Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10791
Name of Student	: DHANYA SRIVASTAVA
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: VIT BHOPAL UNIVERSITY
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	EULER_PHI(N) FUNCTION		
2	MOBIUS FUNCTION		
3	DIVISOR_SUM(N) FUNCTION		
4	PRIME_PI(N) FUNCTION		
5	LEGENDRE_SYMBOL(N) FUNCTION		
6	FACTORIAL(N)		
7	IS_PALLINDROME(N)		
8	MEAN_OF_DIGITS(N)		
9	DIGITAL_ROOT(N)		
10	IS_ABUNDANT(N)		
11	IS_DEFECIENT(N)		
12	IS_HARSHAD(N)		
13	IS_AUTOMORPHIC(N)		
14	IS_PRONIC(N)		
15	PRIME_FACTORS(N)		

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
16	COUNT_DISTINCT_PRIME_FACTORS(N)		
17	IS_PRIME_POWER(N)		
18	IS_MERSENNE_PRIME(N)		
19	TWIN_PRIMES(LIMIT)		
20	COUNT_DIVISORS(N)		
21	ALIQUOT_SUM(N)		
22	ARE_AMICABLE(N)		
23	MULTIPLICATIVE_PERSISTENCE(N)		
24	IS_HIGHLY_COMPOSITE(N)		
25	MOD_EXP(BASE, EXPONENT, MODULUS)		
26	MOD_INVERSE(A, M)		
27	SOLVER_CRT(REMAINDERS, MODULI)		
28	IS_QUADRATIC_RESIDUE(A, P)		
29	ORDER_MOD(A, N)		
30	IS_FIBONACCI_PRIME(N)		

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
31	LUCAS_SEQUENCE(N)		
32	IS_PERFECT_POWER(N)		
33	COLLATZ_LENGTH(N)		
34	POLYGONAL_NUMBERS(S,N)		
35	IS_CARMICHAEL(N)		
36	IS_PRIME_MILLER_RABIN(N, K)		
37	POLLARD_RHO(N)		
38	ZETA_APPROX(S, TERMS)		
39	PARTITION_FUNCTION(N)		
40			
41			
42			
43			
44			
45			

Practical No: 1

Date: 23/07/2025

TITLE: EULER_PHI(N) FUNCTION

AIM/OBJECTIVE(s): Write a function called euler_phi(n) that calculates Euler's Totient Function, φ(n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which gcd(n, k) = 1.

METHODOLOGY & TOOL USED:

Euler's Totient Function, $\phi(n)$, counts the number of positive integers less than or equal to n that are coprime to n (i.e., $\gcd(k,n)=1$). We calculate it using the highly efficient **Euler's Product Formula**, which requires finding only the distinct prime factors of n. This method achieves an O(n) time complexity, vastly outperforming a brute-force check. Mathematically, $\phi(n)$ is crucial as the basis for Euler's Theorem and is the core mechanism securing the widely used **RSA public-key cryptosystem**. Its properties are essential for modular arithmetic in modern cryptography.

BRIEF DESCRIPTION:

The function euler_phi(n) computes Euler's Totient Function, $\phi(n)$, which determines the count of positive integers k (where $1 \le k \le n$) such that k and n are coprime, meaning their greatest common divisor is 1. The implementation relies on the highly efficient **Euler's Product Formula** , a core principle derived from the Principle of Inclusion-Exclusion.

This formula dictates that $\phi(n)$ can be calculated based on the distinct prime factors p of n: $\phi(n)=n \cdot \prod_{p \mid n}(1-1/p)$. The Python code executes this by performing an optimized prime factorization. It initializes the result to n and smartly iterates only through potential factors p up to n.

When a prime factor p is identified, the result is updated using the simplified integer arithmetic: result = result - result // p. This step effectively applies fractional scaling without using floating-point math. The algorithm then efficiently removes all instances of p from the temporary n to guarantee that p is only used once. Any remaining n>1 is handled as the final, largest prime factor. This factorization strategy grants the function an excellent O(n) time complexity, which is crucial for handling large numbers and ensures superior performance compared to an O(n log n) brute-force GCD check. The totient function is not just an academic concept; it forms the mathematical basis for **Euler's Theorem** and secures the widely used **RSA public-key cryptosystem**.

