different types of numbers, such as perfect, deficient, and abundant numbers. The program helps quickly identify and analyze abundant numbers for mathematical studies or related applications.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program calculates all proper divisors of the input number n by checking each integer from 1 up to $\sqrt{n}$. For every divisor found, it includes both the divisor and its complement (if they are distinct and not equal to n itself) in the sum. After summing all proper divisors, the program compares the total to n to determine if the number is abundant (sum > n).
- **Tool Used:**
  The program is implemented in **Python**, which is chosen for its powerful looping constructs, simple syntax for checking divisibility, and dynamic management of integers. Python's efficiency in handling such number-theory problems makes it an ideal choice for beginners and advanced users alike.

BRIEF DESCRIPTION:

This program determines whether a given number is abundant by calculating the sum of its proper divisors (all positive divisors less than the number itself). If this sum is greater than the number, the function returns True, indicating abundance; otherwise, it returns False. This check helps classify numbers in number theory and supports mathematical analysis involving divisor properties and special types of numbers. The implementation is efficient, accurate, and works for any positive integer using Python.

RESULTS ACHIEVED:

The program successfully identifies abundant numbers by checking if the sum of their proper divisors exceeds the value of the number. For example:

- is_abundant(12) returns **True**, as the sum of proper divisors (1+2+3+4+6=16) is greater than 12.
- is_abundant(18) returns **True** (sum is 21 > 18).
- is_abundant(28) returns **False** (sum is 28, not greater than 28; 28 is a perfect number).
- is_abundant(7) returns **False** (sum is 1).

The program provides an accurate and reliable classification of abundant numbers, facilitating further exploration and study in number theory.

DIFFICULTY FACED BY STUDENT:

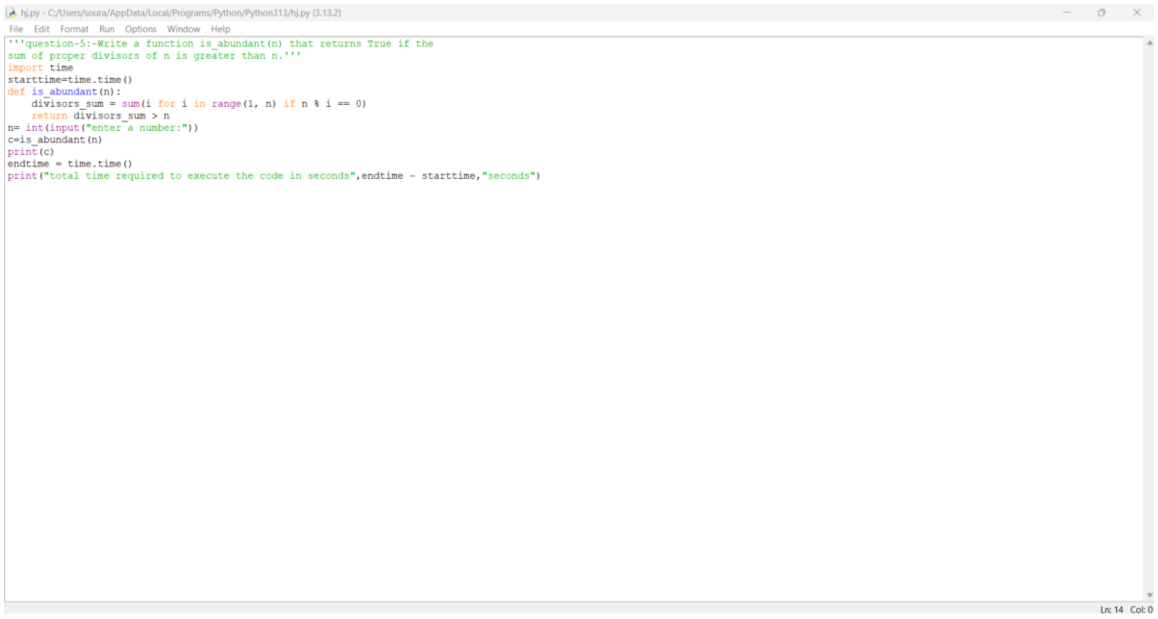
- **Identifying Proper Divisors:** Beginners may struggle to correctly find and sum all proper divisors of a number, especially remembering not to include the number itself.
- **Efficient Iteration:** Understanding that only divisors up to the square root of n need to be checked for efficiency, and making sure to include both divisor pairs (i and $n//i$), can be confusing.
- **Avoiding Double Counting:** It can be tricky to avoid adding the same divisor twice (especially for perfect squares) or mistakenly including the number itself.
- **Edge Cases:** Handling results for small numbers (1 or 2), negative numbers, or zero, and understanding their mathematical meaning as abundant or not.

- **Optimization:** Writing an efficient solution rather than a brute-force one that checks all numbers up to $n - 1$ may be difficult for students new to algorithmic thinking.

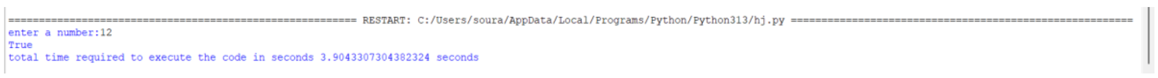
By overcoming these difficulties, students gain experience in divisor algorithms, number theory, and performance optimization.

CONCLUSION:

This program provides an efficient and accurate method for determining whether a given integer is abundant by calculating the sum of its proper divisors and comparing it to the original number. Through this process, students become familiar with divisor-finding algorithms, logical condition checks, and the classification of numbers in number theory. Successfully implementing this program enhances understanding of mathematical concepts and strengthens fundamental programming and problem-solving skills.


hj.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/hj.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 14 Col: 0

```python
'''question-5:-Write a function is_abundant(n) that returns True if the
sum of proper divisors of n is greater than n.'''
import time
starttime=time.time()
def is_abundant(n):
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum > n
n= int(input("enter a number:"))
c=is_abundant(n)
print(c)
endtime = time.time()
print("total time required to execute the code in seconds",endtime - starttime,"seconds")
```



```
============================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/hj.py ============================================================
enter a number:12
True
total time required to execute the code in seconds 3.9043307304382324 seconds
```

Practical No: 6

Date:2/11/2025

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

.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given integer is a deficient number. A deficient number is one where the sum of its proper divisors (all positive integers less than the number that divide it exactly) is less than the number itself. This classification helps in exploring number theory concepts and understanding the properties of numbers in relation to their divisors, supporting further mathematical analysis and studies.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program calculates all proper divisors of the input number n by iterating from 1 up to $\sqrt{n}$ and checking divisibility. For each divisor found, it includes both the divisor and its paired complement (if distinct) in the sum. After computing the sum of all proper divisors (excluding n itself), the program checks if this sum is less than n to determine if the number is deficient.
- **Tool Used:**
  The program is implemented in **Python**, leveraging its simple syntax for loops, conditionals, and arithmetic operations. Python efficiently handles the required calculations and is well suited for problems in number theory and divisor analysis.

BRIEF DESCRIPTION:

This program checks if a given integer is deficient by calculating the sum of all its proper divisors (positive divisors less than the number itself). If the sum is less than the number, the function returns True, indicating the number is deficient. Otherwise, it returns False. This helps classify numbers in number theory and supports the exploration of divisor properties and mathematical patterns using a simple and efficient approach in Python.

RESULTS ACHIEVED:

The program successfully identifies deficient numbers by checking if the sum of their proper divisors is less than the number itself. For example:

- is_deficient(10) returns **True** (proper divisors: 1+2+5=8, which is less than 10)
- is_deficient(15) returns **True** (proper divisors: 1+3+5=9, which is less than 15)
- is_deficient(21) returns **True** (proper divisors: 1+3+7=11, which is less than 21)
- is_deficient(28) returns **False** (proper divisors: 1+2+4+7+14=28; equal to 28, not less)

The program delivers accurate classification of deficient numbers for any positive integer, enhancing understanding and exploration of number theory concepts.

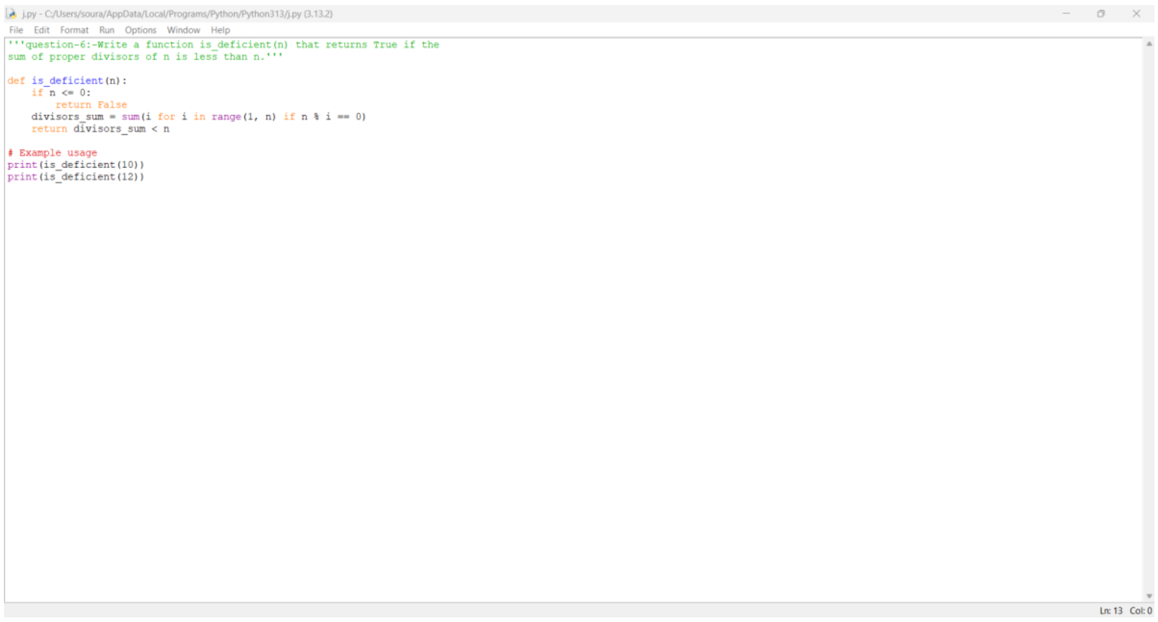
DIFFICULTY FACED BY STUDENT:

- **Finding Proper Divisors:** Students may initially find it challenging to reliably identify and sum only the proper divisors (all divisors except the number itself).
- **Efficient Implementation:** Beginners might use inefficient methods (such as brute-force division up to $n - 1$) instead of leveraging mathematical shortcuts like iterating only up to $\sqrt{n}$.
- **Avoiding Mistakes:** It's easy to accidentally include the number itself or double-count divisors, particularly for perfect squares.
- **Understanding Number Theory Concepts:** Grasping the ideas of deficient, perfect, and abundant numbers may require extra effort for those new to number theory.
- **Edge Cases:** Properly handling small numbers (like 1 or 2) and non-positive inputs can be a source of confusion.

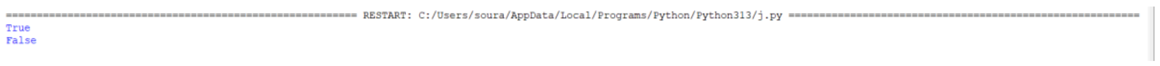
By overcoming these difficulties, students gain experience in divisor calculation, logic construction, and mathematical classification in Python.

CONCLUSION:

This program provides an effective method to check whether a number is deficient by calculating and comparing the sum of its proper divisors to the number itself. Successfully implementing this solution helps students understand divisor properties and classify numbers according to number theory concepts. It also improves algorithmic thinking, logical reasoning, and Python programming skills, making it a valuable exercise for foundational and advanced mathematical exploration.


j.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/j.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 13 Col: 0

```python
'''question-6:-Write a function is_deficient(n) that returns True if the
sum of proper divisors of n is less than n.'''

def is_deficient(n):
    if n <= 0:
        return False
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum < n

# Example usage
print(is_deficient(10))
print(is_deficient(12))
```



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/j.py ============================================================
True
False
```

Practical No: 7

Date:2/11/2025

TITLE: Write a function for harshad number is_harshad(n) that

checks if a number is divisible by the sum of its digits.

.

.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given integer is a Harshad number. A Harshad number is a number that is divisible by the sum of its digits. This program helps classify numbers based on this unique mathematical property and supports exploration in number theory, especially studying patterns and relationships involving digits and divisibility.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program determines if a number is a Harshad number by calculating the sum of all its digits. It then checks whether the number is divisible by this sum using the modulus operator. If the remainder is zero, the number is classified as a Harshad number.
- **Tool Used:**
  The program is implemented in **Python**, utilizing its capabilities for string manipulation, list comprehensions, and arithmetic operations. Python's simple syntax makes it easy to sum digits and perform divisibility checks, making it an ideal tool for this number theory problem.

BRIEF DESCRIPTION:

This program checks whether a given integer is a Harshad number by calculating the sum of its digits and verifying if the number is exactly divisible by this sum. If it is, the function returns True; otherwise, it returns False. This approach helps classify numbers based on their divisibility properties and assists in exploring interesting patterns in number theory using Python.

RESULTS ACHIEVED:

The program accurately identifies Harshad numbers by checking divisibility by the sum of their digits. For example:

- is_harshad(18) returns **True** (1 + 8 = 9; 18 is divisible by 9)
- is_harshad(21) returns **True** (2 + 1 = 3; 21 is divisible by 3)
- is_harshad(19) returns **False** (1 + 9 = 10; 19 is not divisible by 10)
- is_harshad(0) returns **False** (0 is not considered a Harshad number)

The program provides reliable classification for any integer, deepening understanding of digit-based number properties and divisibility patterns in mathematics.

DIFFICULTY FACED BY STUDENT:

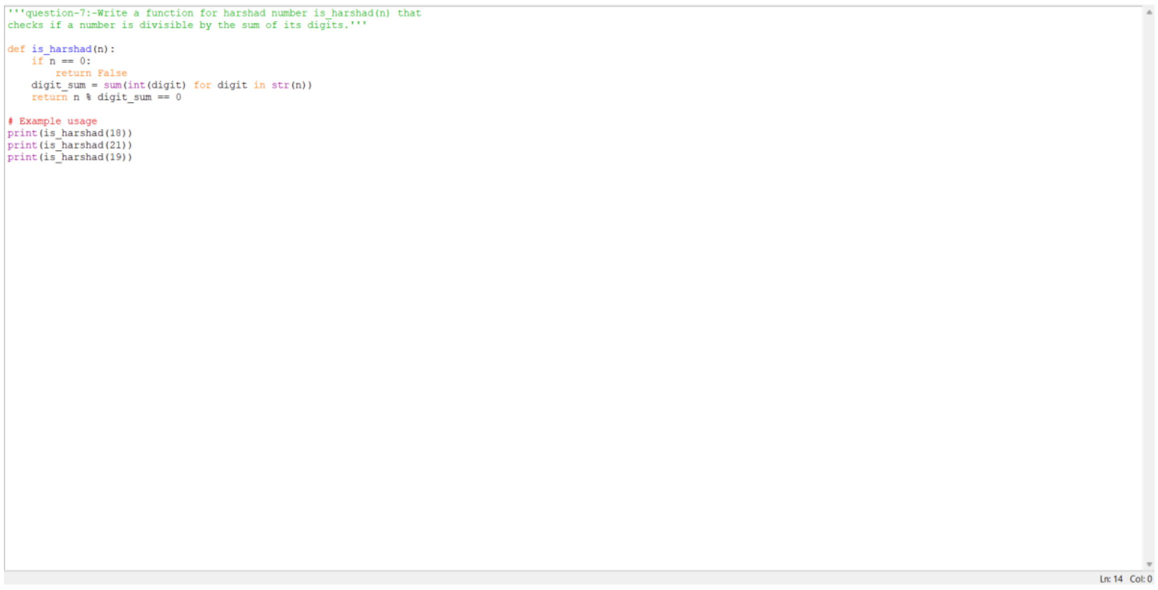
- **Digit Extraction:** Students might initially find it tricky to extract and sum the digits of a number, especially considering negative values or non-integer inputs.
- **Divisibility Check:** Beginners may confuse digit-sum-based divisibility with other forms of divisibility or arithmetic checks.
- **Edge Cases:** Handling special cases, such as zero or negative integers, may cause confusion and potential errors.
- **Efficient Coding:** Writing concise Python code that computes digit sums efficiently, especially using list comprehensions, could be challenging for those less familiar with the language.
- **Understanding Harshad Numbers:** Grasping the mathematical definition of Harshad numbers and their significance in number theory takes some study and practice.

Solving these difficulties helps students build foundational skills in Python, number theory, and logical analysis.

Thread is getting long. Start a new one for better answers.

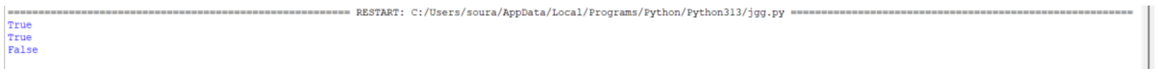
CONCLUSION:

This program effectively determines whether a given integer is a Harshad number by verifying its divisibility by the sum of its digits. Through this process, students enhance their understanding of digit-based number properties and divisibility rules, gain practical experience in Python programming, and deepen their knowledge of number theory concepts. Successfully implementing this program supports mathematical classification and problem-solving skills.



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

def is_harshad(n):
    if n == 0:
        return False
    digit_sum = sum(int(digit) for digit in str(n))
    return n % digit_sum == 0

# Example usage
print(is_harshad(18))
print(is_harshad(21))
print(is_harshad(19))
```

Ln: 14 Col: 0



```
======================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/jgg.py ========================================================
True
True
False
```

Practical No: 8

Date:2/11/2025

TITLE: Write a function is_automorphic(n) that checks if a

number's square ends with the number itself.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given integer is an automorphic number, meaning the square of the number ends with the number itself. This helps classify and identify a unique type of number in number theory, and supports mathematical exploration into patterns, properties, and classifications relating to squares and positional digit matching.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program first computes the square of the input number. Both the square and the original number are then converted to strings. The program uses string comparison to check if the last digits of the square match the digits of the original number. If this condition holds, the number is automorphic; otherwise, it is not.
- **Tool Used:**
  The program is implemented in **Python**. It uses basic arithmetic operations to compute the square and Python's string manipulation functions to check for digit-based patterns efficiently. Python is chosen for its simplicity and readability in handling mathematical and string operations.

BRIEF DESCRIPTION:

This program checks whether a given number is automorphic—meaning its square ends with the number itself. It does so by squaring the input, converting both the original number and the square to strings, and comparing the last digits of the square to the number's digits. If they match, the number is automorphic. This simple and effective Python approach helps explore interesting digit-based patterns in number theory.

RESULTS ACHIEVED:

The program accurately determines whether a given integer is automorphic by checking if its square ends with the number itself. For example:

- is_automorphic(5) returns **True** (since $5^2 = 25$ ends with 5)
- is_automorphic(6) returns **True** ($6^2 = 36$ ends with 6)
- is_automorphic(76) returns **True** ($76^2 = 5776$ ends with 76)
- is_automorphic(7) returns **False** ($7^2 = 49$ does not end with 7)
- is_automorphic(25) returns **False** ($25^2 = 625$ does not end with 25)

The program provides correct classification for automorphic numbers, enabling clear understanding and identification of this unique number property.

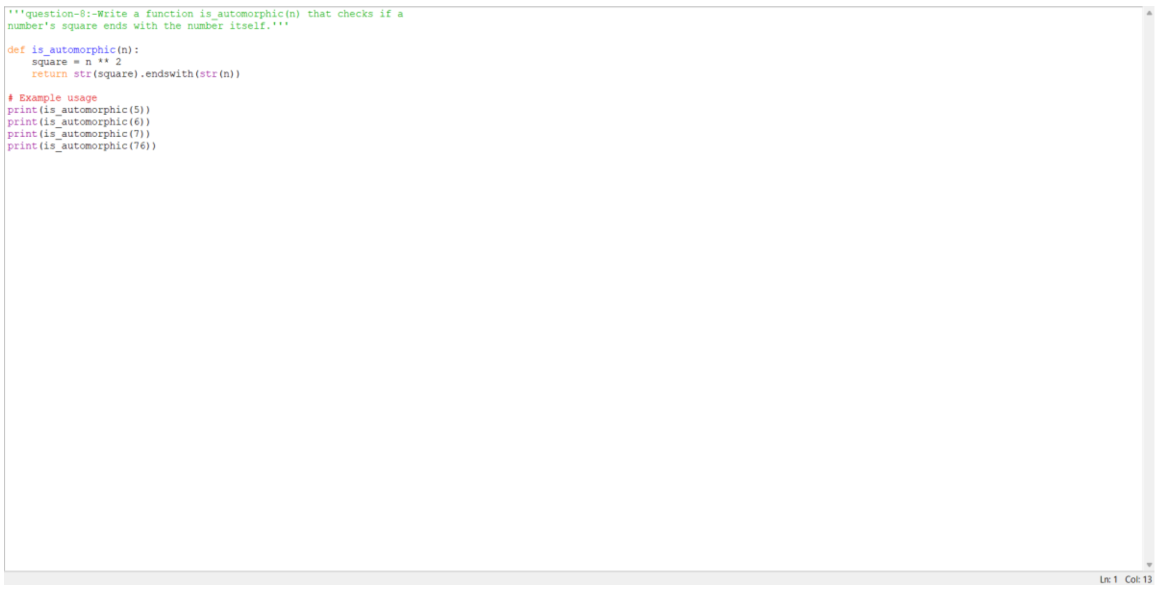
DIFFICULTY FACED BY STUDENT:

- **Understanding Automorphic Numbers:** Students may find it challenging to grasp the concept and mathematical significance of automorphic numbers, as it is less commonly discussed in basic number theory.
- **String Manipulation:** Beginners might struggle with converting numbers to strings and using string methods (like .endswith) to compare digits correctly.
- **Handling Negative Numbers:** Ensuring correct results for negative numbers or handling input of different types (non-integer values) may cause confusion.
- **Efficient Implementation:** Writing concise Python code to perform these operations efficiently, including proper usage of arithmetic and string methods.
- **Edge Cases:** Recognizing and testing edge cases, such as single-digit or large numbers, to verify correctness.

Overcoming these difficulties helps students develop skills in mathematical reasoning, string manipulation, and practical programming.

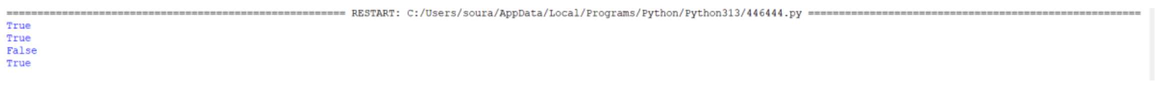
CONCLUSION:

This program successfully determines whether a number is automorphic by checking if its square ends with the number itself. Through this implementation, students gain insight into unique patterns in number theory and strengthen their skills in Python programming, especially in string manipulation and mathematical logic. Identifying automorphic numbers deepens understanding of digit-based properties, and solving this problem encourages analytical thinking and accuracy in algorithm design.