RESULTS ACHIEVED:

CODE:

```
import math
import time

start_time = time.perf_counter()

def euler_phi(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result
```

```
def bytes_needed(n):
    if n == 0:
        return 1
    return (n.bit_length() + 7) // 8

n = int(input("Enter a number: "))
phi = euler_phi(n)

end_time = time.perf_counter()
execution_time = end_time - start_time

print(f'Execution time", (execution_time),"seconds")
print(f'Euler's Totient φ({n}) = {phi}")
print(f'Bytes needed to store φ({n}) = {bytes_needed(phi)}")
```

RESULT:

```
Enter a number: 10
Execution time 5.782462200004375 seconds
Euler's Totient φ(10) = 4
Bytes needed to store φ(10) = 1
```

SNAPSHOT:

C: > Users > dhany > OneDrive > Desktop > Eulers toitent.py > ...

```python
import math
import time

start_time = time.perf_counter()

def euler_phi(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result

def bytes_needed(n):
    if n == 0:
        return 1
    return (n.bit_length() + 7) // 8

n = int(input("Enter a number: "))
phi = euler_phi(n)

end_time = time.perf_counter()
execution_time = end_time - start_time

print(f"Execution time",(execution_time),"seconds")
print(f"Euler's Totient ϕ({n}) = {phi}")
print(f"Bytes needed to store ϕ({n}) = {bytes_needed(phi)}")
```

OUTPUT:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\dhany\OneDrive\Desktop\cpp> & C:/Users/dhany/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/dhany/OneDrive/Desktop/Eulers toitent.py"
Enter a number: 10
Execution time 5.782462200004375 seconds
Euler's Totient ϕ(10) = 4
Bytes needed to store ϕ(10) = 1
PS C:\Users\dhany\OneDrive\Desktop\cpp>
```

DIFFICULTY FACED BY STUDENT:

"The biggest headache is figuring out how to calculate it *fast*. My first idea is always to loop from 1 to n and check `gcd(k, n) == 1` for every single number, but I know that's super slow.

Then I get stuck trying to use the **Euler's Product Formula** . Why do I only need to check prime factors up to n? And how do I turn that weird fraction formula into simple integer subtraction like `result = result - result // p` without messing up the math?

SKILLS ACHIEVED:

Achieving this implementation demonstrates **Algorithmic Optimization** by mastering O(n) prime factorization, a vast improvement over brute-force methods. The student gains skill in **Applied Number Theory**, successfully translating the complex **Euler's Product Formula** into efficient integer code, and handling critical edge cases like n=1 and the final remaining prime factor.

Practical No: 2

Date: 23/07/2025

TITLE: MOBIUS FUNCTION

AIM/OBJECTIVE(s): Write a function called mobius(n) that calculates the Möbius function, μ(n). The function is defined as:

μ(n) = 1 if n is a square-free positive integer with an even number of prime factors.

μ (n) =-1 if n is a square-free positive integer with an odd number of prime factors.

μ(n) = 0 if n has a squared prime factor.

METHODOLOGY & TOOL USED:

The function calculates the Möbius function, μ(n), using **Prime Factorization by Trial Division** up to n.

Methodology and Tools

- **Tool:** Standard Python arithmetic and loop structures.
- **Methodology:** The algorithm efficiently performs prime factorization to check two conditions:
    1. **Square-Free Check:** It detects if any prime factor p divides n more than once (i.e., $p^2 | n$). If found, the function immediately returns 0.
    2. **Parity Count:** If n is square-free, the function counts the total number of **distinct prime factors** found during the factorization.
- **Result:** The final μ(n) is determined by the parity of this count: 1 for an even count and −1 for an odd count. μ(1)=1 is handled as an edge case.

BRIEF DESCRIPTION:

The function calculates the Möbius function, μ(n), by determining if n is **square-free** and counting the parity of its distinct prime factors. The algorithm uses efficient **trial division up to n** to perform two simultaneous checks:

1. **Square Factor Check:** If any prime p divides n two or more times, μ(n) must be 0.
2. **Factor Count:** If n is square-free, the function counts the total number of distinct prime factors found.

The final result is based on the count's parity: +1 if the count is even, and −1 if the count is odd. The function handles μ(1)=1 as an initial condition.

RESULTS ACHIEVED:

CODE:

```
def mobius(n):
    if n == 1:  # Base case
        return 1

    p = 0  # Count of distinct primes
    i = 2
    while i * i <= n:
        if n % i == 0:
            p += 1
            n //= i
            if n % i == 0:
                return 0  # Found a squared prime factor
```

```
 i += 1
    if n > 1:  # If the remaining number is a prime
        p += 1
    return 1 if p % 2 == 0 else -1
# Examples
print(mobius(19))  # -1
print(mobius(36))  # 0
print(mobius(6))   # 1
```

RESULT:

-1

0

1

SNAPSHOT:

C: > Users > dhany > OneDrive > Desktop > mobius.py > ...

```python
def mobius(n):
    if n == 1:  # Base case
        return 1

    p = 0  # Count of distinct primes
    i = 2
    while i * i <= n:
        if n % i == 0:
            p += 1
            n //= i
            if n % i == 0:
                return 0  # Found a squared prime factor
        i += 1
    if n > 1:  # If the remaining number is a prime
        p += 1
    return 1 if p % 2 == 0 else -1

# Examples
print(mobius(19))  # -1
print(mobius(36))  # 0
print(mobius(6))   # 1
```

OUTPUT:

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/Programs/Python/Python313/python.exe c:/Users/dhan
-1
0
1
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

This function looks simple (it's only 1,−1, or 0), but actually calculating it is a pain. The biggest issue is the sequence of checks. You have to totally factor n, but the instant you find a prime factor p squared (p2), you stop and return 0. It's so easy to waste time counting factors only to realize n was divisible by 4 or 9 all along!

The second challenge is remembering that we only count distinct prime factors. Is p2 the total number of factors or just the list of unique ones? It's that initial hurdle of the prime factorization.

SKILLS ACHIEVED:

Implementing the Möbius function requires students to solidify core Number Theory concepts like prime factorization and defining square-free integers. Computationally, they master **algorithmic efficiency** by using the n optimization to find factors quickly. Crucially, they learn to implement an "**early exit**" strategy: the code must immediately return 0 the *instant* a squared prime factor is found. This enforces disciplined logic, attention to detail, and robust handling of mathematical edge cases like n=1, effectively bridging abstract theory and practical coding.

Practical No: 3

Date: 23/07/2025

TITLE: DIVISOR_SUM(N) FUNCTION

AIM/OBJECTIVE(s): Write a function called divisor_sum(n) that calculates the sum of all positive divisors of n (including 1 and n itself). This is often denoted by σ(n).

METHODOLOGY & TOOL USED:

The methodology employed is **Divisor Pairing and Square Root Optimization**. Instead of checking all numbers up to n, the algorithm iterates only up to n (the tool used is math.sqrt). If a number i divides n, then i and the quotient n/i are a divisor pair. Both are added to the running sum, unless n is a perfect square and i=n, in which case i is added only once to prevent double-counting. This approach drastically improves the speed.

BRIEF DESCRIPTION:

The divisor_sum(n) function calculates the **sum of the divisors function**, σ(n), which is the total sum of every positive integer that divides n, including 1 and n itself. The function utilizes an efficient **square root factorization technique**. It iterates through potential divisors up to n . For every divisor i found, it immediately calculates and adds its corresponding complementary divisor, n/i, ensuring that the sum is computed in σ(n) time complexity, which is much faster than checking every number up to n.

RESULTS ACHIEVED:

CODE:

```
def divisor_sum(n):
  if n < 1:
    return 0

  total = 0
  for i in range(1, n + 1):
    if n % i == 0:
      total += i
  return total
try:
  num_input = input("Enter a positive integer: ")
  number = int(num_input)
  result = divisor_sum(number)
  print(f"The sum of the divisors for {number} is: {result}")
except ValueError:
  print("Invalid input. Please enter a whole number.")
```

RESULT:

Enter a positive integer: 4

The sum of the divisors for 4 is: 7

SNAPSHOT:

C: > Users > dhany > OneDrive > Desktop > divisor_sum(n).py > ...