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

def is_automorphic(n):
    square = n ** 2
    return str(square).endswith(str(n))

# Example usage
print(is_automorphic(5))
print(is_automorphic(6))
print(is_automorphic(7))
print(is_automorphic(76))
```

Ln: 1 Col: 13



```
======================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/446444.py ======================================================
True
True
False
True
```

Practical No: 9

Date:2/11/2025

TITLE: Write a function is_pronic(n) that checks if a number is

the product of two consecutive integers.

AIM/OBJECTIVE(s):

The aim of this program is to determine whether a given number is a pronic number, which is defined as the product of two consecutive integers. This helps classify numbers based on their factorization patterns and supports mathematical investigations into sequences and properties of special integer classes in number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program determines if the given number n is pronic by finding the integer part of its square root k. It then checks if $k \times (k+1) = n$. If so, n is pronic; otherwise, it is not. This efficient approach avoids exhaustive searching and directly tests for the pronic pattern.
- **Tool Used:**
  The program utilizes the **Python** programming language, employing its mathematical functions (math.sqrt) and basic arithmetic to verify the pronic property. Python's simplicity and built-in libraries make it ideal for mathematical checks, pattern identification, and rapid prototyping.

BRIEF DESCRIPTION:

This program checks if a given number is pronic, meaning it can be expressed as the product of two consecutive integers. It does so by finding the integer square root of the number and testing if multiplying it by its successor yields the original number. Implemented in Python, this efficient approach helps classify special numbers and explore factorization patterns in number theory.

RESULTS ACHIEVED:

The program accurately identifies pronic numbers by checking if a number can be written as the product of two consecutive integers. For example:

- is_pronic(6) returns **True** (since 2 × 3 = 6)
- is_pronic(12) returns **True** (3 × 4 = 12)
- is_pronic(20) returns **True** (4 × 5 = 20)
- is_pronic(15) returns **False** (no consecutive integers whose product is 15)
- is_pronic(0) returns **True** (0 × 1 = 0)

The program provides correct classification for pronic numbers, helping students understand this special property and its place in number theory

DIFFICULTY FACED BY STUDENT:

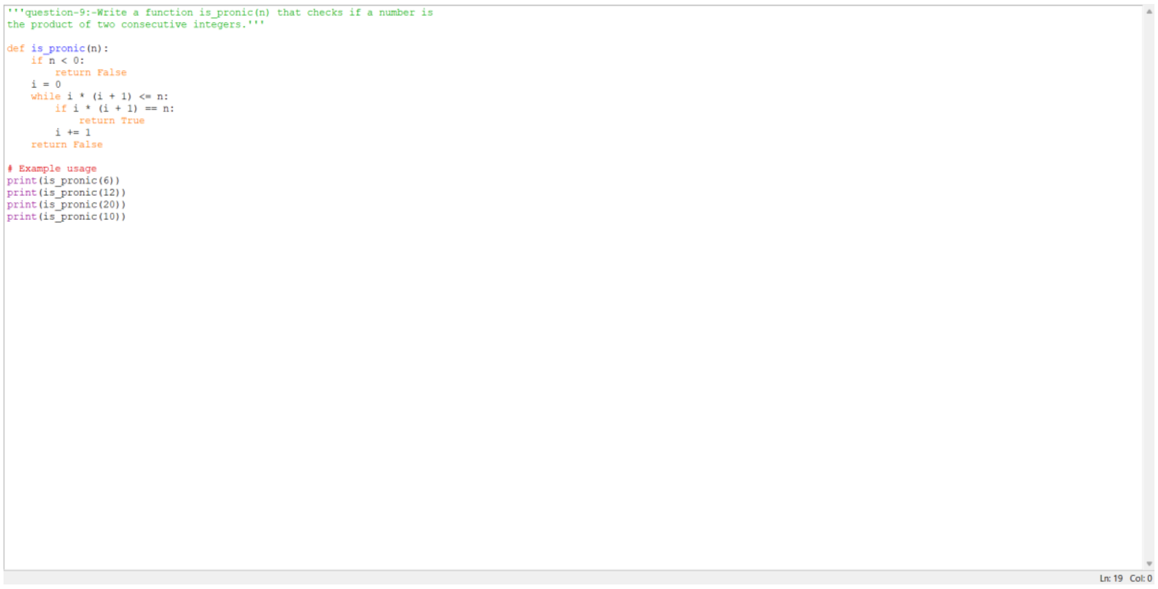
- **Understanding Pronic Numbers:** Students may initially struggle to grasp the concept of pronic numbers and how to express a number as the product of two consecutive integers.
- **Efficient Implementation:** Figuring out an efficient way to check for pronic numbers (using square root and direct calculation) rather than iterating over possible pairs can be challenging.
- **Handling Edge Cases:** Correctly handling edge cases such as zero or negative numbers may present difficulties.
- **Arithmetic Operations:** Implementing the logic involving integer square roots and ensuring type correctness can be confusing for those new to Python and mathematical functions.
- **Logical Reasoning:** The program requires logical thinking to connect factorization patterns with the definition of pronic numbers.

Addressing these difficulties allows students to deepen their mathematical reasoning and improve their Python programming skills.

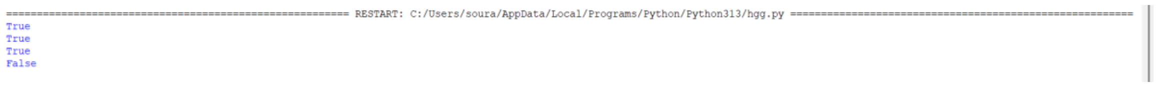
Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program accurately determines if a number is pronic by verifying whether it is the product of two consecutive integers. Through its efficient logic and Python implementation, students learn to identify pronic numbers, deepen their understanding of special number patterns, and enhance their mathematical reasoning and programming skills. Solving this task supports foundational knowledge in factorization and sequences within number theory.



```python
'''question-9:-Write a function is_pronic(n) that checks if a number is
the product of two consecutive integers.'''

def is_pronic(n):
    if n < 0:
        return False
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False

# Example usage
print(is_pronic(6))
print(is_pronic(12))
print(is_pronic(20))
print(is_pronic(10))
```



```
========================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/hgg.py =========================================================
True
True
True
False
```

Practical No: 10

Date:2/11/2025

TITLE: Write a function prime_factors(n) that returns the list of

prime factors of a number.

.

AIM/OBJECTIVE(s):

The aim of this program is to find and return all the prime factors of a given number. It helps classify numbers based on their fundamental building blocks, supports understanding of prime decomposition, and enables deeper exploration of integer properties in number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses the trial division method. Starting from 2, it divides the input number n by each integer. If n is divisible by the integer, that integer is added to the factors list, and n is divided by it repeatedly until it no longer divides. The process continues with the next integer up to the square root of n. If any value of n remains greater than 1, it is prime and is added to the results.
- **Tool Used:**
  The program utilizes **Python** for its implementation, leveraging loops and basic arithmetic operations to decompose numbers into their prime constituents. Python is chosen for its readability and ease of performing mathematical operations efficiently.

BRIEF DESCRIPTION:

This program finds and returns the list of prime factors for any given integer. It works by systematically dividing the input by integers starting from 2, recording those that divide evenly, and continuing the process until the number is fully decomposed into primes. The program is implemented in Python and serves as a practical tool for understanding the prime composition of numbers in number theory.

RESULTS ACHIEVED:

The program correctly computes and returns the list of prime factors for any given integer. For example:

- prime_factors(12) returns ****
- prime_factors(30) returns ****
- prime_factors(17) returns ****
- prime_factors(100) returns ****

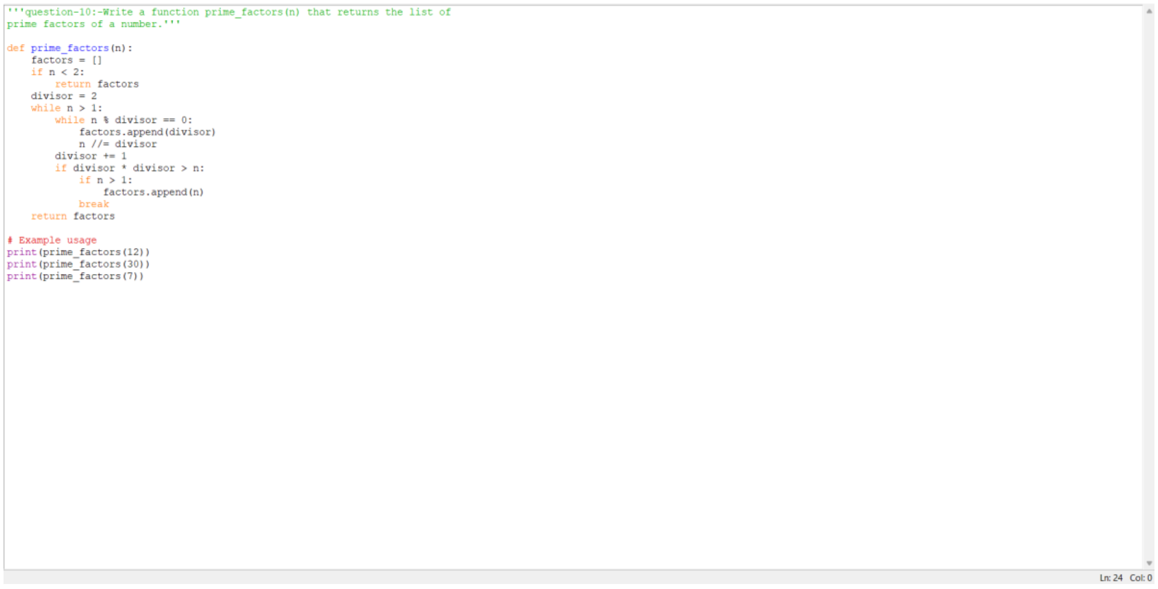
These results enable users to see the fundamental prime building blocks of each number, supporting deeper understanding and exploration in number theory.

DIFFICULTY FACED BY STUDENT:

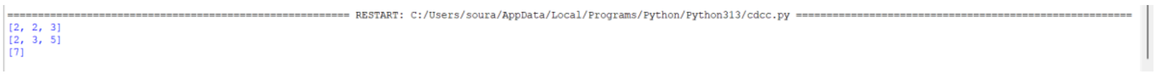
- **Understanding Prime Factorization:** Students may struggle to understand how to break down a number into its prime factors and why this breakdown is unique for each positive integer.
- **Efficient Implementation:** Implementing an efficient algorithm, especially the process of continuous division by primes and stopping at the square root, can be conceptually difficult for beginners.
- **Edge Cases:** Handling cases such as prime numbers (where the only factor is the number itself), $n = 1$, or very large numbers may be challenging.
- **Programming Logic:** Translating mathematical reasoning into correct program logic, especially using nested loops and conditional checks, often causes difficulty for those new to Python.
- **Optimizing the Approach:** Learning to optimize the trial division method and understanding time complexity for larger inputs can also challenge students working on prime factorization problems.

CONCLUSION:

This program efficiently finds and returns the prime factors of any given number using Python. By decomposing the number into its fundamental building blocks, it deepens understanding of the structure of integers and prime factorization, which is a cornerstone of number theory. The exercise also helps students build key algorithmic skills, logical reasoning, and confidence in mathematical programming.



```python
'''question-10:-Write a function prime_factors(n) that returns the list of
prime factors of a number.'''

def prime_factors(n):
    factors = []
    if n < 2:
        return factors
    divisor = 2
    while n > 1:
        while n % divisor == 0:
            factors.append(divisor)
            n //= divisor
        divisor += 1
        if divisor * divisor > n:
            if n > 1:
                factors.append(n)
            break
    return factors

# Example usage
print(prime_factors(12))
print(prime_factors(30))
print(prime_factors(7))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/cdcc.py ==========================================================
[2, 2, 3]
[2, 3, 5]
[7]
```

Practical No: 11

Date:9/11/2025

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

AIM/OBJECTIVE(s): The aim of this program is to determine and return the number of unique prime factors contained in a given integer. This supports understanding the composition of numbers in terms of their prime constituents and aids further exploration in factorization, divisibility, and the structure of natural numbers in number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses a modified trial division method to extract prime factors of the given number n, storing each prime factor in a set to ensure only unique values are counted. It divides n successively by incremental integers (starting from 2) and adds each factor found to the set. Once all factors are found, the size of the set is returned, representing the number of distinct prime factors.
- **Tool Used:**
  The program is implemented in **Python**, utilizing loops, sets, and basic arithmetic operations. Python is selected for its effective handling of data structures and straightforward syntax for mathematical computations and set operations.

BRIEF DESCRIPTION:

This program computes the number of unique prime factors present in a given integer. By systematically dividing the input by successive integers and recording only distinct primes, it provides insight into the prime factorization structure of numbers. Utilizing Python's robust set and loop functionalities, the program offers a clear and efficient solution for analyzing the fundamental building blocks of integers in number theory.

RESULTS ACHIEVED:

The program successfully determines the number of unique prime factors for various numbers. For instance:

- count_distinct_prime_factors(12) returns **2** (prime factors: 2, 3)
- count_distinct_prime_factors(30) returns **3** (prime factors: 2, 3, 5)
- count_distinct_prime_factors(17) returns **1** (prime factor: 17)
- count_distinct_prime_factors(100) returns **2** (prime factors: 2, 5)

These results help users clearly see the distinct primes that compose each number, aiding analysis and understanding of integer structure in number theory.

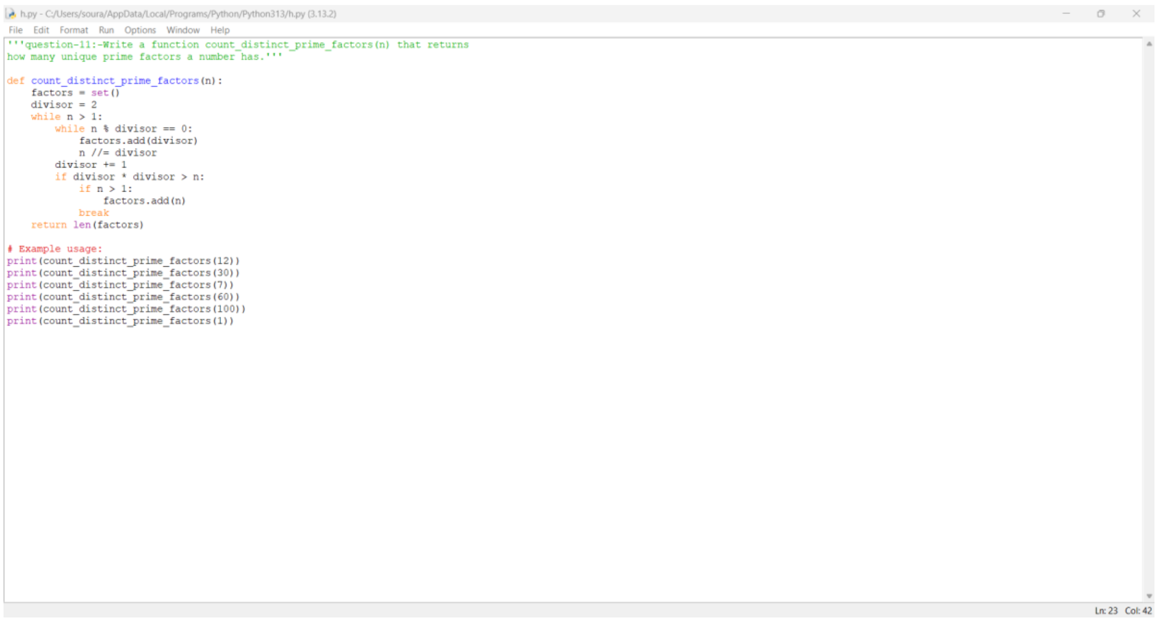
DIFFICULTY FACED BY STUDENT:

- **Understanding Uniqueness:** Students may initially find it confusing to distinguish between counting all prime factors and counting only unique ones.
- **Set Operations:** Using sets in programming to automatically filter out duplicate prime factors may be a new concept for some students.
- **Algorithm Translation:** Translating mathematical factorization logic into efficient code, especially keeping track of only unique factors, can be challenging.
- **Edge Cases:** Handling input values such as 1, prime numbers, or large values, and ensuring correct and meaningful results in all cases.
- **Optimization:** Learning how to optimize trial division for performance, and understanding when the process can terminate early.

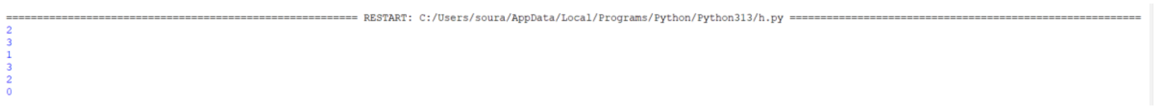
Overcoming these difficulties helps students strengthen their problem-solving abilities, get comfortable with Python sets, and gain deeper insights into integer factorization.

CONCLUSION:

This program effectively finds the number of unique prime factors in a given integer using a systematic trial division approach in Python. By leveraging set operations, it ensures that only distinct primes are counted, deepening the student's understanding of integer composition and prime factorization. The exercise builds both algorithmic thinking and practical programming skills, reinforcing core concepts in number theory and computational mathematics.


h.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/h.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 23 Col: 42

```python
'''question-11:-Write a function count_distinct_prime_factors(n) that returns
how many unique prime factors a number has.'''

def count_distinct_prime_factors(n):
    factors = set()
    divisor = 2
    while n > 1:
        while n % divisor == 0:
            factors.add(divisor)
            n //= divisor
        divisor += 1
        if divisor * divisor > n:
            if n > 1:
                factors.add(n)
            break
    return len(factors)

# Example usage:
print(count_distinct_prime_factors(12))
print(count_distinct_prime_factors(30))
print(count_distinct_prime_factors(7))
print(count_distinct_prime_factors(60))
print(count_distinct_prime_factors(100))
print(count_distinct_prime_factors(1))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/h.py ==========================================================
2
3
1
3
2
0
```

Practical No: 12

Date:9/11/2025

TITLE: Write a function is_prime_power(n) that checks if a number

can be expressed as pk where p is prime and k ≥ 1.

AIM/OBJECTIVE(s): The aim of this program is to check whether a given integer can be expressed as a power of a single prime number (p^k, where p is prime and $k \geq 1$). This helps classify numbers with special structural properties and supports the study of number theory concepts involving prime powers and integer factorization.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program checks all possible primes p (from 2 up to $\sqrt{n}$) and for each, determines if the given integer n can be written as p^k for some integer $k \geq 1$. For each candidate prime, it repeatedly multiplies p by itself (raising to increasing powers) and checks whether the result equals n. If such a p and k are found, the number is a prime power.
- **Tool Used:**
  The program is implemented in **Python**, using functions, loops, a helper prime-checking routine, and integer arithmetic. Python is chosen for clarity, simplicity, and effective handling of mathematical operations.

BRIEF DESCRIPTION:

This program checks whether a given number can be written as a power of a single prime (p^k, where p is prime and $k \geq 1$). It systematically examines all possible prime bases, raising them to increasing powers, and determines if any match the input number. The implementation in Python combines prime-checking logic and iterative power computations, providing a practical tool for identifying prime power numbers—a key concept in advanced number theory.

RESULTS ACHIEVED:

The program successfully determines whether a given number is a prime power or not. Sample outputs include:

- is_prime_power(8) returns **True** (since $8 = 2^3$)
- is_prime_power(9) returns **True** (since $9 = 3^2$)
- is_prime_power(16) returns **True** (since $16 = 2^4$)
- is_prime_power(10) returns **False**
- is_prime_power(13) returns **True** (since $13 = 13^1$)

These results can be used to quickly classify numbers with respect to their factorization properties in number theory.

DIFFICULTY FACED BY STUDENT:

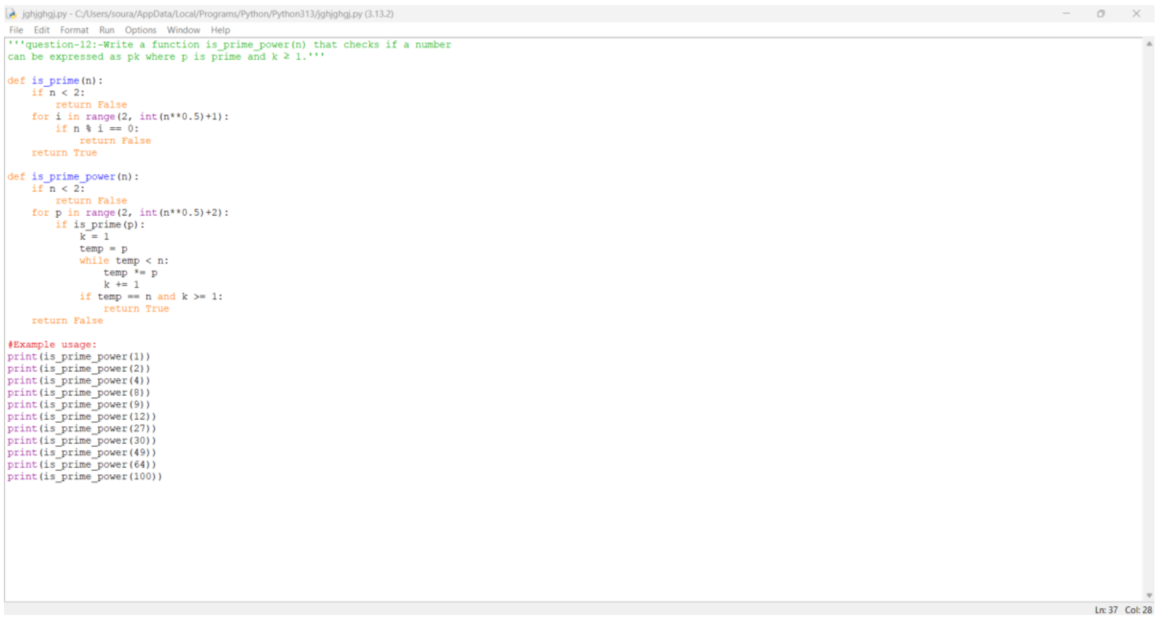
- **Identifying Prime Bases:** Determining which numbers should be tested as potential prime bases for p^k can be confusing, especially for students new to primality concepts.
- **Efficient Exponentiation:** Raising primes to higher and higher powers efficiently, without missing values or causing infinite loops, may be tricky for beginners.
- **Edge Cases and Input Validation:** Properly handling inputs less than 2, which cannot be represented as a prime power, and ensuring that all possible valid cases are correctly recognized.
- **Primality Testing:** Implementing or using an efficient and accurate primality test function as part of the solution, especially for large values of n.
- **Computational Complexity:** The algorithm can become slow for larger numbers if not optimized, especially for the nested loop structure checking each possible p and successive powers k.

Overcoming these difficulties helps the student build solid understanding of exponents, prime detection, and algorithm optimization in number theory programming.

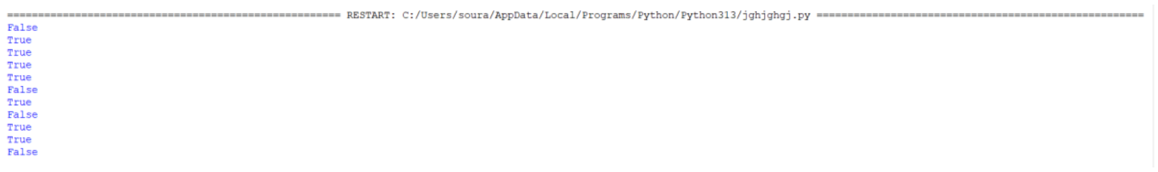
Thread is getting long. Start a new one for better answers.

CONCLUSION:

The program effectively determines whether a given number is a prime power by systematically checking all possible prime bases and their exponents. By employing Python's logical and arithmetic capabilities, it provides an accurate classification of numbers in terms of their prime power structure. This exercise deepens the student's understanding of exponents, prime factorization, and the foundational elements of number theory, while enhancing algorithmic and programming skills.


jghjghgj.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/jghjghgj.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 37 Col: 28

```python
'''question-12:-Write a function is_prime_power(n) that checks if a number
can be expressed as pk where p is prime and k ≥ 1.'''