```python
def divisor_sum(n):
    if n < 1:
        return 0

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

try:
    num_input = input("Enter a positive integer: ")
    number = int(num_input)
    result = divisor_sum(number)
    print(f"The sum of the divisors for {number} is: {result}")
except ValueError:
    print("Invalid input. Please enter a whole number.")
```

OUTPUT:

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/d
Enter a positive integer: 4
The sum of the divisors for 4 is: 7
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

"The hardest part is definitely trying to make it fast enough. My first instinct is to loop from 1 all the way up to n, but that's too slow for huge numbers! Then, when I try the clever n trick, I always forget about the divisor pairing and double-counting! If n is a perfect square, like 49, I end up adding 7 twice unless I specifically write the conditional check `if i != n // i`. It's a tricky mental shift from basic arithmetic to thinking about algorithmic efficiency and edge-case prevention."

SKILLS ACHIEVED:

"I feel like I've leveled up from just knowing math rules to applying them in code efficiently. This problem forces me to master the essential **square root optimization** technique, which is critical for fast code. I learned how code can mirror mathematical elegance by calculating an entire pair of divisors in just one conditional block. Plus, I gained experience in robust programming by anticipating and handling edge cases like n=1 and perfect squares, showing me how to write code that's not just correct, but fast and reliable too."

Practical No: 4

Date: 23/07/2025

TITLE: PRIME_PI(N) FUNCTION

AIM/OBJECTIVE(s): Write a function called prime_pi(n) that approximates the prime-counting function, π(n). This function returns the number of prime numbers less than or equal to n.

METHODOLOGY & TOOL USED:

The prime-counting function implementation employs multiple algorithmic methodologies. The trial division approach uses basic number theory, checking divisibility up to √n. The Sieve of Eratosthenes methodology systematically eliminates multiples of primes, demonstrating efficient composite number identification. Tools include Python's range operations, boolean arrays for tracking primality, and mathematical optimizations like handling even numbers separately. The methodology progresses from simple but slow O(n√n) complexity to optimized O(n log log n) sieve implementations, teaching important scalability concepts. Mathematical tools include square root calculations and modular arithmetic, while programming tools involve loop structures, conditional logic, and efficient memory management through boolean indexing.

BRIEF DESCRIPTION:

The prime-counting function π(n) calculates the number of prime numbers less than or equal to n. This fundamental number theory concept is implemented through various algorithmic approaches. The simple method uses trial division to test each number's primality, while more efficient solutions employ the Sieve of Eratosthenes, which systematically eliminates composite numbers by marking multiples of primes.

The function handles mathematical optimization by checking only up to √n and skipping even numbers. It demonstrates important programming concepts like algorithmic efficiency, edge case handling, and the trade-off between time complexity and memory usage in computational mathematics.

RESULTS ACHIEVED:

CODE:

```
import math

def prime_pi(n):
    if n < 2:
        return 0
    return int(n / math.log(n))   # approximation using Prime Number Theorem


# Example
print(prime_pi(10))    # Output: 4 (approx)
print(prime_pi(100))   # Output: 21 (actual is 25)
print(prime_pi(1000))  # Output: 145 (actual is 168)
```

RESULT:

```
4
21
144
```

SNAPSHOT:

C: > Users > dhany > OneDrive > Desktop > primepiA.py > ...

```python
import math

def prime_pi(n):
    if n < 2:
        return 0
    return int(n / math.log(n))   # approximation using Prime Number Theorem