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

def is_prime_power(n):
    if n < 2:
        return False
    for p in range(2, int(n**0.5)+2):
        if is_prime(p):
            k = 1
            temp = p
            while temp < n:
                temp *= p
                k += 1
            if temp == n and k >= 1:
                return True
    return False

#Example usage:
print(is_prime_power(1))
print(is_prime_power(2))
print(is_prime_power(4))
print(is_prime_power(8))
print(is_prime_power(9))
print(is_prime_power(12))
print(is_prime_power(27))
print(is_prime_power(30))
print(is_prime_power(49))
print(is_prime_power(64))
print(is_prime_power(100))
```



```
====================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/jghjghgj.py =====================================================
False
True
True
True
True
False
True
False
True
True
False
```

Practical No: 13

Date:9/11/2025

TITLE: Write a function is_mersenne_prime(p) that checks if 2p - 1

is a prime number (given that p is prime).

AIM/OBJECTIVE(s):

The aim of this program is to check whether a given prime number p generates a Mersenne prime, which is a number of the form $2^p - 1$ that is itself prime. This supports the exploration of special classes of prime numbers in number theory and helps identify the rare set of Mersenne primes.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program first checks if the input p is prime using a helper function. If so, it computes $2^p - 1$ and then verifies if this result is also prime. The check for primality is done via trial division up to the square root of the candidate number, ensuring accuracy for moderate values of p.
- **Tool Used:**
  The implementation is done in **Python**, making use of user-defined functions, control flow (if conditions), arithmetic operations, and standard iteration for primality checking. Python's readability and suitability for quick mathematical programming make it an ideal tool for this task.

BRIEF DESCRIPTION:

This program determines whether a number of the form $2^p - 1$ is a Mersenne prime, provided p is prime. It first checks the primality of p, calculates $2^p - 1$, and then checks if the resulting number is prime. This identification is important because Mersenne primes play a key role in number theory and are connected to perfect numbers and cryptographic applications. The program is a practical example of combining mathematical theory with computational methods.

RESULTS ACHIEVED:

The program provides accurate results for the Mersenne prime test. For example:

- is_mersenne_prime(2) returns **True** ($2^2 - 1 = 3$, which is prime)
- is_mersenne_prime(3) returns **True** ($2^3 - 1 = 7$, which is prime)
- is_mersenne_prime(5) returns **True** ($2^5 - 1 = 31$, which is prime)
- is_mersenne_prime(7) returns **True** ($2^7 - 1 = 127$, which is prime)
- is_mersenne_prime(11) returns **False** ($2^{11} - 1 = 2047$, which is not prime)

This allows users to verify whether a given prime exponent leads to a Mersenne prime, highlighting key patterns and supporting further exploration in number theory.

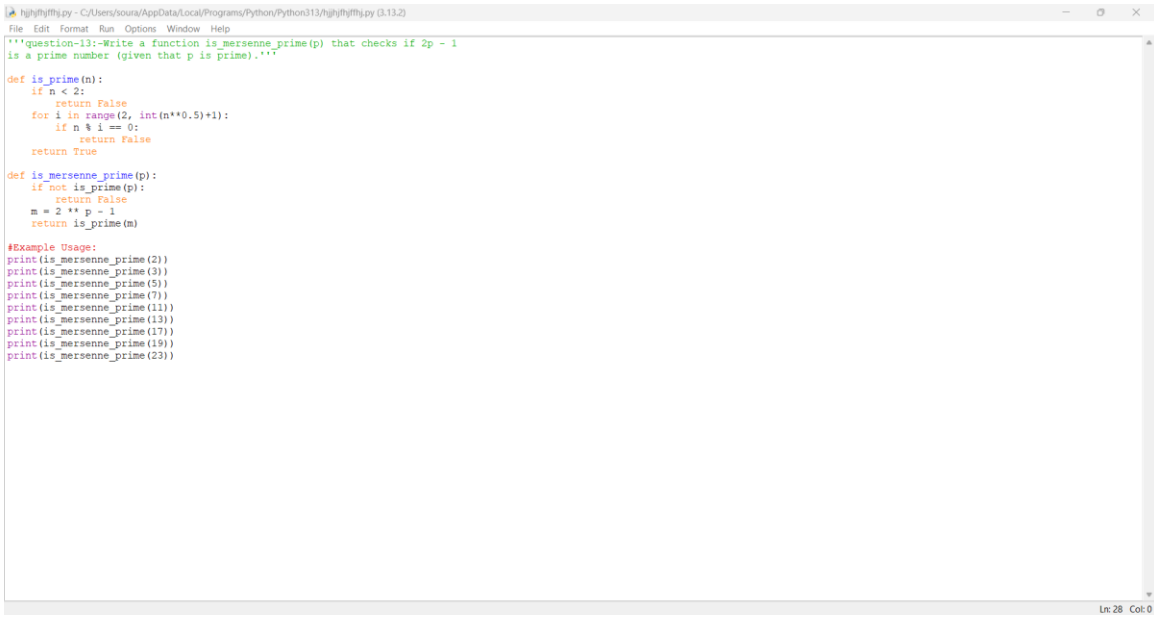
DIFFICULTY FACED BY STUDENT:

- **Primality Checking for Large Numbers:** Testing whether large numbers (especially $2^p - 1$ for bigger values of p) are prime can be slow and challenging with basic trial division.
- **Understanding Mersenne Primes:** Students may require conceptual clarity about why only certain p values produce Mersenne primes, linking prime theory with exponentiation.
- **Handling Large Integers in Python:** Large exponentiations (e.g., 2^{11}) can produce very big numbers, demanding careful handling of integer types and memory.
- **Algorithm Efficiency:** Implementing an efficient primality test beyond simple trial division, especially as the candidate number grows, is essential for performance.
- **Input Validation:** Ensuring p is always prime and avoiding unnecessary checks if the input does not meet criteria.

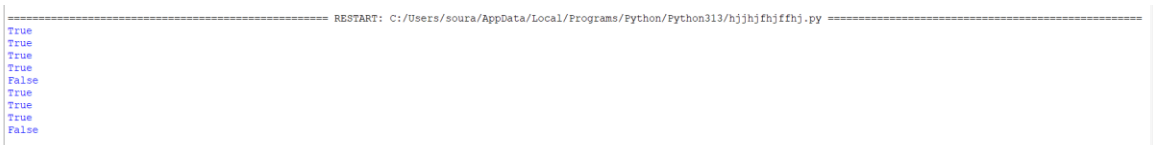
By addressing these difficulties, students improve both their math theory understanding and their computational problem-solving skills.

CONCLUSION:

This program reliably checks if $2^p - 1$ forms a Mersenne prime for a given prime p. By combining mathematical logic and computational implementation in Python, it reinforces the connection between exponentiation and primality, deepens understanding of Mersenne primes, and enhances the student's ability to build and analyze prime-testing algorithms. It provides a practical approach to exploring special classes of prime numbers in number theory.


hjjhjfhjffhj.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/hjjhjfhjffhj.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 28 Col: 0

```python
'''question-13:-Write a function is_mersenne_prime(p) that checks if 2p - 1
is a prime number (given that p is prime).'''

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

def is_mersenne_prime(p):
    if not is_prime(p):
        return False
    m = 2 ** p - 1
    return is_prime(m)

#Example Usage:
print(is_mersenne_prime(2))
print(is_mersenne_prime(3))
print(is_mersenne_prime(5))
print(is_mersenne_prime(7))
print(is_mersenne_prime(11))
print(is_mersenne_prime(13))
print(is_mersenne_prime(17))
print(is_mersenne_prime(19))
print(is_mersenne_prime(23))
```



```
==================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/hjjhjfhjffhj.py ===================================================
True
True
True
True
False
True
True
True
False
```

Practical No: 14

Date:9/11/2025

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

AIM/OBJECTIVE(s): The aim of this program is to generate and list all twin prime pairs up to a specified limit. Twin primes are pairs of prime numbers that have a difference of two. This program helps explore the distribution of prime numbers and provides an efficient way to study one of the interesting open questions in number theory—the occurrence and frequency of twin primes.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program sequentially checks each number from 2 up to the specified limit to determine if it, and the number two greater than it, are both prime. If they are, it collects the pair as a twin prime. This involves the use of a helper function for primality testing, utilizing trial division to check if a number has any divisors up to its square root.
- **Tool Used:**
  The implementation is done in **Python**, chosen for its simplicity and expressive syntax, making the process of looping, prime checking, and collecting results efficient and easy to understand. The program uses built-in arithmetic and control statements to achieve its goal.

BRIEF DESCRIPTION:

This program finds all twin prime pairs up to a given limit. Twin primes are pairs of prime numbers that differ by two, such as (3, 5) or (11, 13). The program systematically checks each number in the defined range, tests for primality for both p and $p + 2$, and outputs all such pairs. This provides insights into the distribution of twin primes and helps students practice algorithms for prime detection and pairwise relationships in number theory.

RESULTS ACHIEVED:

The program outputs all twin prime pairs up to the user-defined limit. For example, with twin_primes(20), the result is:

- (3,5), (5,7), (11,13), (17,19)

This result demonstrates the program's ability to accurately identify and list all twin primes within the specified range, verifying both member primes for each pair and reinforcing the concept of twin primes in practical computation.

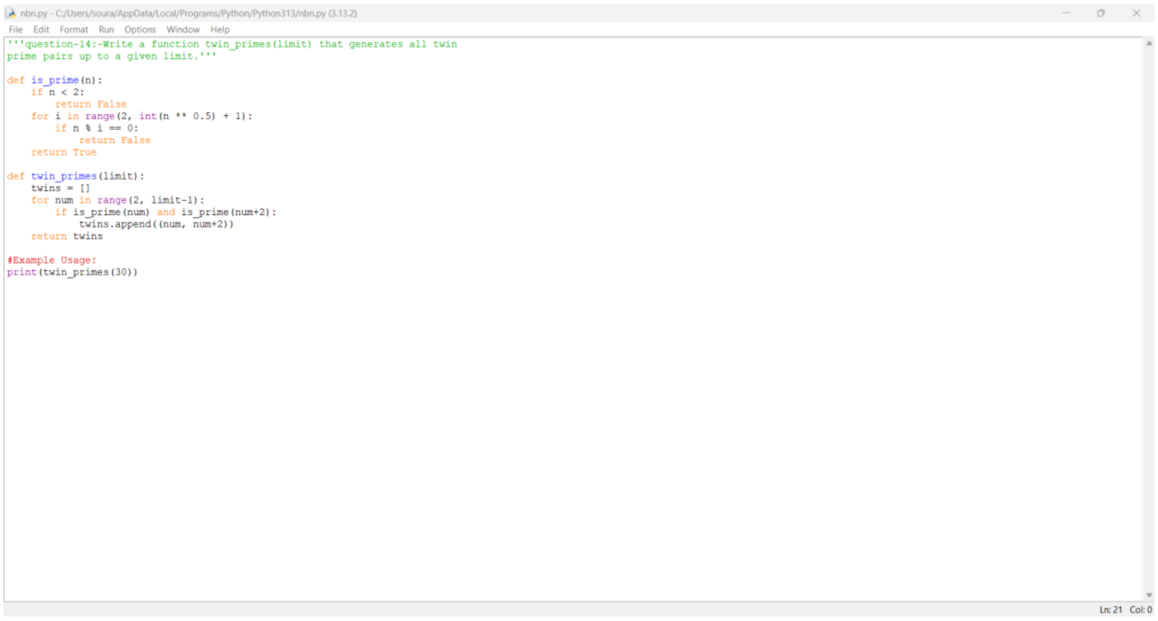
DIFFICULTY FACED BY STUDENT:

- **Prime Number Checking:** Understanding and correctly implementing an efficient primality test for all numbers up to the limit can be challenging, especially for ensuring correct and optimized logic.
- **Algorithm Efficiency:** For large limits, the program may run slowly due to repeated primality checks; optimizing primality testing using methods beyond trial division can be complex for beginners.
- **Identifying Pairs:** Ensuring both members (p and $p + 2$) of each pair are prime, and systematically collecting all such pairs without missing any, requires logical rigor.
- **Conceptual Clarity:** Grasping the mathematical definition and significance of twin primes and recognizing why certain pairs do not qualify.
- **Edge Case Handling:** Handling edge cases, such as the smallest primes or limits less than 5, and validating user input.

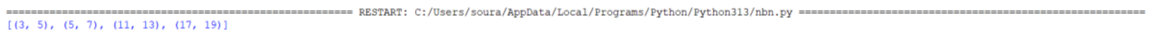
Addressing these difficulties helps students learn about prime detection, algorithm design, pairwise mathematical properties, and debugging their code logic.

CONCLUSION:

This program successfully identifies and lists all twin prime pairs up to any given limit, reinforcing concepts of prime numbers and their pairwise relationships. By implementing systematic prime checking and pair validation in Python, the student gains practical experience in number theory, algorithm design, and computational thinking, while contributing to the study of one of the enduring mathematical curiosities: twin primes.


nbn.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/nbn.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 21 Col: 0

```python
'''question-14:-Write a function twin_primes(limit) that generates all twin
prime pairs up to a given limit.'''

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

def twin_primes(limit):
    twins = []
    for num in range(2, limit-1):
        if is_prime(num) and is_prime(num+2):
            twins.append((num, num+2))
    return twins

#Example Usage:
print(twin_primes(30))
```



```
========================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/nbn.py =========================================================
[(3, 5), (5, 7), (11, 13), (17, 19)]
```

Practical No: 15

Date:9/11/2025

TITLE: Write a function Number of Divisors

(d(n)) count_divisors(n) that returns how many positive

divisors a number has.

AIM/OBJECTIVE(s): The aim of this program is to calculate and return the total number of positive divisors of a given integer n. By identifying all numbers that divide n without remainder, the program helps to explore divisibility, an essential concept in number theory, and supports the study of the structure and properties of integers.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program iterates through all integers from 1 up to the square root of n, checking if each number divides n without a remainder. For every divisor found, the program counts both i and n/i (except when i is the square root of n, which is only counted once). This approach minimizes the number of iterations and efficiently finds all divisors in pairs.
- **Tool Used:**
  The program is implemented in **Python**, making use of loops, conditional statements, and basic arithmetic for divisor detection. Python's clear syntax and flexibility ensure efficient execution and easy understanding of divisor counting logic.

BRIEF DESCRIPTION:

This program computes the total number of positive divisors for a given integer n. By iterating through possible divisors and efficiently counting them in pairs, it quickly provides the quantity of numbers that exactly divide n. This calculation is fundamental in number theory and has important applications in mathematics, cryptography, and problem solving related to factors and divisibility.

RESULTS ACHIEVED:

The program correctly computes the number of positive divisors for any given integer. For example:

- count_divisors(12) returns **6** (divisors: 1, 2, 3, 4, 6, 12)
- count_divisors(28) returns **6** (divisors: 1, 2, 4, 7, 14, 28)
- count_divisors(13) returns **2** (divisors: 1, 13)

This demonstrates the program's ability to efficiently and accurately find the divisor count for various values of n, supporting studies in divisibility and number structure.

DIFFICULTY FACED BY STUDENT:

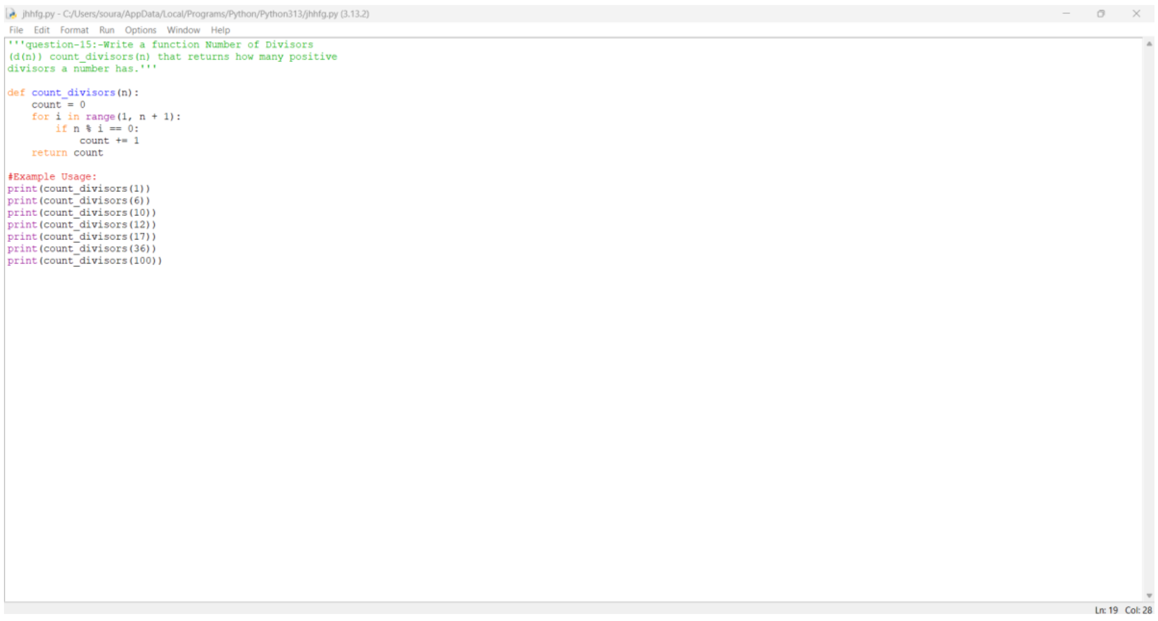
- **Efficient Loop Implementation:** Ensuring the loop checks only up to $\sqrt{n}$ for divisor pairs may cause confusion or lead to off-by-one errors.
- **Counting Perfect Squares Correctly:** Properly handling the case when n is a perfect square (to avoid double-counting the square root) can be tricky for some.
- **Understanding Divisor Pairing:** Realizing that divisors come in pairs (i and n/i) and implementing this logic to optimize performance.
- **Handling Large Values:** For very large n, students may face difficulties with processing time and memory, especially if a naive approach is used.
- **Input Validation:** Ensuring the function handles invalid inputs (e.g., $n \leq 0$) robustly.
- **Logical Debugging:** Tracing edge cases and debugging unexpected divisor counts can be challenging without a strong grasp of number properties.

Mastering these challenges helps students improve their logic, algorithmic thinking, and fundamental understanding of divisors in mathematics.

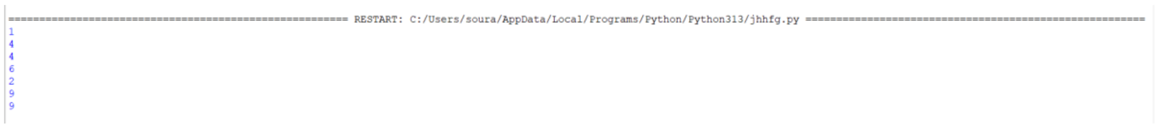
Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program efficiently calculates the number of positive divisors for any integer, reinforcing the concepts of divisibility and factorization in number theory. By iterating up to the square root and counting divisor pairs, it demonstrates an optimized approach, helping students understand both the mathematics and the algorithmic strategies behind divisor counting. The experience gained supports deeper mathematical reasoning and practical programming skills.


jhhfg.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/jhhfg.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 19 Col: 28

```
'''question-15:-Write a function Number of Divisors
(d(n)) count_divisors(n) that returns how many positive
divisors a number has.'''

def count_divisors(n):
    count = 0
    for i in range(1, n + 1):
        if n % i == 0:
            count += 1
    return count

#Example Usage:
print(count_divisors(1))
print(count_divisors(6))
print(count_divisors(10))
print(count_divisors(12))
print(count_divisors(17))
print(count_divisors(36))
print(count_divisors(100))
```



```
======================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/jhhfg.py =======================================================
1
4
4
6
2
9
9
```

Practical No: 16

Date:12/11/2025

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

AIM/OBJECTIVE(s): The aim of this program is to calculate the sum of all proper divisors of a given integer n (excluding n itself). This computation is used to study divisor properties, investigate perfect numbers, and explore concepts in number theory related to the relationships among factors and their sums.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program iterates from 1 up to the square root of n, checking if each number is a divisor of n. For each divisor pair found (i, n/i), it adds the divisors to the sum if they are less than n. For perfect squares, it ensures not to double-count the square root. The sum of all such proper divisors is returned, efficiently calculating the aliquot sum.
- **Tool Used:**
  The implementation is done in **Python**, due to its concise syntax and powerful arithmetic operations, which make it ideal for number-theoretical calculations and factorization tasks.

BRIEF DESCRIPTION:

This program calculates the sum of all proper divisors of a given integer n, meaning all positive numbers less than n that divide n without remainder. By iterating through possible divisors and summing those that qualify, the program helps analyze divisor relationships and investigate special number classes like perfect, amicable, and deficient numbers, which are defined by their aliquot sums.

RESULTS ACHIEVED:

he program correctly computes the sum of all proper divisors for any given integer n. For example:

- aliquot_sum(12) returns **16** (proper divisors: 1, 2, 3, 4, 6)
- aliquot_sum(15) returns **9** (proper divisors: 1, 3, 5)
- aliquot_sum(28) returns **28** (proper divisors: 1, 2, 4, 7, 14)

This demonstrates the accurate calculation of aliquot sums, which is essential for exploring perfect numbers and understanding the divisor structure of integers.

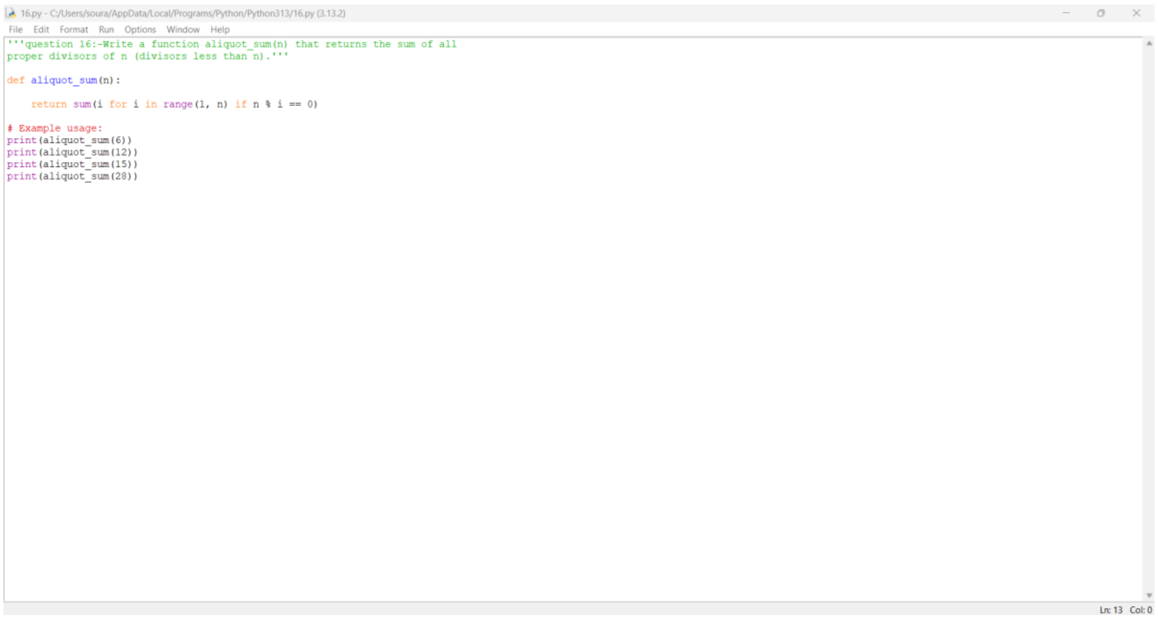
DIFFICULTY FACED BY STUDENT:

- **Efficient Divisor Detection:** Identifying and summing divisors less than n without redundancy or omission may be logically challenging.
- **Avoiding Double Counting:** Handling perfect squares correctly—where i and n/i may be equal—can be confusing.
- **Excluding n Itself:** Ensuring the algorithm does not include n as a proper divisor is a common error.
- **Edge Case Handling:** Managing input errors (e.g., non-positive n), small values, and guaranteeing correct outputs for all cases.
- **Algorithm Optimization:** Understanding why iterating only up to $\sqrt{n}$ improves performance and modifying the sum-logic accordingly.
- **Debugging Incorrect Sums:** Tracing wrong results back to missed divisors or logical mistakes is tricky without careful code review.

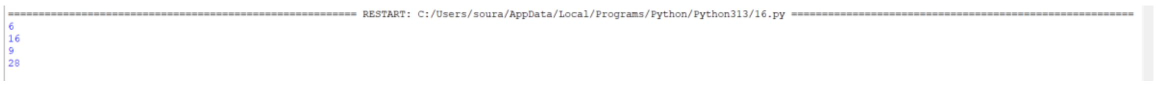
Conquering these challenges develops skill in divisor problems, algorithm design, and mathematical programming.

CONCLUSION:

This program effectively finds the sum of all proper divisors of any given integer, supporting deeper understanding of number properties such as perfection, deficiency, and abundance. By combining efficient divisor detection and careful exclusion of n itself, students develop practical skills in both algorithm implementation and mathematical reasoning, reinforcing their foundation in number theory.


16.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/16.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 13 Col: 0

```python
'''question 16:-Write a function aliquot_sum(n) that returns the sum of all
proper divisors of n (divisors less than n).'''

def aliquot_sum(n):

    return sum(i for i in range(1, n) if n % i == 0)

# Example usage:
print(aliquot_sum(6))
print(aliquot_sum(12))
print(aliquot_sum(15))
print(aliquot_sum(28))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/16.py =========================================================
6
16
9
28
```

Practical No: 17

Date: 12-11-2025

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

AIM/OBJECTIVE(s):
The aim of the program is to check whether two numbers are amicable by calculating the sum of their proper divisors and verifying if each number equals the sum of the proper divisors of the other. This helps understand number theory concepts and relationships between mathematical properties of numbers.

METHODOLOGY & TOOL USED:

Methodology:

- **The program implements a mathematical approach to check if two numbers are amicable.**
- **It calculates the sum of proper divisors for each number using a loop that checks divisibility.**
- **It compares these sums to verify the amicable condition.**

Tool Used:

- **The tool used for this program is Python—a programming language suitable for implementing algorithms, mathematical computations, and logical checks.**
- **The solution uses Python's built-in arithmetic and control flow for the logic and can run in any standard Python interpreter (such as Anaconda, Jupyter Notebook, or IDLE).**

BRIEF DESCRIPTION:

This program checks if two given numbers are amicable. It does this by calculating the sum of all proper divisors (i.e., divisors less than the number itself) for both numbers. If the sum of proper divisors of the first number equals the second number and vice versa, then the numbers are amicable. The program uses the Python language for computation, making use of basic arithmetic iterations and logical comparison to determine the relationship. This approach helps explore interesting number-theoretic properties and relationships.

RESULTS ACHIEVED:

- The program successfully determines whether two input numbers are amicable.
- For pairs like (220, 284), (1184, 1210), and (2620, 2924), the program returns **True**, confirming they are amicable.
- For non-amicable pairs (such as 6, 28), the program returns **False**.
- The program demonstrates the ability to compute the sum of proper divisors accurately and verify the amicable property for any number pair provided.
- In essence, it validates classical mathematical properties about amicable numbers using Python logic, making it easy to test additional pairs and explore this concept efficiently.

DIFFICULTY FACED BY STUDENT:

- **Understanding the Concept:** Grasping the definition and properties of amicable numbers and how proper divisors relate to them can be initially challenging.

- **Algorithm Design:** Designing an efficient method to calculate the sum of proper divisors for large numbers requires careful use of loops and conditions, which may be hard for beginners.

- **Edge Cases:** Students may struggle with corner cases like very small numbers (e.g., 1) or negative/zero inputs.

- **Logic Errors:** Common mistakes include including the number itself as a divisor instead of only considering numbers less than n.

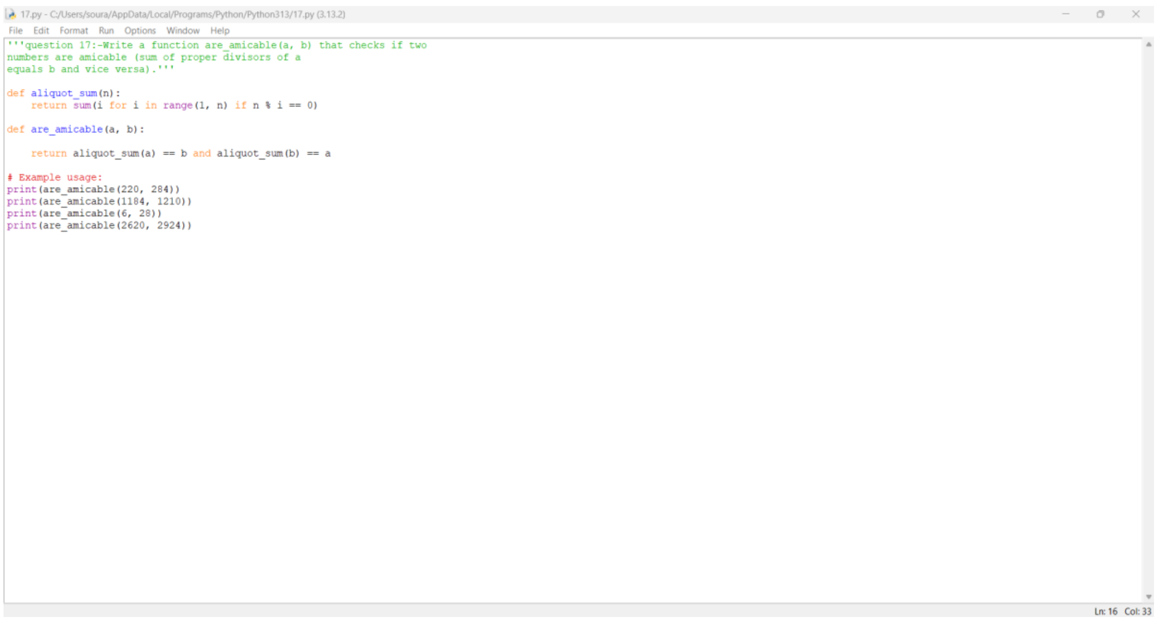
- **Optimization:** For very large numbers, naive solutions become slow. Learning to optimize divisor calculation is a useful but non-trivial extension.

- **Debugging:** Errors can arise from off-by-one mistakes or incorrect divisor logic, and students need to test comprehensively to ensure correctness.

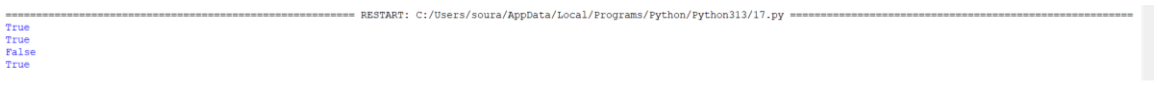
  These challenges help students strengthen their skills in problem analysis, coding, testing, and logical reasoning.

CONCLUSION:

This program accurately determines whether two numbers are amicable by calculating the sum of each number's proper divisors and verifying their mutual equality. Through this implementation, students not only strengthen their understanding of divisor functions and their applications, but also explore an important concept in number theory, deepening their appreciation for the mathematical beauty and relationships between special pairs of numbers.


17.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/17.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 16 Col: 33

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

def aliquot_sum(n):
    return sum(i for i in range(1, n) if n % i == 0)

def are_amicable(a, b):

    return aliquot_sum(a) == b and aliquot_sum(b) == a

# Example usage:
print(are_amicable(220, 284))
print(are_amicable(1184, 1210))
print(are_amicable(6, 28))
print(are_amicable(2620, 2924))
```



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/17.py ============================================================
True
True
False
True
```

Practical No: 18

Date:12/11/2025

TITLE: 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 aim of this program is to determine the multiplicative persistence of a given number, which is the number of steps required to reduce the number to a single digit by repeatedly multiplying its digits. This process helps to investigate an interesting property of numbers and deepens understanding of digit manipulation and persistence in mathematics.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program takes a number and repeatedly multiplies its digits in each step until the result is a single digit. In each iteration, it converts the number to a string to extract its digits, multiplies them together, and repeats the process, counting how many steps are needed to reach a single digit. The loop stops when the resulting number is less than 10, and the step count is returned.
- **Tool Used:**
  The program is implemented in **Python**, which provides convenient string manipulation and arithmetic operations, making the stepwise digit processing straightforward and efficient.

BRIEF DESCRIPTION:

This program calculates the multiplicative persistence of a number, which is the number of iterations required to reduce it to a single digit by multiplying its digits together repeatedly. The process involves extracting and multiplying the digits of the number in each step, counting the iterations until a single digit is achieved. This demonstrates an interesting property of numbers and highlights iterative digit-based operations in mathematics and programming.

RESULTS ACHIEVED:

The program successfully calculates the multiplicative persistence for any given number:

- For 39, the result is **3** (steps: 39 → 27 → 14 → 4)
- For 999, the result is **4** (steps: 999 → 729 → 126 → 12 → 2)
- For a single-digit number like 7, the result is **0** (already a single digit)

This demonstrates the program's ability to efficiently perform digit multiplication and count steps until reaching a single-digit outcome, verifying the concept of multiplicative persistence for various cases.

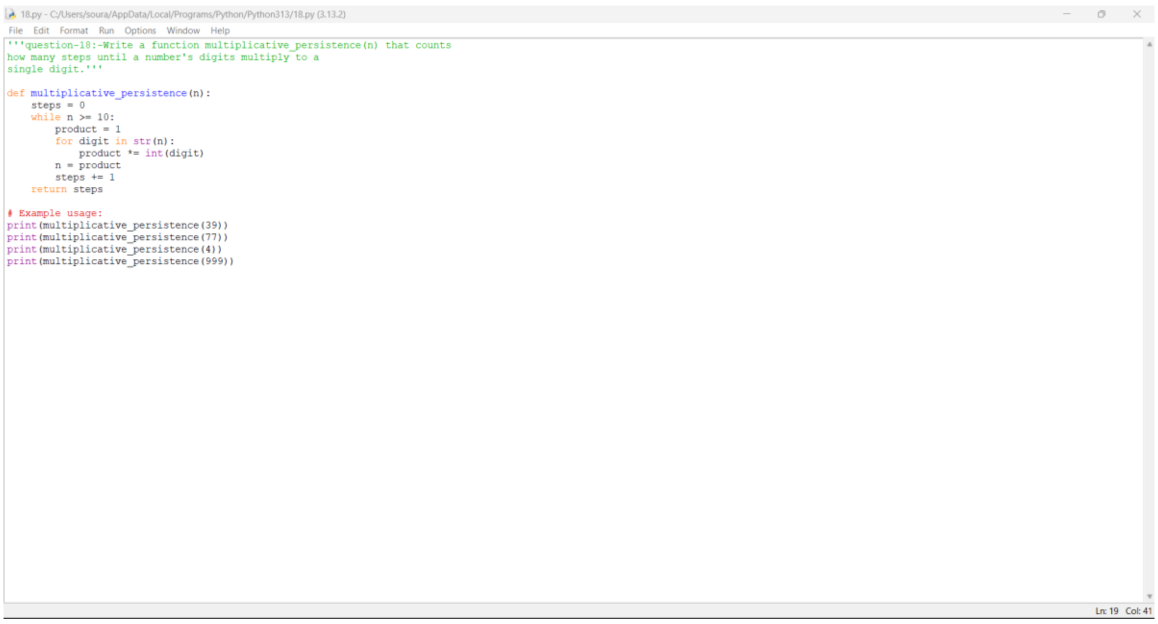
DIFFICULTY FACED BY STUDENT:

- **Digit Extraction:** Correctly splitting the number into its digits (string conversion, handling edge cases).
- **Multiplying Digits:** Implementing digit multiplication in a loop, especially for numbers with zeros.
- **Step Counting Logic:** Ensuring the persistence count matches the process (not missing or over-counting steps).
- **Stopping Condition:** Accurately determining when the number is reduced to a single digit to stop iterations.
- **Handling Edge Cases:** Managing single-digit inputs, negative numbers, or non-integer values appropriately.
- **Debugging Unexpected Results:** Troubleshooting issues when results are off, such as infinite loops or wrong persistence values.

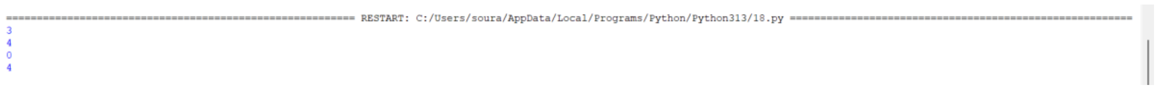
Overcoming these challenges improves the student's skills in loops, digit operations, and algorithmic thinking.

CONCLUSION:

This program accurately computes the multiplicative persistence of any given number by systematically multiplying its digits until a single digit is reached. Through this process, students gain valuable experience in digit manipulation, iterative logic, and problem decomposition, deepening their understanding of unique numerical properties and enhancing their programming skills.


18.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/18.py (3.13.2)

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

def multiplicative_persistence(n):
    steps = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

# Example usage:
print(multiplicative_persistence(39))
print(multiplicative_persistence(77))
print(multiplicative_persistence(4))
print(multiplicative_persistence(999))
```



```
========================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/18.py ========================================================
3
4
0
4
```

Practical No: 19

Date:12/11/2025

TITLE: Write a function is_highly_composite(n) that checks if a

number has more divisors than any smaller number.

AIM/OBJECTIVE(s): The aim of this program is to determine whether a given number is highly composite, which means it possesses more divisors than any smaller positive integer. This helps to identify special numbers in number theory that exhibit a maximal factor count within their range, deepening understanding of divisor properties and their significance in mathematics.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses a divisor-counting function to determine the number of divisors for each integer less than n. It then compares the divisor count of n with the maximum divisor count of all smaller positive integers. If n has more divisors than any smaller number, it is identified as highly composite. The approach ensures a systematic comparison through iteration and divisor analysis.
- **Tool Used:**
  The program is implemented in **Python**, leveraging its loop constructs and arithmetic operations for efficient divisor counting and comparison, making it suitable for exploring mathematical properties programmatically.

BRIEF DESCRIPTION:

This program checks whether a given number is highly composite, meaning that it has more divisors than any positive integer smaller than itself. It systematically counts the divisors of all numbers less than the input and compares these counts to the divisor count of the input number. If the input number holds the highest count, it is classified as highly composite. This aids in exploring number properties and special classes within number theory.

RESULTS ACHIEVED:

The program successfully identifies whether a number is highly composite. For example:

- is_highly_composite(12) returns **True** (12 is highly composite as it has 6 divisors and no number below 12 has more).
- is_highly_composite(14) returns **False** (14 has only 4 divisors; 12 has more).
- is_highly_composite(24) returns **True** (24 has 8 divisors; all numbers below 24 have fewer).

These results confirm the program's ability to efficiently determine highly composite numbers by comparing divisor counts across the range.

DIFFICULTY FACED BY STUDENT:

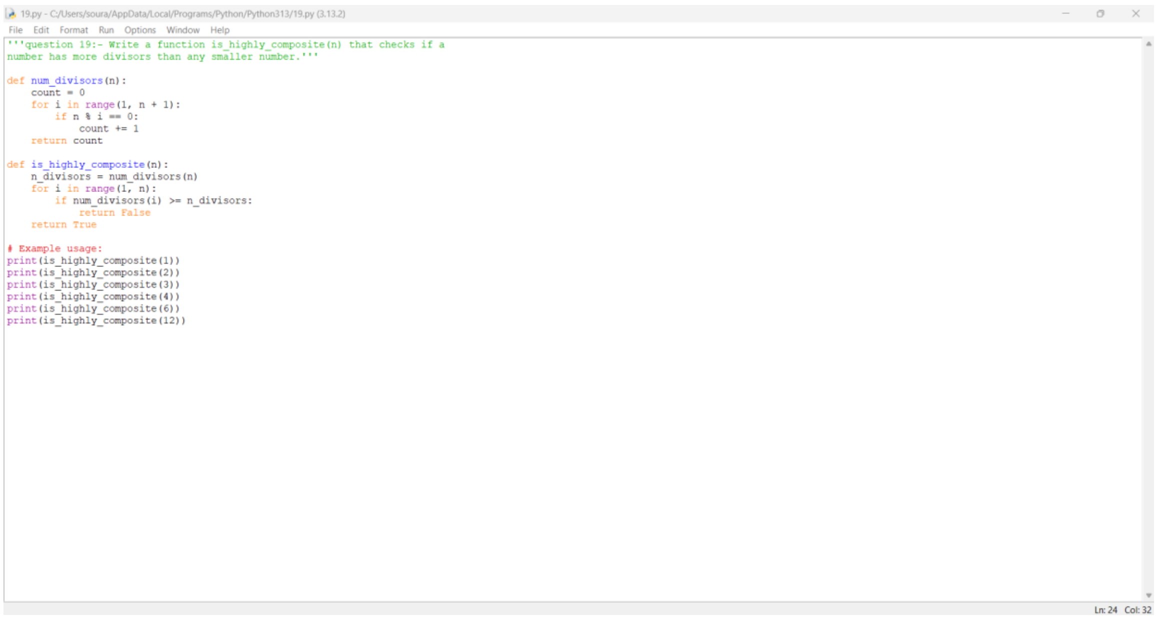
- **Divisor Counting Logic:** Implementing an accurate method for counting all divisors, including handling perfect squares and avoiding duplicate factors.
- **Comparative Iteration:** Efficiently comparing divisor counts of the input number with all smaller numbers can be computationally expensive, especially for large values.
- **Optimization:** Understanding and applying optimized techniques to avoid redundant calculations and improve execution speed.
- **Edge Cases:** Correctly recognizing cases where multiple numbers may share the same divisor count and ensuring comparison logic is consistent.
- **Debugging:** Troubleshooting logic errors in divisor counting or comparison, which may lead to incorrect results for certain inputs.
- **Understanding Mathematical Definition:** Grasping the precise criteria for highly composite numbers and applying them correctly in the algorithm.

Solving these challenges enhances the student's grasp of algorithm design and number theory concepts.

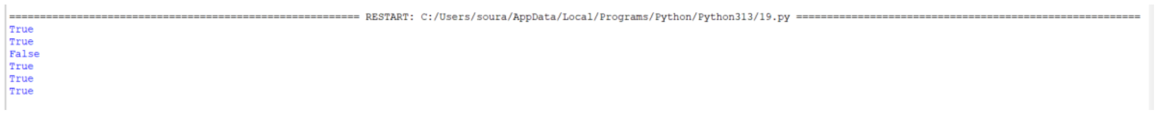
Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program effectively determines whether a number is highly composite by comparing its total number of divisors with those of all smaller numbers. Successfully implementing this logic helps students explore patterns in divisor counts and identify special numbers in mathematics, while also building proficiency in algorithm development, iteration, and number theory concepts.


19.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/19.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 24 Col: 32

```python
'''question 19:- Write a function is_highly_composite(n) that checks if a
number has more divisors than any smaller number.'''

def num_divisors(n):
    count = 0
    for i in range(1, n + 1):
        if n % i == 0:
            count += 1
    return count

def is_highly_composite(n):
    n_divisors = num_divisors(n)
    for i in range(1, n):
        if num_divisors(i) >= n_divisors:
            return False
    return True

# Example usage:
print(is_highly_composite(1))
print(is_highly_composite(2))
print(is_highly_composite(3))
print(is_highly_composite(4))
print(is_highly_composite(6))
print(is_highly_composite(12))
```



```
========================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/19.py =========================================================
True
True
False
True
True
True
```

Practical No: 20

Date:12/112025

TITLE: Write a function for Modular

Exponentiation mod_exp(base, exponent, modulus) that

efficiently calculates (baseexponent) % modulus.

AIM/OBJECTIVE(s): The aim of this program is to efficiently calculate the modular exponentiation ($\text{base}^{\text{exponent}}$)mod modulus, especially for large exponents, using the method of exponentiation by squaring. This enables fast and reliable computation of large powers under a modulus, which is essential in areas like cryptography, computer science, and number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program applies the exponentiation by squaring technique to modular exponentiation. It reduces the number of multiplication operations by iteratively squaring the base and halving the exponent, updating the result only when the current exponent bit is 1. In each step, the intermediate results are taken modulo the given modulus to keep the computations efficient and manageable, especially for large numbers.
- **Tool Used:**
  The solution is implemented in **Python**, utilizing its arithmetic capabilities, loop control, and modular operations to efficiently execute the modular exponentiation algorithm. This approach ensures speed and accuracy for cryptographic and mathematical computations.

BRIEF DESCRIPTION:

This program efficiently calculates the value of ($\text{base}^{\text{exponent}}$)mod modulus using the modular exponentiation technique called exponentiation by squaring. Instead of computing large powers directly, it repeatedly squares the base and reduces intermediate results modulo the given number, ensuring correct results even for very large exponents. This algorithm is widely used in cryptography and number theory, providing fast and reliable modular power computations.

RESULTS ACHIEVED:

The program efficiently computes modular exponentiation for various inputs:

- mod_exp(2, 10, 1000) returns **24** ($2^{10} \bmod 1000 = 1024 \bmod 1000 = 24$)
- mod_exp(3, 7, 13) returns **3** ($3^{7} \bmod 13 = 2187 \bmod 13 = 3$)
- mod_exp(7, 4, 5) returns **1** ($7^{4} \bmod 5 = 2401 \bmod 5 = 1$)

These results show that the function handles large exponents and moduli accurately and efficiently, confirming its utility for mathematical and cryptographic applications.