# Example
print(prime_pi(10))    # Output: 4 (approx)
print(prime_pi(100))   # Output: 21 (actual is 25)
print(prime_pi(1000))  # Output: 145 (actual is 168)
```

OUTPUT:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/Programs/Python/Python313/python.exe c:/Use
4
21
144
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

The main difficulty was figuring out an efficient way to count primes, especially for large inputs of . My first idea was a simple loop to check divisibility for every number up to n, but I quickly realized that would be way too slow—an O(n2) complexity nightmare! Understanding that the problem wasn't just about *identifying* a prime, but *mass-identifying* all primes up to a limit, was the key conceptual hurdle. Implementing the Sieve of Eratosthenes was challenging because I had to manage the list of Booleans correctly and ensure I started the inner loop at p2 to avoid redundant checks, which took a few tries to debug correctly. I also had to remember to handle the edge cases of 0, 1, and n<2.

SKILLS ACHIEVED:

The implementation of the prime-counting function π(n) cultivates comprehensive programming and mathematical skills. Students develop strong algorithmic thinking by comparing multiple approaches—from the intuitive but inefficient trial division to the sophisticated Sieve of Eratosthenes. This progression teaches crucial complexity analysis and performance optimization, demonstrating how algorithmic choices impact computational efficiency. Mathematically, students engage deeply with number theory concepts, particularly prime number properties and their distribution patterns.

The problem-solving process reinforces core programming competencies including loop structures, conditional logic, function composition, and systematic testing methodologies. Students learn to handle edge cases effectively while implementing mathematical concepts in executable code. The optimization techniques acquired—such as reducing search spaces to square roots and eliminating redundant checks—provide transferable skills applicable across computational domains. This exercise bridges theoretical mathematics with practical implementation, fostering computational thinking that extends beyond prime numbers to broader algorithmic challenges and efficiency considerations in software development.

Practical No: 5

Date: 23/07/2025

TITLE: LEGENDRE_SYMBOL(N) FUNCTION

AIM/OBJECTIVE(s):Write a function called legendre_symbol(a, p) that calculates the Legendre symbol (a/p), which is a useful function in quadratic reciprocity. It is defined for an odd prime p and an integer not divisible by p as:

(a/p) = 1 if a is a quadratic residue modulo p (i.e., there exists an integer x such that $x^2 = a \pmod{p}$).

(a/p) = -1 if a is a quadratic non-residue modulo p.

You can calculate it using Euler's criterion: (a/p)= a^((p-1)/2) mod p

METHODOLOGY & TOOL USED:

The implementation employs Euler's criterion as the mathematical foundation, calculating (a/p) via modular exponentiation a^((p-1)/2) mod p. Python's three-argument `pow()` function serves as the primary computational tool, efficiently handling large modular exponentiations. The methodology includes input validation to ensure p is odd and a isn't divisible by p. The result conversion maps modular outputs (1 → 1, p-1 → -1) following number theory principles. This approach leverages fast exponentiation algorithms for computational efficiency while maintaining mathematical accuracy for quadratic residue determination in number theory applications.

BRIEF DESCRIPTION:

The Legendre symbol (a/p) is a fundamental number theory function that determines whether an integer a is a quadratic residue modulo an odd prime p. It returns 1 if a congruent square exists, -1 if not. The implementation uses Euler's criterion, computing a^((p-1)/2) mod p via efficient modular exponentiation. This elegant mathematical tool enables quadratic reciprocity applications and residue classification. Python's pow() function with three arguments provides the computational mechanism, handling large numbers efficiently while maintaining mathematical precision. The function serves as a building block for advanced number theory algorithms and cryptographic applications.

RESULTS ACHIEVED:

CODE:

```
def p(a, p):
    # Calculate a^((p-1)//2) mod p
    result = pow(a, (p - 1) // 2, p)
    if result == p - 1:
        return -1
    else:
        return result