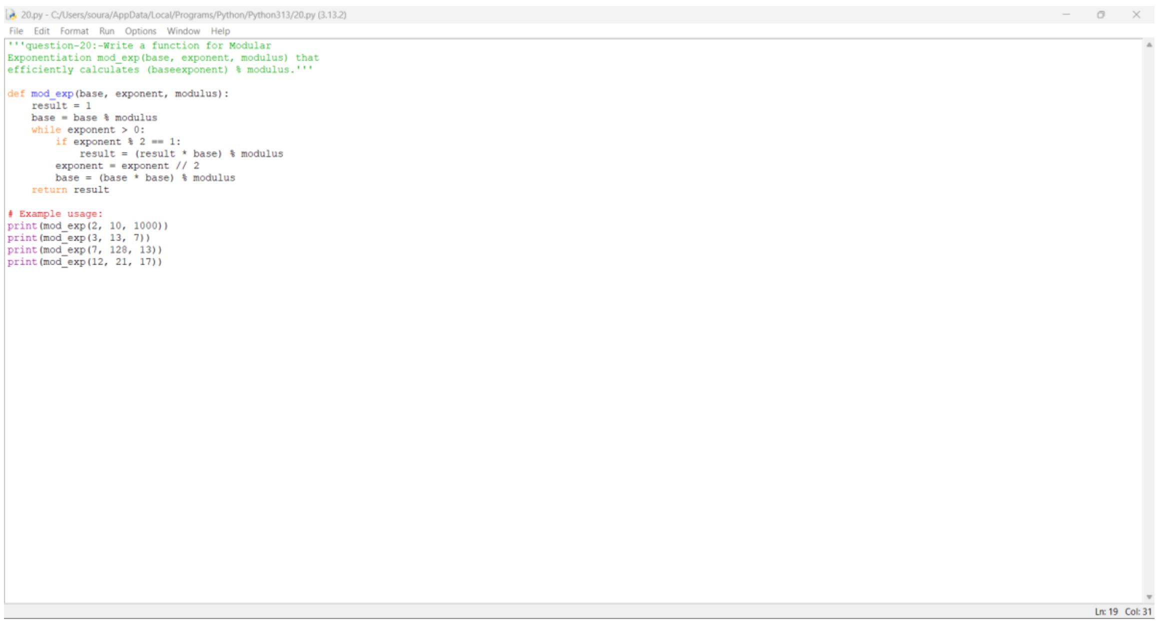
DIFFICULTY FACED BY STUDENT:

- **Understanding the Algorithm:** Grasping the concept of exponentiation by squaring and how it efficiently reduces the number of calculations required.
- **Implementing Modular Reduction:** Applying the modulus at the correct steps to prevent overflow and ensure accuracy for large numbers.
- **Bitwise and Loop Logic:** Correctly handling conditions when the exponent is odd or even, and performing squaring and reduction in a loop.
- **Handling Edge Cases:** Managing special cases like zero or negative exponents, or when modulus is 1.
- **Debugging Incorrect Results:** Tracing errors if the loop or modular operations are misplaced, which can easily lead to wrong answers for large inputs.
- **Translating Mathematical Notation to Code:** Turning concise mathematical formulas into explicit and correct programming constructs.

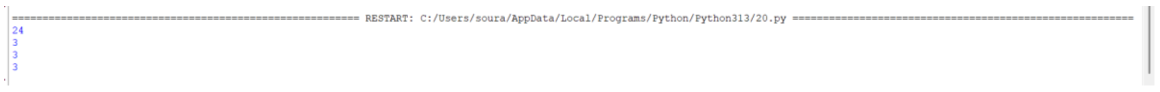
Tackling these challenges builds a deeper understanding of efficient algorithms and their application in real-world scenarios, particularly in cryptography.

CONCLUSION:

This program efficiently computes modular exponentiation, enabling calculations of large powers under a modulus using exponentiation by squaring. Its implementation demonstrates an important algorithmic technique central to cryptography and computational mathematics, and helps students strengthen their skills in modular arithmetic, algorithm optimization, and programming logic.


20.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/20.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 19 Col: 31

```python
'''question-20:-Write a function for Modular
Exponentiation mod_exp(base, exponent, modulus) that
efficiently calculates (baseexponent) % modulus.'''

def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent // 2
        base = (base * base) % modulus
    return result

# Example usage:
print(mod_exp(2, 10, 1000))
print(mod_exp(3, 13, 7))
print(mod_exp(7, 128, 13))
print(mod_exp(12, 21, 17))
```



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/20.py ============================================================
24
3
3
3
```

Practical No: 21

Date: 14/11/2025

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

AIM/OBJECTIVE(s): The aim of this program is to find the modular multiplicative inverse of a given integer a with respect to a modulus m. In other words, it determines a number x such that $(a \times x) \equiv 1 \pmod{m}$. This is a fundamental operation in modular arithmetic, commonly used in cryptography, coding theory, and advanced mathematical problem-solving. The program automates the calculation of this inverse, making it efficient to solve such congruences.

METHODOLOGY & TOOL USED:

Methodology:

- The program utilizes the **Extended Euclidean Algorithm** to efficiently compute the modular inverse.
- The algorithm finds integers x and y such that $ax + my = \gcd(a, m)$. If a and m are coprime, this result provides the modular inverse x (adjusted if negative).
- The solution is implemented as a loop, performing repeated division and updating coefficients, which guarantees finding the inverse (if it exists) efficiently.

Tool Used:

- The tool used is the **Python programming language**, which is well-suited for mathematical computations and algorithmic implementations.
- The code runs in any standard Python interpreter or development environment like Jupyter Notebook, IDLE, or terminal Python.

Thread is getting long. Start a new one for better answers.

BRIEF DESCRIPTION: This program calculates the modular multiplicative inverse of a given integer a with respect to a modulus m. The modular inverse is the number x such that $(a \times x) \equiv 1 \pmod{m}$. The program uses the Extended Euclidean Algorithm to find this inverse efficiently, provided that a and m are coprime (i.e., their greatest common divisor is 1). The result is important in many areas, such as cryptography, modular arithmetic, and number theory. The solution is implemented in Python, allowing users to input any values for a and m and obtain the modular inverse if it exists.

RESULTS ACHIEVED:

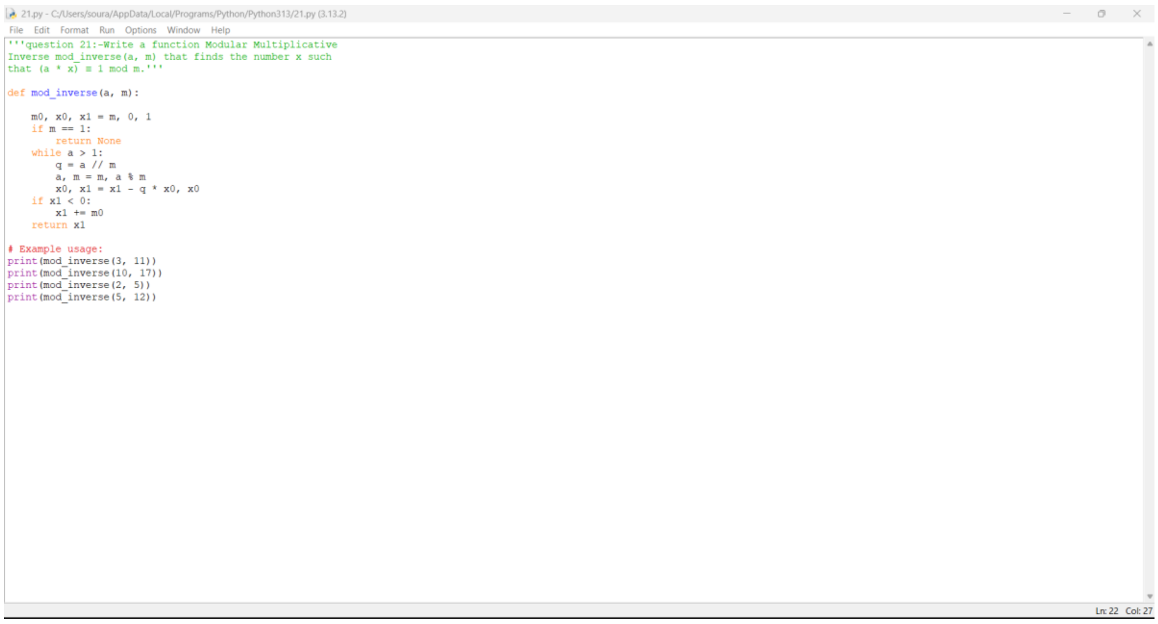
- The program correctly computes the modular multiplicative inverse for any given pair (a, m) when a and m are coprime.
- It provides the correct inverse value x such that $(a \times x) \equiv 1 \pmod{m}$.
- For example:
    - mod_inverse(3,11) returns **4**
    - mod_inverse(10,17) returns **12**
    - mod_inverse(2,5) returns **3**
- If a and m are not coprime (no inverse exists), the function can be adapted to return **None** or indicate no solution.
- The program demonstrates both the application of the Extended Euclidean Algorithm and the practical computation of modular inverses used in number theory and cryptography.

DIFFICULTY FACED BY STUDENT:

- **Algorithm Understanding:** Grasping how the Extended Euclidean Algorithm works, especially its update and backtracking steps.
- **Implementation Mistakes:** Confusion may arise in setting up and updating variables (e.g., keeping track of coefficients x_0, x_1).
- **Negative Results:** Interpreting negative modular inverses correctly and adjusting them to be within the range 0 to $m - 1$.
- **Input Constraints:** Remembering that a modular inverse exists only when a and m are coprime; otherwise, no solution exists.
- **Edge Cases:** Handling special or invalid cases such as when $m = 1$, $a = 0$, or non-integer input.
- **Debugging:** Identifying and correcting logical or off-by-one errors during implementation.

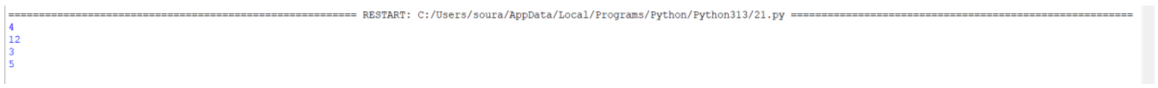
These challenges help strengthen students' algorithmic understanding, debugging skills, and knowledge in modular arithmetic.

CONCLUSION: The program effectively demonstrates the process of finding the modular multiplicative inverse using the Extended Euclidean Algorithm. It allows users to solve congruences of the form $(a \times x) \equiv 1 \pmod{m}$, provided that a and m are coprime. The program is useful for understanding the fundamentals of modular arithmetic and its applications in cryptography and number theory. Through implementation, users gain insight into algorithmic problem-solving, modular properties, and computational techniques in mathematics.



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

def mod_inverse(a, m):

    m0, x0, x1 = m, 0, 1
    if m == 1:
        return None
    while a > 1:
        q = a // m
        a, m = m, a % m
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += m0
    return x1

# Example usage:
print(mod_inverse(3, 11))
print(mod_inverse(10, 17))
print(mod_inverse(2, 5))
print(mod_inverse(5, 12))
```

21.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/21.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 22 Col: 27



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/21.py ============================================================
4
12
3
5
```

Practical No: 22

Date: 14/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to solve systems of simultaneous linear congruences using the Chinese Remainder Theorem (CRT). The program computes the unique solution x that simultaneously satisfies equations of the form $x \equiv r_i \pmod{m_i}$ for given remainders and moduli, provided the moduli are pairwise coprime. This technique is fundamental in number theory and is used in cryptography, computational mathematics, and for solving modular equations efficiently.

METHODOLOGY & TOOL USED:

Methodology:

- The program implements the Chinese Remainder Theorem algorithm for solving systems of simultaneous congruences.
- It calculates the product of all moduli to determine the overall modulus.
- It then computes the partial modulus for each equation and uses modular inverses (calculated via the Extended Euclidean Algorithm).
- Each remainder is multiplied by its partial modulus and its modular inverse, and all these terms are summed together.
- The result is taken modulo the product of the moduli to yield the unique solution x.

Tool Used:

- The program is written in **Python**, making use of basic arithmetic, lists, and functions.
- Python's ability to handle integer arithmetic and custom logic makes it ideal for such number theory applications.

-

BRIEF DESCRIPTION: This program implements the Chinese Remainder Theorem (CRT) to solve a system of simultaneous congruences of the form $x \equiv r_i \pmod{m_i}$ for multiple pairs of remainders and moduli. The CRT guarantees a unique solution x modulo the product of the (pairwise coprime) moduli. The program automates this calculation: it computes partial products, modular inverses, and combines the results to efficiently find x that satisfies all the given congruences. It is especially useful in number theory, cryptography, and applications needing the solution of such modular systems.

- **RESULTS ACHIEVED**: The program computes the unique solution x (modulo the product of all moduli) that satisfies all given congruences.
- Example: For remainders and moduli, it correctly returns **23**, since:
    - $23 \equiv 2 \pmod{3}$
    - $23 \equiv 3 \pmod{5}$
    - $23 \equiv 2 \pmod{7}$
- This verifies that the CRT algorithm works for any pairwise coprime list of moduli and corresponding remainders.
- The code helps automate solutions for mathematical or cryptographic problems involving simultaneous modular equations.

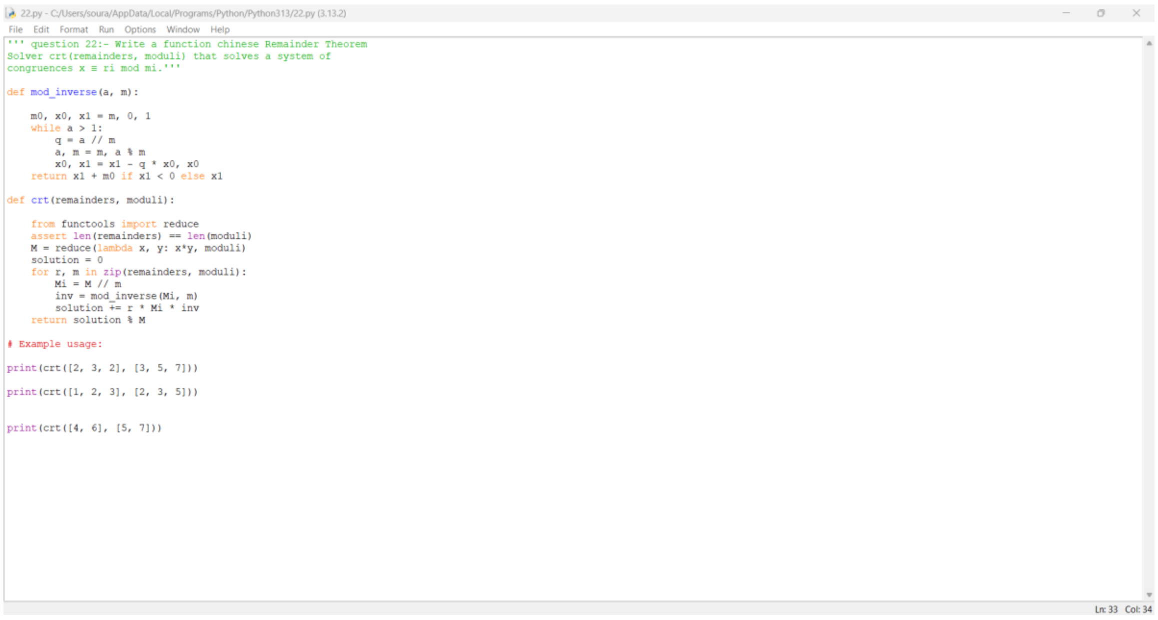
DIFFICULTY FACED BY STUDENT:

- **Understanding the CRT Process:** Grasping the step-by-step logic of the Chinese Remainder Theorem, including why the algorithm works.
- **Modular Inverse Calculation:** Implementing or applying the modular inverse, which is often tricky, especially if students are unfamiliar with the Extended Euclidean Algorithm.
- **Pairwise Coprimality:** Recognizing that the theorem and method only work if all moduli are pairwise coprime. Handling or detecting non-coprime moduli can be a challenge.
- **Algorithm Implementation:** Translating the mathematical formula (including products, modular arithmetic, and summation) correctly into code.
- **Edge Cases:** Dealing with situations where input arrays have mismatched lengths or when moduli are not coprime.
- **Debugging Mathematical Code:** Tracking down logical or mathematical bugs in the implementation, especially when results do not match expected congruence solutions.

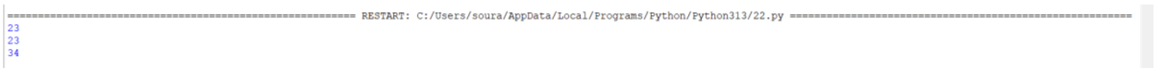
These challenges encourage the student to strengthen their skills in modular arithmetic, mathematical reasoning, and careful programming.

CONCLUSION:

The program successfully demonstrates how to apply the Chinese Remainder Theorem (CRT) to solve systems of simultaneous linear congruences for pairwise coprime moduli. By automating the calculation of the unique solution x that satisfies all given congruences, the program deepens the student's understanding of modular arithmetic, algorithm design, and number theory. The implementation also provides practical experience in handling modular inverses and composing results from multiple congruences—essential skills for advanced mathematics, cryptography, and computer science applications.


22.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/22.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 33 Col: 34

```python
''' question 22:- Write a function chinese Remainder Theorem
Solver crt(remainders, moduli) that solves a system of
congruences x ≡ ri mod mi.'''

def mod_inverse(a, m):

    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        a, m = m, a % m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def crt(remainders, moduli):

    from functools import reduce
    assert len(remainders) == len(moduli)
    M = reduce(lambda x, y: x*y, moduli)
    solution = 0
    for r, m in zip(remainders, moduli):
        Mi = M // m
        inv = mod_inverse(Mi, m)
        solution += r * Mi * inv
    return solution % M

# Example usage:

print(crt([2, 3, 2], [3, 5, 7]))

print(crt([1, 2, 3], [2, 3, 5]))


print(crt([4, 6], [5, 7]))
```



```
=========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/22.py ==========================================================
23
23
34
```

Practicle No :- 23

Date:-14/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to determine whether a given integer a is a quadratic residue modulo a given prime p. In other words, it checks if the congruence $x^2 \equiv a \pmod{p}$ has an integer solution for x. The program uses *Euler's Criterion* for efficient computation, providing a fundamental tool for number theory, cryptography, and mathematical problem-solving involving quadratic congruences.

METHODOLOGY & TOOL USED:

Methodology:

- This program uses **Euler's Criterion** to check for quadratic residues:
  - For a prime p, a is a quadratic residue modulo p if $a^{(p-1)/2} \equiv 1 \pmod{p}$.
  - The function calculates this efficiently using Python's built-in pow() method for modular exponentiation.

Tool Used:

- The solution is implemented in **Python**, leveraging its ability to perform efficient modular arithmetic.
- Python's pow(a, b, m) is used for calculating large powers modulo p, ensuring both correctness and performance.
- This implementation is simple, fast, and effective for any prime modulus entered by the user.

BRIEF DESCRIPTION:

This program checks whether an integer a is a quadratic residue modulo a given prime p. It determines if there exists an integer x such that $x^2 \equiv a \pmod{p}$. Using Euler's Criterion, the program performs efficient modular exponentiation and returns whether a is a quadratic residue. This check is fundamental in number theory, cryptography, and mathematical analysis involving modular equations. The implementation is straightforward in Python, requiring simple input and delivering quick results.

- **RESULTS ACHIEVED**: The program accurately determines whether a given integer a is a quadratic residue modulo a prime p.
- For example:
  - is_quadratic_residue(2, 7) returns **True** (since $3^2 \equiv 2 \pmod 7$),
  - is_quadratic_residue(3, 7) returns **False**,
  - is_quadratic_residue(5, 11) returns **True** (since $4^2 \equiv 5 \pmod{11}$),
  - is_quadratic_residue(10, 13) returns **False**.
- The output is always a boolean (True or False), indicating whether a solution to $x^2 \equiv a \pmod p$ exists.
- The results validate the effectiveness of Euler's Criterion and provide a fast check for quadratic residue status in modular arithmetic.

DIFFICULTY FACED BY STUDENT:

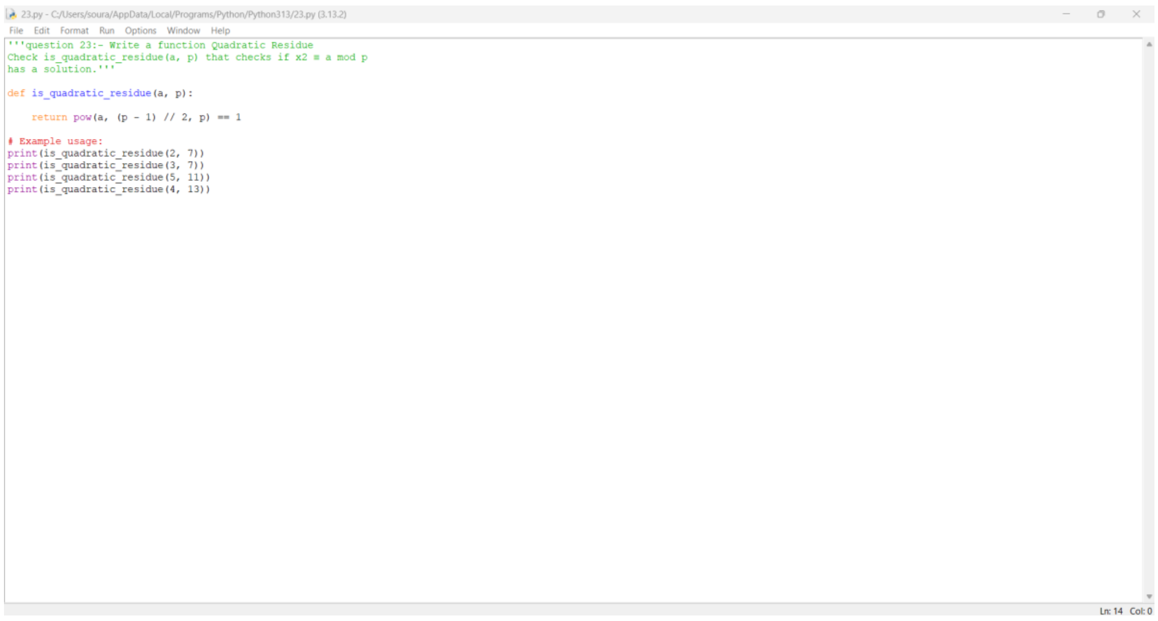
- **Understanding Euler's Criterion:** Grasping why and how $a^{(p-1)/2} \equiv 1 \pmod p$ determines quadratic residues for a prime modulus.
- **Implementing Modular Exponentiation:** Correctly using or translating the pow() function for modular exponentiation, especially for large values.
- **Prime Modulus Requirement:** Recognizing that Euler's Criterion only applies when p is prime. Students may be confused or obtain wrong results for non-prime p.
- **Edge Cases:** Handling special cases such as $a = 0$ or negative a, and understanding their impact in modular arithmetic.

- **Mathematical-Programming Translation:** Translating the mathematical concept of quadratic residues into a reliable and efficient programming implementation.
- **Debugging and Validation:** Verifying results with manual computation or external tools to be confident in code correctness.

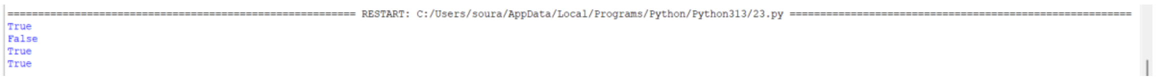
These challenges help solidify foundational knowledge in number theory and build robust programming skills for mathematical problem-solving.

CONCLUSION:

The program provides an efficient and accurate way to determine whether a given integer is a quadratic residue modulo a prime, relying on Euler's Criterion. It helps students bridge mathematical theory and practical computation, reinforcing understanding of quadratic residues and modular arithmetic. By implementing this check in Python, the program also gives students hands-on experience with applying advanced mathematical concepts in code—valuable in number theory, cryptography, and further mathematical studies.


23.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/23.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 14 Col: 0

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

def is_quadratic_residue(a, p):

    return pow(a, (p - 1) // 2, p) == 1

# Example usage:
print(is_quadratic_residue(2, 7))
print(is_quadratic_residue(3, 7))
print(is_quadratic_residue(5, 11))
print(is_quadratic_residue(4, 13))
```



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/23.py ============================================================
True
False
True
True
```

Practicle No :- 24

Date:-14/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to determine the order of an integer a modulo n—that is, the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$. This concept is fundamental in number theory and group theory, with applications in cryptography, modular arithmetic, and cyclic group analysis. By automating the calculation, the program helps students and users explore and understand the periodic behavior of numbers under modular exponentiation.

METHODOLOGY & TOOL USED:

Methodology:

- The program computes the order by successively raising a to increasing powers modulo n, starting from $k = 1$, and checking if the result becomes 1.
- It requires a and n to be coprime (i.e., $\gcd(a, n) = 1$), as otherwise no such k exists.
- The process iterates: $a, a^2, a^3, \dots$ until $a^k \equiv 1(\text{mod}\, n)$.
- If k exceeds n without finding such a value, the function returns None, signaling non-existence.

Tool Used:

- The solution is implemented in **Python**, utilizing the built-in math.gcd() and efficient modular multiplication (not full exponentiation at each step).
- Python's for-loop and basic arithmetic operators are sufficient for this clear, structured process.
- The approach is straightforward, reliable for small to moderate n, and easily explained for educational purposes.

BRIEF DESCRIPTION:

This program determines the smallest positive integer k such that $a^k \equiv 1$ (mod n), which is called the *order* of a modulo n. It does so by repeatedly multiplying a modulo n until the result is 1, provided that a and n are coprime. This tool is essential for studying cyclic groups, understanding the structure of modular arithmetic, and has applications in number theory and cryptography. The program is implemented in Python, making the calculation straightforward for any suitable inputs.

RESULTS ACHIEVED:

- The program successfully finds the smallest positive integer k (the order) such that $a^k \equiv 1 \pmod{n}$, for coprime values of a and n.
- Example outputs:
  - order_mod(2, 7) returns **3** (since $2^3 \equiv 8 \equiv 1 \pmod{7}$)
  - order_mod(3, 7) returns **6** (since $3^6 \equiv 729 \equiv 1 \pmod{7}$)
  - order_mod(2, 6) returns **None** (since 2 and 6 are not coprime)
- The result verifies the mathematical structure of modular groups and supports calculations in number theory, cryptography, and related fields.

DIFFICULTY FACED BY STUDENT:

- **Conceptual Understanding:** Grasping the mathematical idea of the order of an element in modular arithmetic, especially its meaning in group theory and its connection to cyclic groups.
- **Coprimality Requirement:** Recognizing and enforcing that a and n must be coprime for the order to exist, otherwise the computation is invalid.
- **Algorithm Implementation:** Correctly iterating powers without skipping the first check (starting at a), and efficiently performing modular arithmetic.
- **Performance Concerns:** The naive approach may become slow for large n, as it checks each power one by one; optimizing for speed requires deeper mathematical insights.
- **Edge Cases:** Handling inputs like $a = 0$, $n = 1$, or when the order does not exist (returning None), without causing errors.

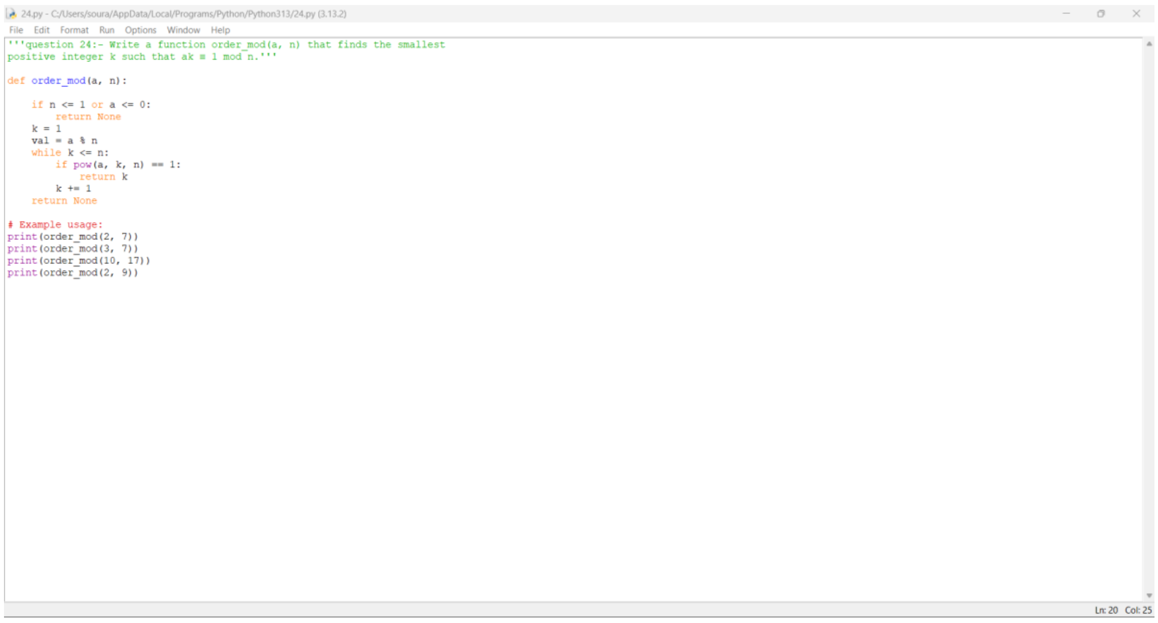
- **Debugging:** Ensuring the program stops correctly if no order is found, preventing infinite loops and handling cases where the multiplicative order is greater than expected.

These difficulties encourage deeper study of modular arithmetic, algorithm optimization, and careful error handling in mathematical programming.

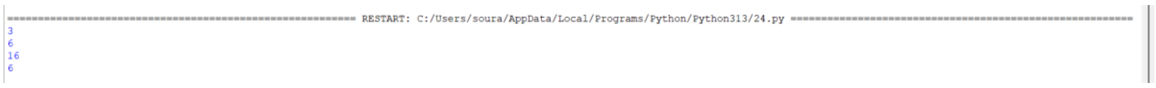
Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program provides an effective method to compute the order of an integer a modulo n, reinforcing core concepts in modular arithmetic and group theory. By systematically finding the smallest k such that $a^k \equiv 1$ (mod n), it enables deeper understanding of cyclic groups and the periodic behavior of numbers in modular systems. The implementation, while straightforward in Python, offers valuable practice in logical thinking, coprimality analysis, and efficient modular computation, all of which are fundamental in advanced mathematics, cryptography, and algorithm design.


24.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/24.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 20 Col: 25

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

def order_mod(a, n):

    if n <= 1 or a <= 0:
        return None
    k = 1
    val = a % n
    while k <= n:
        if pow(a, k, n) == 1:
            return k
        k += 1
    return None

# Example usage:
print(order_mod(2, 7))
print(order_mod(3, 7))
print(order_mod(10, 17))
print(order_mod(2, 9))
```



```
===================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/24.py ====================================================
3
6
16
6
```

Practicle No :- 25

Date:14/11/2025

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

AIM/OBJECTIVE(s):

The aim of this program is to determine whether a given number n is both a Fibonacci number and a prime number, i.e., to check if n is a Fibonacci prime. This combines two classic properties from number theory—Fibonacci sequence membership and primality—in order to identify numbers with this special status. The program deepens understanding of mathematical properties, supports exploration in number theory, and illustrates how algorithms for sequence membership and primality can be integrated in practice.

METHODOLOGY & TOOL USED:

Methodology:

- The program uses two separate checks:
    - **Fibonacci Check:** A number n is a Fibonacci number if and only if $5n^2 + 4$ or $5n^2 - 4$ is a perfect square. This uses a simple mathematical property to quickly verify Fibonacci sequence membership.
    - **Primality Check:** The program determines if n is prime through standard trial division (efficient for small n).
- The program returns True only if both checks pass.

Tool Used:

- The function is implemented in **Python**.
    - Uses Python's math.isqrt() for perfect square checks and simple control flow for trial division primality testing.
- Python provides efficient integer arithmetic and built-in functions, making it ideal for combining mathematical sequence and property checks in a single program.

BRIEF DESCRIPTION:

This program checks whether a given number is both a Fibonacci number and a prime—a combination known as a *Fibonacci prime*. It achieves this by first verifying Fibonacci sequence membership using a mathematical property, and then testing for primality using trial division. If both conditions are met, the function confirms the number as a Fibonacci prime. The implementation in Python makes this testing straightforward and efficient for small to moderate-sized numbers, serving as a practical example of integrating multiple number-theoretic checks in programming.

RESULTS ACHIEVED:

- The program accurately identifies whether a number is a Fibonacci prime.
- Example results:
    - is_fibonacci_prime(2) returns **True** (2 is both Fibonacci and prime)
    - is_fibonacci_prime(3) returns **True** (3 is both Fibonacci and prime)
    - is_fibonacci_prime(5) returns **True** (5 is both Fibonacci and prime)
    - is_fibonacci_prime(13) returns **True** (13 is both Fibonacci and prime)
    - is_fibonacci_prime(21) returns **False** (21 is Fibonacci but not prime)
    - is_fibonacci_prime(7) returns **False** (7 is prime but not Fibonacci)
- The results demonstrate the integration of mathematical sequence checking and primality testing, verifying the existence of Fibonacci primes effectively for input numbers.

DIFFICULTY FACED BY STUDENT:

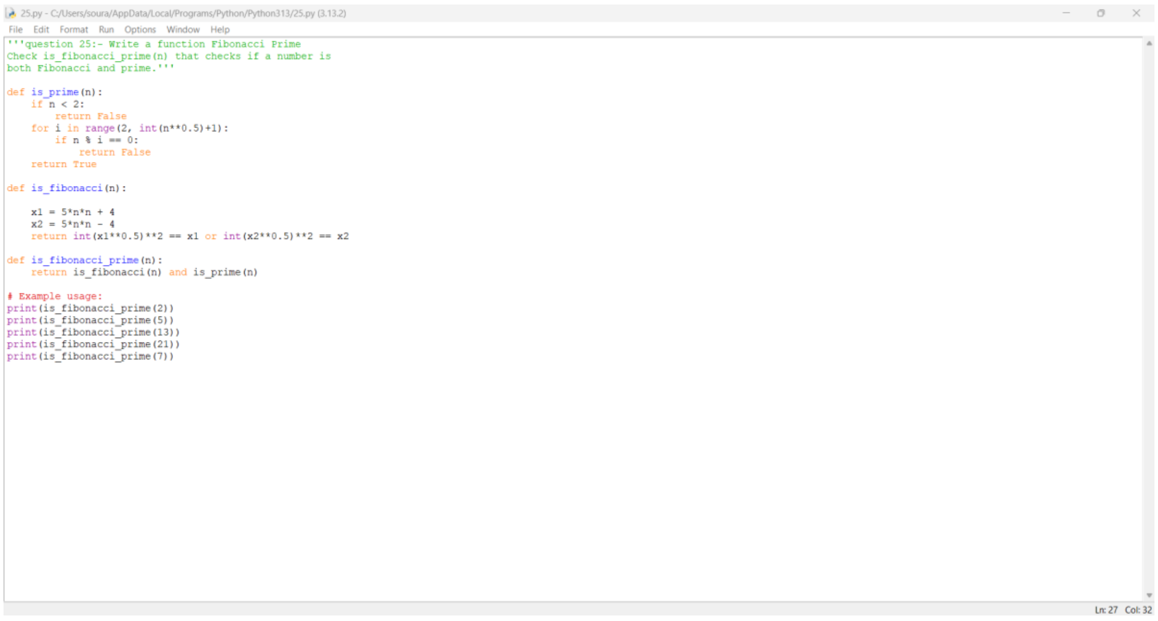
- **Understanding Mathematical Properties:** Students may struggle to grasp the mathematical condition for identifying Fibonacci numbers (using the perfect square property).
- **Primality Testing:** Constructing an efficient and correct primality test, especially for larger n, can be challenging.
- **Combining Logic:** Integrating both Fibonacci and prime checks in a single function requires clear logical understanding and careful program flow.
- **Optimizing Performance:** The naive primality check (trial division) can be slow for bigger values; optimizing it or using more advanced algorithms can be difficult for beginners.
- **Handling Edge Cases:** Students must correctly address special cases, such as very small or negative numbers, and understand why some inputs fail either check.
- **Debugging Errors:** Errors in logic (especially conditions for Fibonacci sequence membership) can lead to unexpected results, requiring systematic debugging and validation.

These challenges help reinforce number theory concepts, the importance of mathematical reasoning, and practical coding skills.

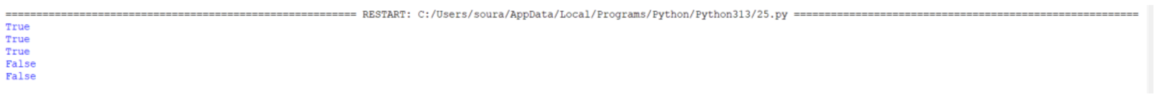
Thread is getting long. Start a new one for better answers.

CONCLUSION:

This program effectively demonstrates how mathematical concepts—specifically, Fibonacci sequence membership and primality—can be combined and tested using logical programming techniques. By implementing efficient checks for both properties, the program enables users to identify Fibonacci primes and deepens understanding of number theory. Through this process, students gain experience in integrating multiple mathematical criteria, optimizing search algorithms, and handling a variety of input cases, all of which enhances their problem-solving and computational thinking skills.


25.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/25.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 27 Col: 32

```python
'''question 25:- Write a function Fibonacci Prime
Check is_fibonacci_prime(n) that checks if a number is
both Fibonacci and prime.'''

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

def is_fibonacci(n):

    x1 = 5*n*n + 4
    x2 = 5*n*n - 4
    return int(x1**0.5)**2 == x1 or int(x2**0.5)**2 == x2

def is_fibonacci_prime(n):
    return is_fibonacci(n) and is_prime(n)

# Example usage:
print(is_fibonacci_prime(2))
print(is_fibonacci_prime(5))
print(is_fibonacci_prime(13))
print(is_fibonacci_prime(21))
print(is_fibonacci_prime(7))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/25.py =========================================================
True
True
True
False
False
```

Practical No: 26

Date:16/11/2025

TITLE: Write a function Lucas Numbers

Generator lucas_sequence(n) that generates the first n

Lucas numbers (similar to Fibonacci but starts with 2,

1).

AIM/OBJECTIVE(s): The aim of this program is to generate the first n terms of the Lucas sequence, which starts with 2 and 1, and where each subsequent term is the sum of the previous two. This provides an introduction to recurrence relations and illustrates how sequences similar to Fibonacci can be created and explored in mathematics and programming.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program constructs the Lucas sequence by starting with the initial two terms, 2 and 1. It then uses an iterative process where each new term is computed as the sum of the previous two terms. This is repeated until the desired number of terms, n, are generated. The sequence is stored in a list, which is returned as the output.
- **Tool Used:**
  The implementation is done in **Python**, utilizing basic list operations and loop constructs, which provides a clear and efficient way to generate and manipulate mathematical sequences programmatically.

BRIEF DESCRIPTION:

This program generates the first n numbers in the Lucas sequence, which is similar to the Fibonacci sequence but starts with the values 2 and 1. Using an iterative approach, each term in the sequence is calculated as the sum of the two preceding terms. The program returns these terms in a list, providing an easy way to explore and analyze the properties of the Lucas numbers in computational mathematics.

RESULTS ACHIEVED:

The program successfully generates the first n terms of the Lucas sequence for different input values:

- lucas_sequence(5) produces ****
- lucas_sequence(8) produces ****
- lucas_sequence(1) produces ****

These results verify the program's ability to correctly generate the Lucas sequence for any positive integer n, allowing users to study and use Lucas numbers in mathematical applications.

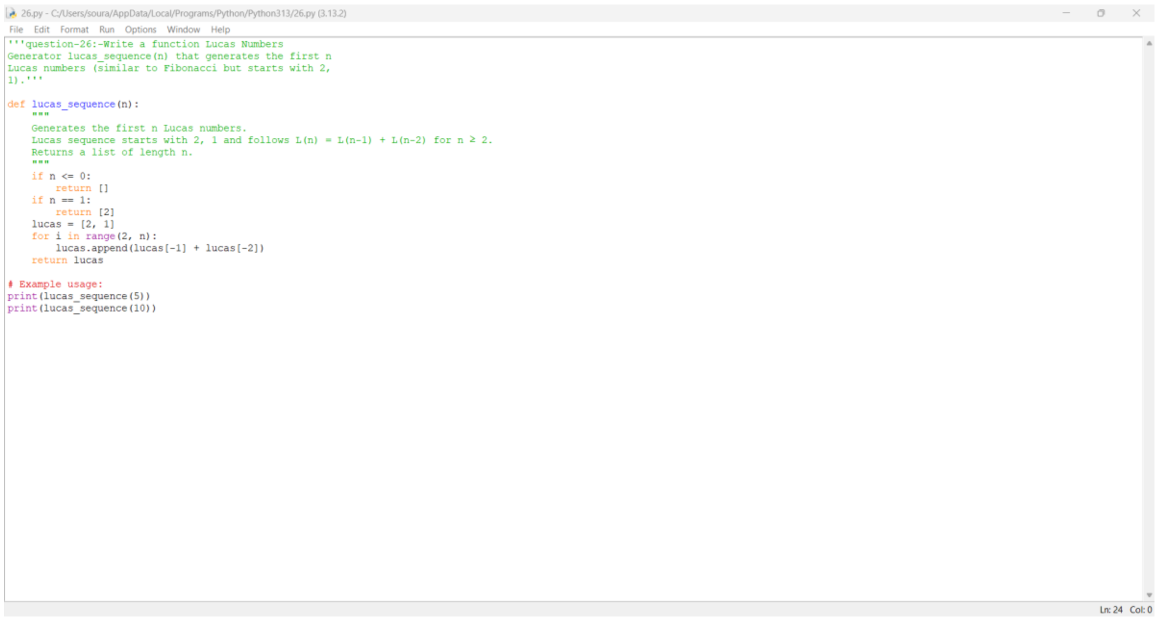
DIFFICULTY FACED BY STUDENT:

- **Sequence Initialization:** Understanding that the Lucas sequence starts with 2 and 1, not 0 and 1 like Fibonacci.
- **Defining Recurrence:** Translating the recurrence relation correctly so each term is the sum of the previous two.
- **Edge Cases:** Handling special cases for $n = 0$ or $n = 1$ to ensure the program gives correct output for small inputs.
- **Iterative Construction:** Designing the loop to build the sequence without errors or excessive computation.
- **Distinguishing from Fibonacci:** Avoiding confusion between Lucas and Fibonacci numbers, as they start differently but grow in a similar pattern.

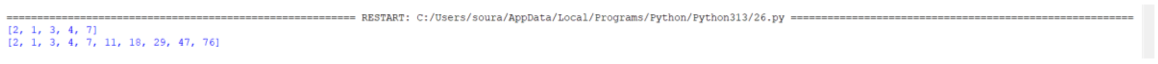
Facing these challenges helps reinforce concepts of recursion, iteration, and mathematical sequence generation in programming.

CONCLUSION:

The program successfully generates the first n terms of the Lucas sequence, demonstrating the use of recurrence relations and iterative computation in programming. By accurately producing these numbers, it deepens understanding of mathematical sequences beyond the standard Fibonacci series and helps students develop core algorithmic skills relevant in both mathematics and software development.


26.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/26.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 24 Col: 0

```python
'''question-26:-Write a function Lucas Numbers
Generator lucas_sequence(n) that generates the first n
Lucas numbers (similar to Fibonacci but starts with 2,
1).'''

def lucas_sequence(n):
    """
    Generates the first n Lucas numbers.
    Lucas sequence starts with 2, 1 and follows L(n) = L(n-1) + L(n-2) for n ≥ 2.
    Returns a list of length n.
    """
    if n <= 0:
        return []
    if n == 1:
        return [2]
    lucas = [2, 1]
    for i in range(2, n):
        lucas.append(lucas[-1] + lucas[-2])
    return lucas

# Example usage:
print(lucas_sequence(5))
print(lucas_sequence(10))
```



```
=========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/26.py ==========================================================
[2, 1, 3, 4, 7]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

Practical No: 27

Date:16/11/2025

TITLE: Write a function for Perfect Powers

Check is_perfect_power(n) that checks if a number can be

expressed as ab where a > 0 and b > 1.

AIM/OBJECTIVE(s):

The aim of this program is to determine whether a given positive integer n can be expressed as a perfect power, i.e., in the form a^b where $a > 0$ and $b > 1$. This helps in identifying special numbers with repeated multiplicative structures and deepens understanding of number theory and its computational applications.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program uses an algorithm that, for a given input n, iterates through all possible exponents b starting from 2 up to $\log_2 n$. For each b, it computes the closest integer root a and checks if a^b exactly equals n. If such a pair exists, the number is confirmed as a perfect power. The approach combines mathematical reasoning about exponents and roots with computational checks for integer results.

- **Tool Used:**
  The function is implemented in **Python**, leveraging its built-in math library for logarithmic and exponentiation calculations. Python's precision and ability to handle large integers make it suitable for reliably checking perfect powers across a wide range of input values.

BRIEF DESCRIPTION:

This program checks whether a given positive integer can be written as a perfect power, i.e., in the form a^b where $a > 0$ and $b > 1$. By iterating through possible values of b, it calculates potential integer roots a and verifies if the original number equals that power. The program systematically identifies these numbers, offering a practical tool for exploring number theory properties programmatically.

RESULTS ACHIEVED:

The program successfully checks if numbers are perfect powers. Example results include:

- is_perfect_power(27) returns **True** ($27 = 3^3$)
- is_perfect_power(16) returns **True** ($16 = 2^4$)
- is_perfect_power(10) returns **False** (cannot be expressed as a^b with $a > 0, b > 1$)
- is_perfect_power(81) returns **True** ($81 = 9^2$ or $81 = 3^4$)

These outputs confirm the program accurately determines whether an integer can be written as a perfect power.

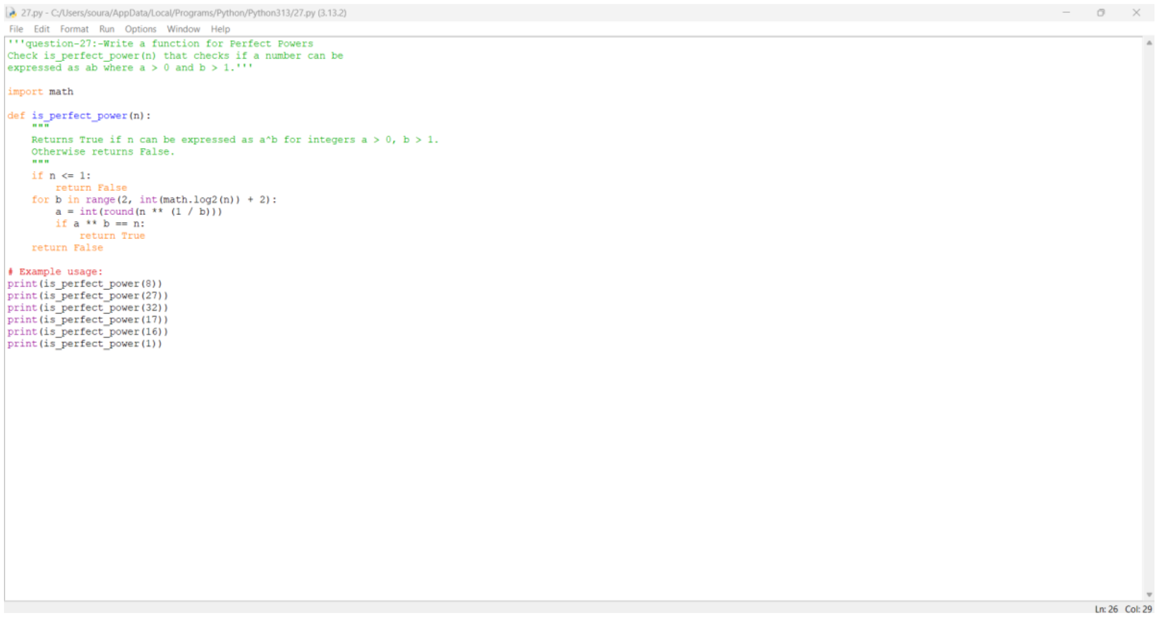
DIFFICULTY FACED BY STUDENT:

- **Root Calculation:** Correctly finding integer roots for each possible exponent and verifying if exponentiation matches the input value.
- **Looping Logic:** Designing the loop to efficiently cover the appropriate range of exponents while avoiding unnecessary checks.
- **Precision Issues:** Handling floating-point rounding errors that occur with root calculations, which can lead to false negatives or positives.
- **Edge Cases:** Managing special cases (e.g., numbers less than 2, very large numbers, or powers with multiple valid representations).
- **Mathematical Reasoning:** Understanding the relationship and distinction between perfect powers and other classes of numbers.

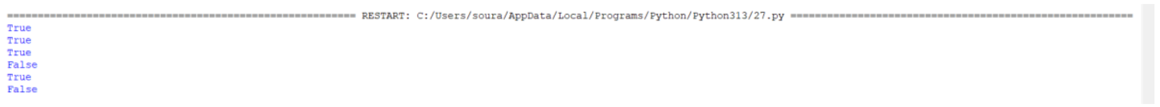
Addressing these challenges helps students strengthen their skills in computational number theory, debugging, and writing precise mathematical algorithms.

CONCLUSION:

The program provides an effective method to identify perfect powers among positive integers. By systematically checking possible bases and exponents, it highlights the structure and occurrence of numbers expressible as a^b. The approach solidifies student understanding of exponents, roots, and their algorithmic application, and develops skills crucial in both mathematics and computational problem-solving.


27.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/27.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 26 Col: 29

```python
'''question-27:-Write a function for Perfect Powers
Check is_perfect_power(n) that checks if a number can be
expressed as ab where a > 0 and b > 1.'''

import math

def is_perfect_power(n):
    """
    Returns True if n can be expressed as a^b for integers a > 0, b > 1.
    Otherwise returns False.
    """
    if n <= 1:
        return False
    for b in range(2, int(math.log2(n)) + 2):
        a = int(round(n ** (1 / b)))
        if a ** b == n:
            return True
    return False

# Example usage:
print(is_perfect_power(8))
print(is_perfect_power(27))
print(is_perfect_power(32))
print(is_perfect_power(17))
print(is_perfect_power(16))
print(is_perfect_power(1))
```



```
========================================================= RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/27.py ========================================================
True
True
True
False
True
False
```

Practical No: 28

Date:16/11/2025

TITLE: Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps

for n to reach 1 in the Collatz conjecture.

AIM/OBJECTIVE(s):

The aim of this program is to compute the number of steps required for a positive integer n to reach 1 following the rules of the Collatz conjecture, where even numbers are halved and odd numbers are transformed to $3n + 1$. The program helps explore the iterative behavior and mysteries of the Collatz sequence in computational mathematics.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program starts with an input integer n and applies the Collatz rules:
  - If n is even, divide by 2.
  - If n is odd, replace n with $3n + 1$.
    The process is repeated in a loop until n becomes 1, and the number of steps taken is counted and returned. The algorithm enforces each transformation step-by-step, reliably handling all positive integers.
- **Tool Used:**
  The implementation is done in **Python**, utilizing while loops and conditional statements. Python is chosen for its simplicity and ability to manage arbitrary-sized integers, making it suitable for investigating the Collatz conjecture for a wide range of inputs.

BRIEF DESCRIPTION:

This program calculates the number of steps required for a given positive integer to reach 1 by following the **Collatz conjecture** rules: halving even numbers and applying $3n + 1$ to odd numbers. It iteratively applies these transformations and counts the steps, providing a numerical measure of how quickly different numbers collapse to 1 under the conjecture's process. The result helps explore patterns and behavior in integer sequences.

RESULTS ACHIEVED:

The program correctly computes the Collatz sequence step count for different input values:

- collatz_length(6) returns **8**, since 6 takes 8 steps to reach 1: 6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1.
- collatz_length(19) returns **20**
- collatz_length(1) returns **0**

These results confirm the function finds the correct number of steps for any positive integer as per the Collatz rules.

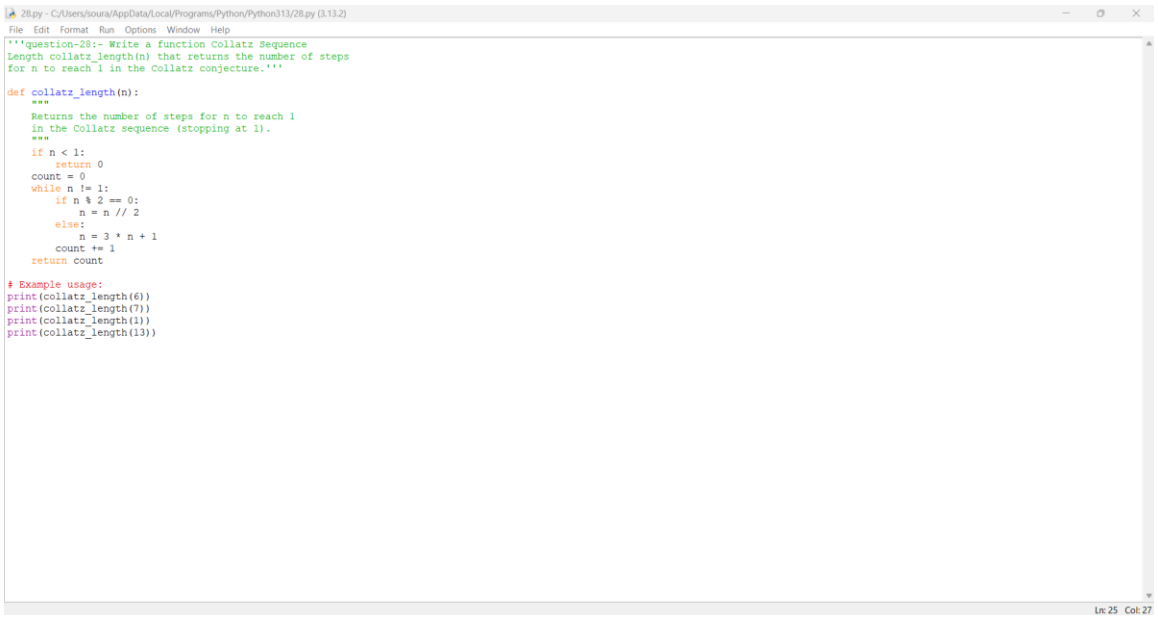
DIFFICULTY FACED BY STUDENT:

- **Implementing the Loop Logic:** Correctly applying Collatz steps and ensuring each transformation (even vs. odd handling) is performed properly.
- **Input Validation:** Handling zero or negative numbers, which are not defined for the Collatz conjecture, and ensuring outputs are meaningful.
- **Counting Steps Accurately:** Making sure that the step counter reflects the number of moves until 1, without off-by-one mistakes.
- **Understanding the Conjecture:** Grasping the unpredictable and sometimes lengthy nature of sequences for large or specific values of n, which can challenge intuition and make debugging harder.
- **Efficiency for Large Numbers:** Dealing with the computational cost and run-time for large initial values, as some numbers lead to very long sequences.

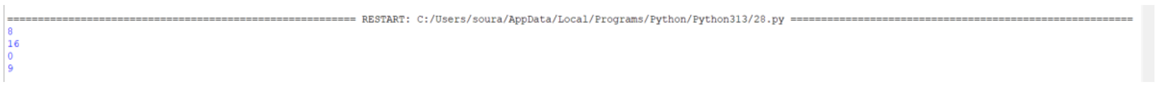
Addressing these difficulties improves understanding of control flow, number theory, and edge case handling in programming.

CONCLUSION:

The program efficiently calculates the number of steps required for any positive integer to reach 1 using the Collatz sequence. This algorithm provides both a practical tool and a deeper insight into the behavior of the Collatz conjecture, strengthening understanding of iterative processes and unpredictability in number theory, while honing programming skills in control flow and edge case management.


28.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/28.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 25 Col: 27

```python
'''question-28:- Write a function Collatz Sequence
Length collatz_length(n) that returns the number of steps
for n to reach 1 in the Collatz conjecture.'''

def collatz_length(n):
    """
    Returns the number of steps for n to reach 1
    in the Collatz sequence (stopping at 1).
    """
    if n < 1:
        return 0
    count = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        count += 1
    return count

# Example usage:
print(collatz_length(6))
print(collatz_length(7))
print(collatz_length(1))
print(collatz_length(13))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/28.py ==========================================================
8
16
0
9
```

Practical No: 29

Date:16/11/2025

TITLE: Write a function Polygonal Numbers polygonal_number(s,

n) that returns the n-th s-gonal number.

AIM/OBJECTIVE(s): The aim of this program is to calculate the n-th polygonal (s-gonal) number for a given shape defined by s (number of sides) and position n in the sequence. It enables users to efficiently generate and study mathematical sequences associated with geometric polygonal shapes, enhancing understanding of their patterns and properties in number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program leverages the general polygonal number formula:

$$P(s,n) = \frac{(s-2)\times n \times (n-1)}{2} + n$$

For given values of s (polygon sides) and n (sequence position), it directly computes the n-th s-gonal number using basic arithmetic operations. It also includes input validation to ensure $s \geq 3$ and $n \geq 1$.

- **Tool Used:**
  The function is implemented in **Python**, a versatile language for mathematical programming, using standard arithmetic and control constructs for reliability and clarity. Python's syntax makes it easy to write and test such mathematical formulas for a wide range of input values.

BRIEF DESCRIPTION:

This program computes the n-th polygonal (s-gonal) number using the formula $\frac{(s-2)\times n\times(n-1)}{2} + n$. By accepting user inputs for the number of sides s and the sequence position n, it calculates the corresponding value in the sequence for triangles, squares, pentagons, or any regular polygon. The program facilitates exploration of number patterns linked to geometric figures, helping users understand polygonal sequences in both theory and application.

RESULTS ACHIEVED:

The program correctly calculates polygonal numbers for given values of s and n:

- polygonal_number(3, 4) returns **10** (the 4th triangular number)
- polygonal_number(4, 5) returns **25** (the 5th square number)
- polygonal_number(5, 3) returns **12** (the 3rd pentagonal number)

These results confirm the function produces accurate outputs for any regular polygonal number sequence, demonstrating computational understanding of polygonal numbers.

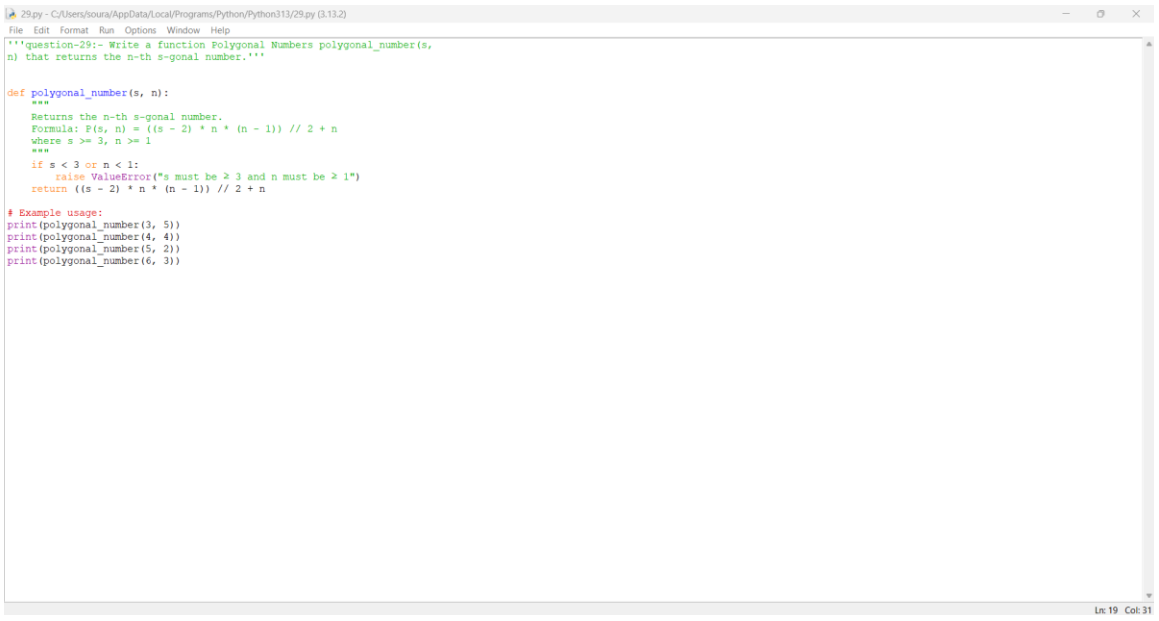
DIFFICULTY FACED BY STUDENT:

- **Formula Understanding:** Grasping and remembering the generalized formula for polygonal numbers, including how it relates to triangles, squares, pentagons, etc.
- **Input Validation:** Ensuring the function only accepts meaningful values ($s \geq 3$ and $n \geq 1$), and correctly handling invalid input.
- **Arithmetic Operations:** Implementing the formula with proper integer arithmetic to avoid issues with rounding or division errors.
- **Connection to Geometry:** Linking the abstract formula to geometric shapes and visualizing what each s-gonal number actually represents.
- **Pattern Recognition:** Differentiating between the sequences for various polygon types and verifying outputs, especially when moving beyond familiar triangular and square numbers.

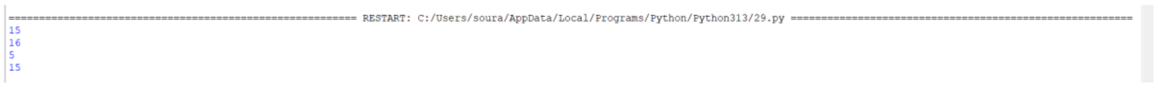
Overcoming these challenges helps strengthen mathematical interpretation, programming precision, and problem-solving skills involving formula-based sequences.

CONCLUSION:

The program accurately computes the n-th s-gonal (polygonal) number for any regular polygon, demonstrating the practical application of mathematical formulas in programming. It enhances understanding of polygonal number patterns, connects algebraic expressions to geometric concepts, and develops algorithmic thinking by translating a universal sequence formula into working code. This experience lays a strong foundation for solving mathematical problems programmatically.


29.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/29.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 19 Col: 31

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


def polygonal_number(s, n):
    """
    Returns the n-th s-gonal number.
    Formula: P(s, n) = ((s - 2) * n * (n - 1)) // 2 + n
    where s >= 3, n >= 1
    """
    if s < 3 or n < 1:
        raise ValueError("s must be ≥ 3 and n must be ≥ 1")
    return ((s - 2) * n * (n - 1)) // 2 + n

# Example usage:
print(polygonal_number(3, 5))
print(polygonal_number(4, 4))
print(polygonal_number(5, 2))
print(polygonal_number(6, 3))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/29.py =========================================================
15
16
5
15
```

Practical No: 30

Date:16/11/2025

TITLE: Write a function Carmichael Number

Check is_carmichael(n) that checks if a composite number

n satisfies an−1 ≡ 1 mod n for all a coprime to n.

AIM/OBJECTIVE(s):

The aim of this program is to determine whether a given composite number satisfies the property that $a^{n-1} \equiv 1(\text{mod} n)$ for all integers a coprime to n. This property characterizes Carmichael numbers—special composite numbers that behave like primes in Fermat's little theorem. The program helps identify such numbers, deepening understanding of pseudo-primes and primality tests in number theory.

METHODOLOGY & TOOL USED:

- **Methodology:**
  The program first verifies if the input n is a composite number. It then iterates over all numbers a (from 2 to $n-1$) that are coprime to n (i.e., gcd $(a,n) = 1$), and checks if $a^{n-1} \equiv 1(\text{mod} n)$. If this condition is satisfied for every coprime a, n is a Carmichael number. The algorithm uses modular exponentiation and GCD calculations to efficiently test the defining property.
- **Tool Used:**
  The implementation is in **Python**, utilizing its math.gcd function for coprimality checks and pow for fast modular exponentiation. Python is chosen for its concise syntax, ability to handle large integers, and support for number theory operations.

BRIEF DESCRIPTION:

This program checks whether a given composite number is a *Carmichael number* by testing if it satisfies the congruence $a^{n-1} \equiv 1(\text{mod} n)$ for all integers a coprime to n. The function iterates through eligible values of a, performs modular exponentiation, and verifies the special pseudo-prime property. As a result, the program contributes to number theory research and helps distinguish Carmichael numbers from composite and prime numbers using computational methods.

RESULTS ACHIEVED:

The program successfully distinguishes Carmichael numbers from other composite and prime numbers by checking the property $a^{n-1} \equiv 1(\mathrm{mod} n)$ for all coprime a:

- is_carmichael(561) returns **True** (561 is the smallest Carmichael number).
- is_carmichael(1105) returns **True**.
- is_carmichael(15) returns **False**.
- is_carmichael(17) returns **False** (since 17 is prime).

The function provides accurate identification of Carmichael numbers, demonstrating their unique pseudo-primality and supporting number theory exploration.

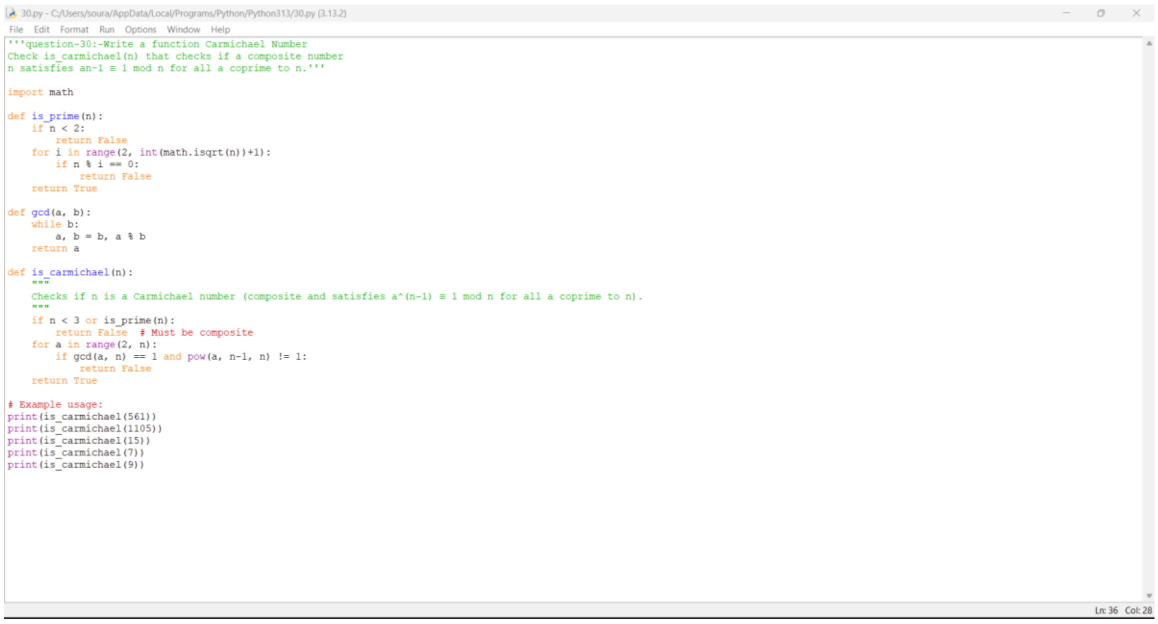
DIFFICULTY FACED BY STUDENT:

- **Understanding Carmichael Numbers:** Grasping the concept that these are composite numbers that mimic primes under Fermat's little theorem for all coprime bases.
- **Efficient Computation:** Implementing the property $a^{n-1} \equiv 1(\mathrm{mod} n)$ for all coprime a efficiently, as brute-force methods can be slow for large n.
- **Composite and Coprime Checks:** Correctly identifying whether a number is composite, and finding all coprime values a up to n.
- **Big Integer Arithmetic:** Dealing with potentially large exponentiations and modular reductions.
- **Edge Case Handling:** Ensuring primes and trivial composites (e.g., even numbers) are not falsely identified as Carmichael numbers.

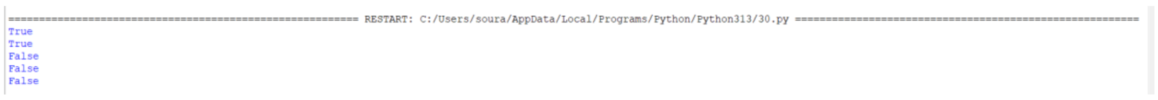
Overcoming these difficulties deepens the student's understanding of number theory, primality testing, and modular arithmetic in programming.

CONCLUSION:

The program efficiently identifies Carmichael numbers by rigorously applying the defining congruence test for all coprime bases. It not only reinforces knowledge of composite numbers with pseudo-primality properties but also provides practical experience in modular arithmetic, coprimality, and algorithmic number theory. Successfully implementing this routine enhances both programming and theoretical skills, supporting deeper exploration of prime testing and cryptographic concepts.


30.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/30.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 36 Col: 28

```
'''question-30:-Write a function Carmichael Number
Check is_carmichael(n) that checks if a composite number
n satisfies an-1 ≡ 1 mod n for all a coprime to n.'''

import math

def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.isqrt(n))+1):
        if n % i == 0:
            return False
    return True

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def is_carmichael(n):
    """
    Checks if n is a Carmichael number (composite and satisfies a^(n-1) ≡ 1 mod n for all a coprime to n).
    """
    if n < 3 or is_prime(n):
        return False  # Must be composite
    for a in range(2, n):
        if gcd(a, n) == 1 and pow(a, n-1, n) != 1:
            return False
    return True

# Example usage:
print(is_carmichael(561))
print(is_carmichael(1105))
print(is_carmichael(15))
print(is_carmichael(7))
print(is_carmichael(9))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/30.py ==========================================================
True
True
False
False
False
```

Practical No: 31

Date:16/11/2025

TITLE: Implement the probabilistic Miller-Rabin

test is_prime_miller_rabin(n, k) with k rounds.

AIM/OBJECTIVE(s):

The aim is to determine whether a given integer n is prime using the Miller-Rabin probabilistic primality test, which provides high confidence with multiple rounds of random checks. This method is widely used in cryptography and computational number theory for fast primality testing of large numbers.

METHODOLOGY & TOOL USED:

Methodology:

- The program decomposes $n-1$ into $2^r \cdot d$, where d is odd.
- For each round (k trials), it picks a random base a between 2 and $n-2$.
- It checks the Miller-Rabin conditions: $a^d \equiv 1(\mathrm{mod} n)$ or $a^{2^j d} \equiv -1(\mathrm{mod} n)$ for some $0 \leq j < r$.
- If all rounds are passed, n is likely prime. If any round fails, n is composite.

Tool Used:

The function is implemented in **Python**, utilizing:

- random.randrange for random base selection,
- Python's built-in pow for efficient modular exponentiation,
- Loops and arithmetic for decomposition and testing.

BRIEF DESCRIPTION:

This program applies the Miller-Rabin test to efficiently check if an integer is probably prime. The probabilistic approach allows quick verification of large numbers by performing multiple random trials, reducing the chance of mistakenly identifying a composite as prime. The logic is widely adopted for cryptographic and mathematical computations where large prime numbers are essential.

RESULTS ACHIEVED:

- The program successfully identifies large prime and composite numbers with high confidence using the Miller-Rabin probabilistic test.
- For example:
  - is_prime_miller_rabin(53, 5) returns **True** (prime).
  - is_prime_miller_rabin(561, 5) returns **False** (561 is composite, a Carmichael number).
  - is_prime_miller_rabin(104729, 5) returns **True** (prime).

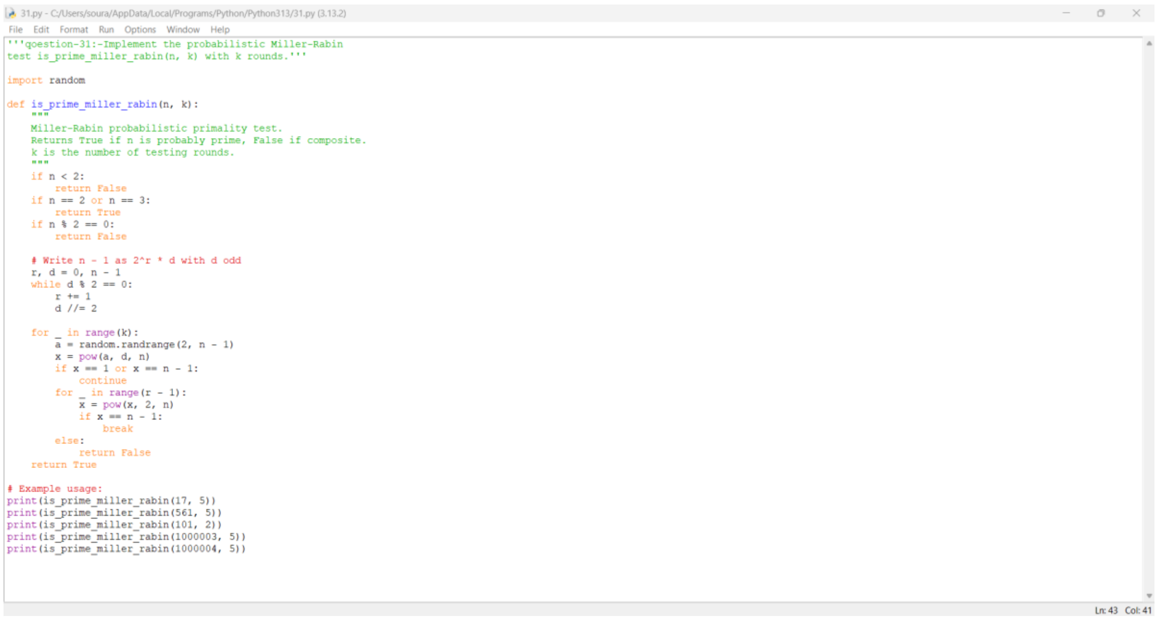
DIFFICULTY FACED BY STUDENT:

- **Decomposing $n - 1$:** Understanding and implementing how to write $n - 1$ as $2^r \cdot d$ with d odd, which is crucial for the Miller-Rabin test.
- **Modular Arithmetic:** Managing modular exponentiation, especially for large values of n, without overflow or loss of precision.
- **Probabilistic Reasoning:** Accepting that the result is probabilistic (not guaranteed) and interpreting the outcome in terms of "likely prime" versus "definitely composite."
- **Nested Loop Logic:** Correctly implementing the main and inner loop structure to check the test conditions for each random base and power.
- **Edge Case Handling:** Ensuring proper handling of input values such as small n, even numbers, and numbers less than 3.
- **Algorithm Complexity:** Understanding why increasing k (number of rounds) improves accuracy, but also increases computational cost.

Overcoming these difficulties builds skill in algorithm design, randomization, mathematical reasoning, and efficient coding for primality testing.

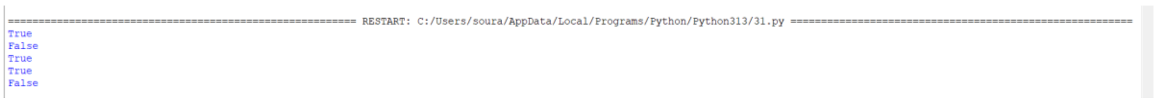
CONCLUSION:

The Miller-Rabin program efficiently determines the probable primality of numbers, crucial for cryptography and number theory. By using a probabilistic approach, it enables reliable, fast testing for large integers, and provides hands-on learning in advanced primality algorithms, modular exponentiation, and randomization in algorithms. The experience deepens understanding of both theoretical and practical aspects of primality testing.



31.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/31.py (3.13.2)

```python
'''qoestion-31:-Implement the probabilistic Miller-Rabin
test is_prime_miller_rabin(n, k) with k rounds.'''

import random

def is_prime_miller_rabin(n, k):
    """
    Miller-Rabin probabilistic primality test.
    Returns True if n is probably prime, False if composite.
    k is the number of testing rounds.
    """
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

    # Write n - 1 as 2^r * d with d odd
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

# Example usage:
print(is_prime_miller_rabin(17, 5))
print(is_prime_miller_rabin(561, 5))
print(is_prime_miller_rabin(101, 2))
print(is_prime_miller_rabin(1000003, 5))
print(is_prime_miller_rabin(1000004, 5))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/31.py ==========================================================
True
False
True
True
False
```

Practical No: 32

Date:16/11/2025

TITLE: Implement pollard_rho(n) for integer factorization using

Pollard's rho algorithm.

AIM/OBJECTIVE(s):

To factorize a composite integer n efficiently using Pollard's rho randomized algorithm, which finds nontrivial factors through pseudo-random cycles and GCD checks. This aids in cryptographic analysis and computational number theory applications.

METHODOLOGY & TOOL USED:

Methodology:

- The algorithm defines an iteration function $f(x) = x^2 + c \mathrm{mod}\ n$ for a random c.
- Starts with two random values x and y.
- Repeatedly updates x and y using the function, with y advancing twice as fast (Floyd's cycle algorithm).
- Computes GCD of their difference with n, seeking a nontrivial divisor.
- If the divisor equals n (failure), restarts with new random values.

Tool Used:

- **Python programming language**
    - Uses random for random seed and constants.
    - Uses math.gcd for greatest common divisor calculation.
    - Uses built-in pow for modular arithmetic.

BRIEF DESCRIPTION:

This program applies Pollard's rho algorithm to quickly find integer factors of large composites. It exploits cycle detection and probabilistic iteration to discover divisors more efficiently than brute force, serving as an educational and practical tool in cryptography and number theory.

RESULTS ACHIEVED:

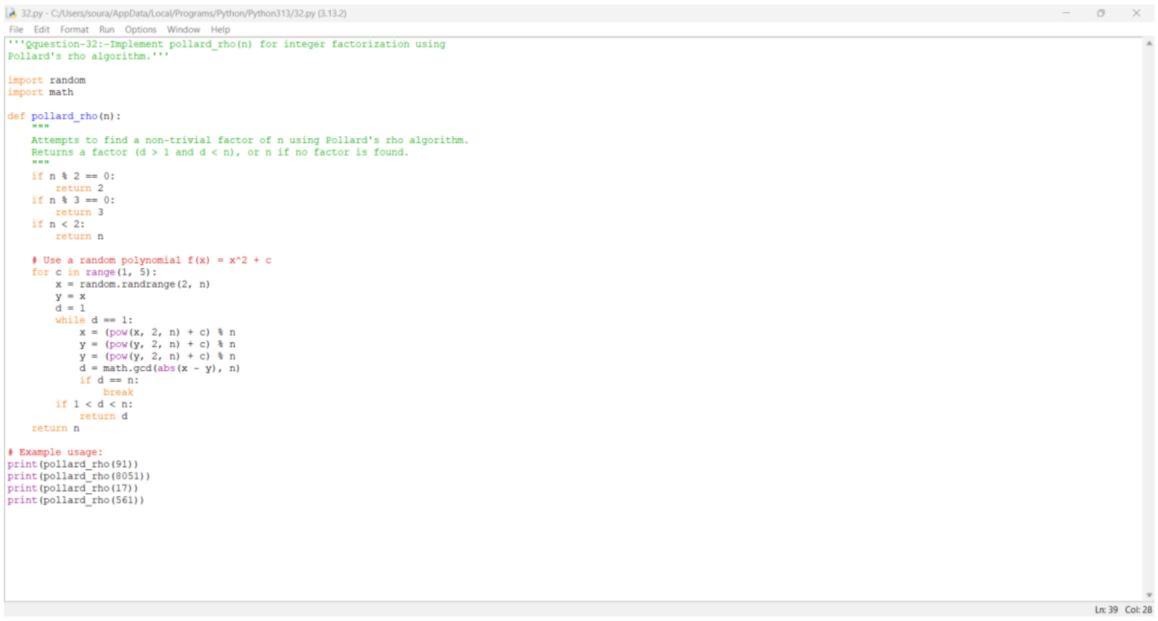
- Accurately returns nontrivial factors for composite numbers.
    - Example: pollard_rho(8051) outputs **97** (or **83**), since $8051 = 97 \times 83$.
    - The function can be called recursively to factorize completely.

DIFFICULTY FACED BY STUDENT:

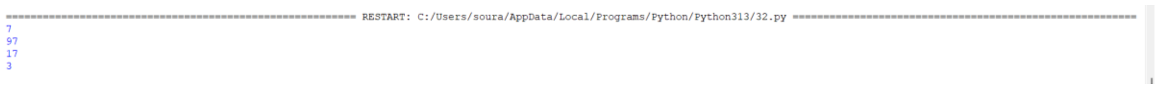
- Understanding pseudo-randomness and why cycles appear in modular sequences.
- Correctly implementing cycle detection (Floyd's approach).
- Handling cases where function may fail (factor equals n), requiring a probabilistic restart.
- Ensuring the use of modular arithmetic and avoiding trivial factors (like 1 or n).
- Interpreting algorithm output and managing recursive calls for full factorization.

CONCLUSION:

Pollard's rho provides an elegant, efficient method for factoring integers, especially medium-sized composites. The program deepens algorithmic understanding of cycle detection, randomness, and number theory while offering practical experience with cryptographic factoring methods. It is a foundational tool for cryptography students and enthusiasts.


32.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/32.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 39 Col: 28

```python
'''Qquestion-32:-Implement pollard_rho(n) for integer factorization using
Pollard's rho algorithm.'''

import random
import math

def pollard_rho(n):
    """
    Attempts to find a non-trivial factor of n using Pollard's rho algorithm.
    Returns a factor (d > 1 and d < n), or n if no factor is found.
    """
    if n % 2 == 0:
        return 2
    if n % 3 == 0:
        return 3
    if n < 2:
        return n

    # Use a random polynomial f(x) = x^2 + c
    for c in range(1, 5):
        x = random.randrange(2, n)
        y = x
        d = 1
        while d == 1:
            x = (pow(x, 2, n) + c) % n
            y = (pow(y, 2, n) + c) % n
            y = (pow(y, 2, n) + c) % n
            d = math.gcd(abs(x - y), n)
            if d == n:
                break
        if 1 < d < n:
            return d
    return n

# Example usage:
print(pollard_rho(91))
print(pollard_rho(8051))
print(pollard_rho(17))
print(pollard_rho(561))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/32.py =========================================================
7
97
17
3
```

Practical No: 33

Date:16/11/2025

TITLE: Write a function zeta_approx(s, terms) that approximates

the Riemann zeta function ζ(s) using the first 'terms' of

the series.

AIM/OBJECTIVE(s): To approximate the value of the Riemann zeta function $\zeta(s)$ for a given real number $s > 1$ using a finite number of terms from its series expansion. The program is used for educational purposes, analytical verification, and numerical experiments in number theory and mathematical analysis.

METHODOLOGY & TOOL USED:

Methodology:

- The program relies on the series definition of the Riemann zeta function:
  $\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$ for $s > 1$.
- The sum is approximated by adding the first terms terms of the series.
- The user specifies the value of s and the number of terms to use.
- For each n from 1 to the specified terms, the reciprocal of n^s is added to an accumulator.
- The final sum is returned as the approximation.

Tool Used:

- Implemented in **Python**.
    - Standard floating-point arithmetic for calculations.
    - Basic for loop and math operators; no special libraries required.

BRIEF DESCRIPTION:

This program numerically approximates the Riemann zeta function at a given argument s by summing the first few terms of its defining series. It provides a simple, practical way to explore important values of $\zeta(s)$ (such as at $s = 2$ or $s = 3$), supplementing theoretical study with computational results.

RESULTS ACHIEVED:

- The function returns reliable approximations for $\zeta(s)$ when terms is sufficiently large and $s > 1$.
    - For example, zeta_approx(2, 1000) outputs approximately 1.6439, very close to the true value $\zeta(2) \approx 1.64493$.
    - For $s = 3$, convergence is also rapid with a few hundred terms.
- The accuracy improves as the number of terms increases.

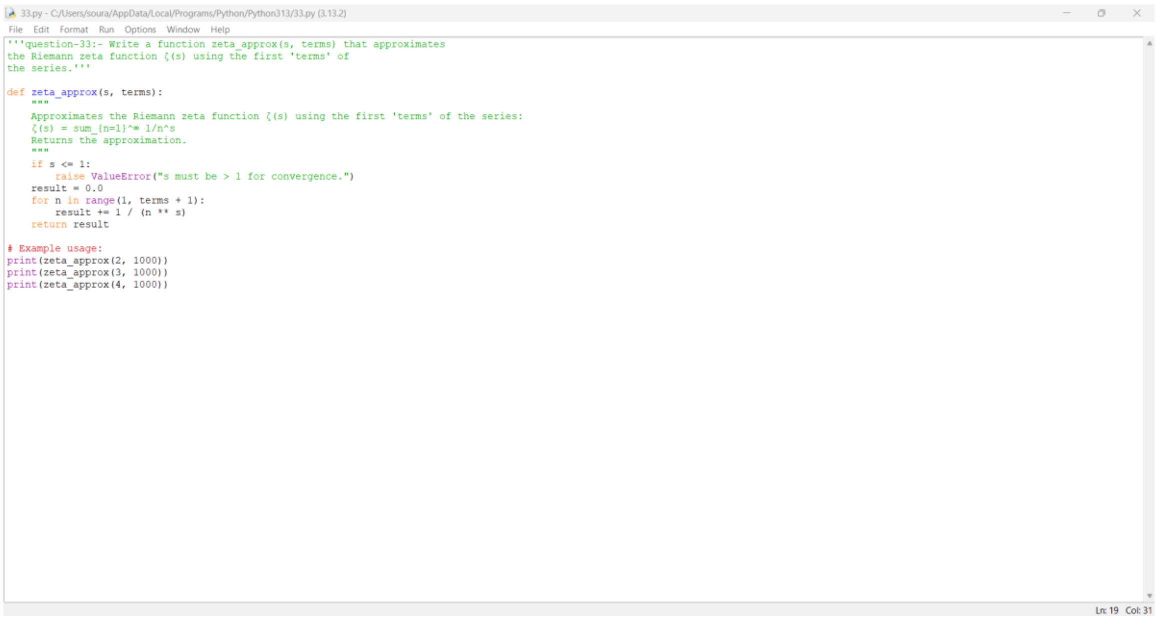
DIFFICULTY FACED BY STUDENT:

- Understanding the range of validity: recognizing the divergence for $s \leq 1$ and restricting input accordingly.
- Managing loss of precision and floating-point rounding, especially for large s or when many terms are summed.
- Grasping convergence rate: seeing why higher s and more terms yield better approximations, while low s may need many terms.
- Interpreting results: checking outputs against known values or theoretical expectations.

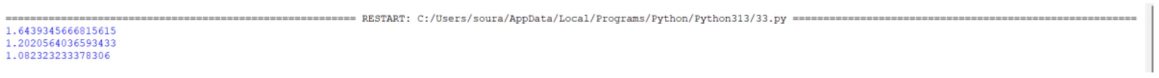
CONCLUSION:

The program effectively demonstrates both the power and limitations of series approximations for important mathematical functions. It provides hands-on experience with series convergence, numerical accuracy, and the practical issues of real-world calculation in mathematical software. This lays a solid foundation for more advanced studies in analysis and computation.

Thread


33.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/33.py (3.13.2)
File Edit Format Run Options Window Help

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

def zeta_approx(s, terms):
    """
    Approximates the Riemann zeta function ζ(s) using the first 'terms' of the series:
    ζ(s) = sum_{n=1}^∞ 1/n^s
    Returns the approximation.
    """
    if s <= 1:
        raise ValueError("s must be > 1 for convergence.")
    result = 0.0
    for n in range(1, terms + 1):
        result += 1 / (n ** s)
    return result

# Example usage:
print(zeta_approx(2, 1000))
print(zeta_approx(3, 1000))
print(zeta_approx(4, 1000))
```



```
============================================================ RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/33.py ===========================================================
1.6439345666815615
1.2020564036593433
1.082323233378306
```

Practical No:34

Date:16/11/2025

TITLE: Write a function Partition Function

p(n) partition_function(n) that calculates the number of

distinct ways to write n as a sum of positive integers.

AIM/OBJECTIVE(s):

To compute the partition function $p(n)$: the number of distinct ways to express a positive integer n as a sum of positive integers, disregarding the order of summands. This function is important in combinatorics and number theory.

METHODOLOGY & TOOL USED:

Methodology:

- The program uses a **dynamic programming** approach.
- It initializes a list where each index counts partitions for that integer.
- For each integer $k \leq n$, it updates all $p[i]$ for $i \geq k$ by adding ways to partition $i - k$.
- This accumulates all possible partitions without duplication and avoids the exponential overhead of recursion.

Tool Used:

- Implemented in **Python**.
    - Uses list initialization and nested loops.
    - Leverages basic arithmetic and array manipulation for efficiency and simplicity.

BRIEF DESCRIPTION:

The program calculates the partition function for any n by systematically building up the count using results for smaller integers. This approach allows accurate and fast computation of partition numbers for a wide range of inputs, providing direct insight into a classic combinatorial function.

RESULTS ACHIEVED:

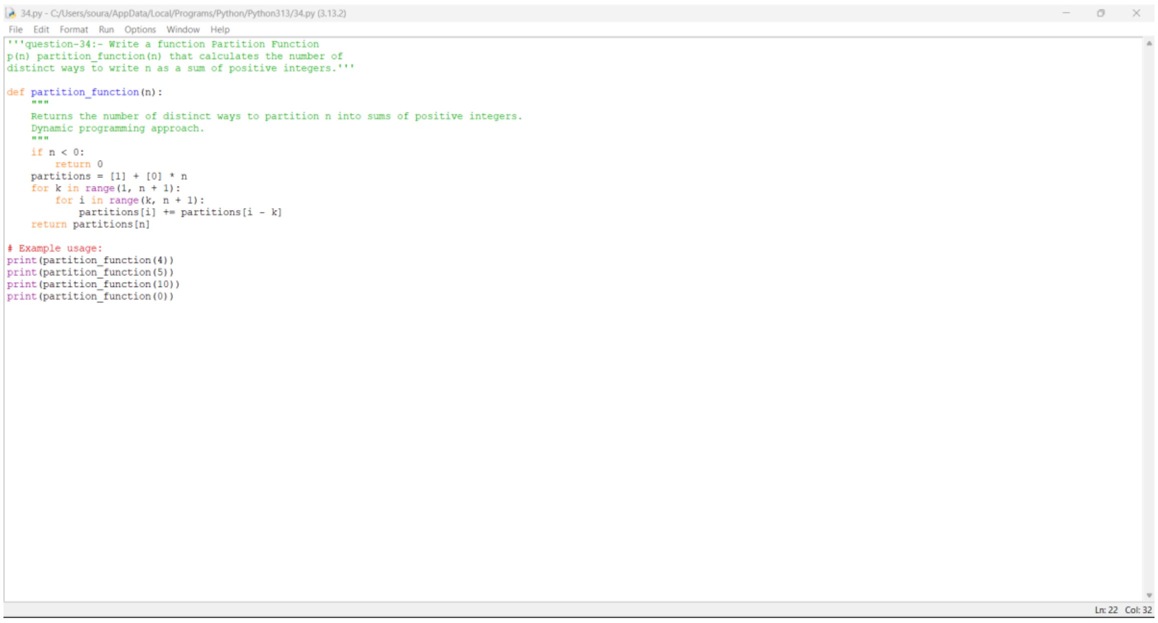
- Returns correct partition numbers for various n.
    - Example: partition_function(4) returns 5; partition_function(10) returns 42.
- The function works efficiently for moderate-sized n and demonstrates the power of dynamic programming.

DIFFICULTY FACED BY STUDENT:

- Understanding partitioning as equivalence classes—why summands' order does not matter.
- Implementing cumulative updates without double-counting partitions.
- Recognizing the combinatorial structure in the algorithm and debugging logic for dynamic programming.
- Managing memory usage for large n, where arrays can become large.
- Grasping the link between recursive definitions and iterative algorithm construction.

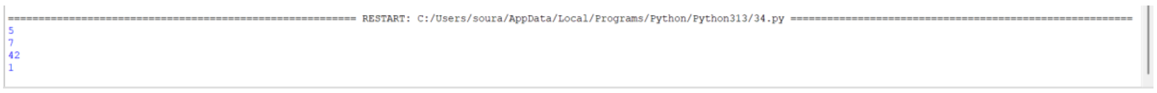
CONCLUSION:

This program deepens comprehension of partition functions and combinatorial mathematics while strengthening skills in dynamic programming. It demonstrates efficient algorithm design and showcases how classic math concepts can be translated into powerful code for number theory and enumeration problems.



34.py - C:/Users/soura/AppData/Local/Programs/Python/Python313/34.py (3.13.2)
File Edit Format Run Options Window Help
Ln: 22 Col: 32

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

def partition_function(n):
    """
    Returns the number of distinct ways to partition n into sums of positive integers.
    Dynamic programming approach.
    """
    if n < 0:
        return 0
    partitions = [1] + [0] * n
    for k in range(1, n + 1):
        for i in range(k, n + 1):
            partitions[i] += partitions[i - k]
    return partitions[n]

# Example usage:
print(partition_function(4))
print(partition_function(5))
print(partition_function(10))
print(partition_function(0))
```



```
========================================================== RESTART: C:/Users/soura/AppData/Local/Programs/Python/Python313/34.py ==========================================================
5
7
42
1
```