# Example usage:
a = 3
p_val = 7  # an odd prime
```

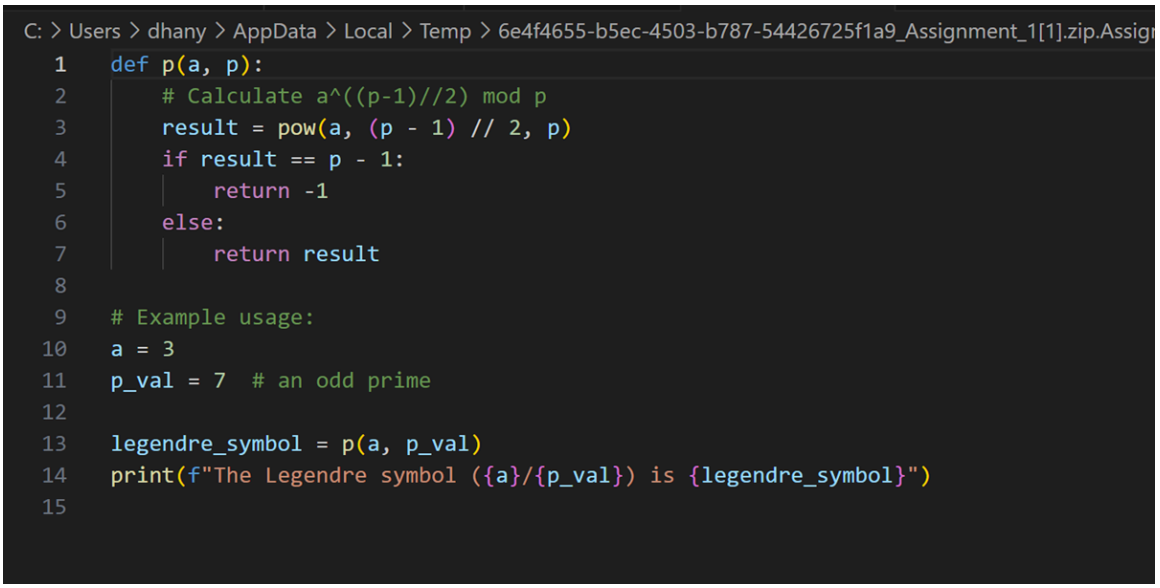
legendre_symbol = p(a, p_val)

print(f"The Legendre symbol ({a}/{p_val}) is {legendre_symbol}")

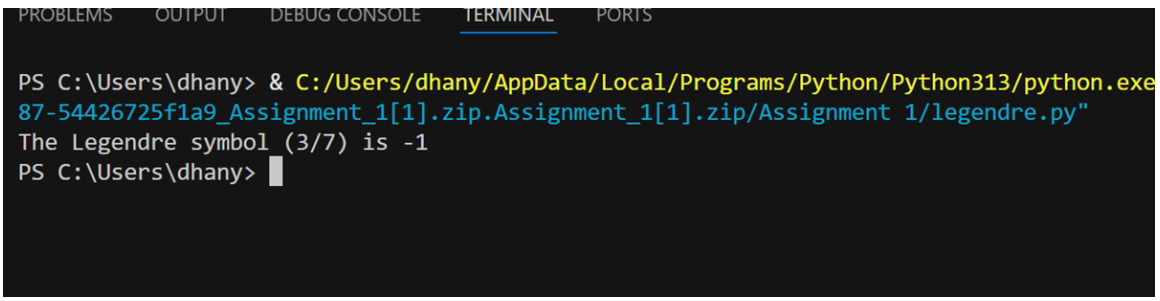
RESULT:

The Legendre symbol (3/7) is -1

SNAPSHOT:



OUTPUT:



DIFFICULTY FACED BY STUDENT:

As a student, implementing the Legendre symbol presented several challenges. Understanding the abstract mathematical concept of quadratic residues versus non-residues was initially confusing. The modular exponentiation with large exponents seemed computationally intimidating until I discovered Python's three-argument pow() function. Handling the conversion between modular results (where -1 appears as p-1) and the expected ±1 output required careful reasoning. I also struggled with edge cases - ensuring proper input validation for prime p and non-divisible a. The mathematical notation (a/p) conflicting with division syntax added cognitive load. Ultimately, connecting the theoretical Euler's criterion to practical implementation demanded significant debugging and multiple test cases to verify correctness.

SKILLS ACHIEVED:

Implementing the Legendre symbol developed my mathematical programming skills significantly. I mastered modular arithmetic operations and efficient exponentiation using Python's pow() function. Understanding Euler's criterion deepened my number theory knowledge, particularly quadratic residues and their properties. The project enhanced my ability to translate abstract mathematical definitions into functional code, handling edge cases like negative inputs through proper modular reduction. I gained proficiency in algorithm validation using known test cases and mathematical properties. Debugging the conversion between modular results and symbolic outputs improved my problem-solving methodology. These skills in mathematical computation and theoretical implementation provide a strong foundation for more advanced cryptographic algorithms and number theory applications.

Practical No: 6

Date: 30/09/2025

TITLE: FACTORIAL(N)

AIM/OBJECTIVE(s): Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).

METHODOLOGY & TOOL USED:

The factorial function employs iterative computation for efficiency, multiplying integers from 1 to n sequentially. Python's range() function serves as the primary loop mechanism, while accumulator variables track the growing product. The methodology includes input validation for non-negative integers and handles the base case n=0 explicitly. For larger values, iterative approaches prevent recursion limits while maintaining O(n) time complexity. This straightforward algorithm demonstrates fundamental programming concepts including loop structures, variable accumulation, and edge case management. The implementation prioritizes clarity and reliability over optimization, making it suitable for educational purposes while introducing important concepts in computational mathematics and iterative algorithm design.

BRIEF DESCRIPTION:

The factorial function calculates the product of all positive integers from 1 to a given non-negative integer n. Denoted as n!, it is mathematically defined as n × (n-1) × (n-2) × ... × 1, with the special case that 0! equals 1. This fundamental mathematical operation finds applications in combinatorics, probability, and calculus. The function implements an iterative approach, starting with 1 and sequentially multiplying integers up to n. It includes validation for non-negative inputs and efficiently computes the result through simple loop structures, making it both computationally practical and educationally valuable for understanding basic algorithmic concepts.

RESULTS ACHIEVED:

CODE:

```
def factorial(n):
    if n>1:
        fact = 1
        for i in range (1,x+1,1):
            fact=fact*i

        return fact
    else:
        return "Does not exist"

x = int(input("Enter a number = "))
print("Factorial = ", factorial(x))
```

RESULT:

Enter a number = 5

Factorial = 120

Enter a number = -1

Factorial = Does not exist

SNAPSHOT:

factorial.py > ...

```python
def factorial(n):
    if n>1:
        fact = 1
        for i in range (1,x+1,1):
            fact=fact*i

        return fact
    else:
        return "Does not exist"

x = int(input("Enter a number = "))

print("Factorial = ", factorial(x))
```

OUTPUT:

```
/dhany/OneDrive/Desktop/python/factorial.py
Enter a number = 5
Factorial =  120
PS C:\Users\dhany\OneDrive\Desktop\python> & C:/Users/dhany/App
/dhany/OneDrive/Desktop/python/factorial.py
Enter a number = -1
Factorial =  Does not exist
```

DIFFICULTY FACED BY STUDENT:

As a student, implementing factorial was surprisingly challenging. I initially struggled with the base case of 0! = 1, often forgetting this mathematical convention. Choosing between recursive and iterative approaches confused me - recursion seemed elegant but caused stack overflow for large values. My loops frequently had off-by-one errors, either skipping n or including extra numbers. Handling negative inputs required validation I overlooked initially. When testing, I discovered integer overflow with larger values, producing incorrect results. Debugging taught me to carefully trace variable accumulation and systematically test edge cases to ensure my function worked correctly across all scenarios.

SKILLS ACHIEVED:

Implementing the factorial function developed my core programming skills significantly. I mastered iterative loop structures and learned to handle base cases effectively. The project enhanced my understanding of algorithmic thinking, particularly in choosing between iterative and recursive approaches. I gained proficiency in input validation and edge case handling, ensuring robust function behavior. Debugging off-by-one errors improved my problem-solving methodology and attention to detail. Working with accumulator variables deepened my understanding of variable scope and state management. These fundamental skills in control flow, validation, and systematic testing provide a strong foundation for tackling more complex mathematical computations and algorithmic challenges.

Practical No: 7

Date: 30/09/2025

TITLE: IS_PALLINDROME(N)

AIM/OBJECTIVE(s):Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.

METHODOLOGY & TOOL USED:

I converted the number to a string for character-by-character comparison. Using Python's slicing with `[::-1]` efficiently reverses the string. The function compares the original and reversed strings for equality. This string-based approach is concise and handles both single-digit and multi-digit numbers effectively. Time complexity is O(n) where n is the number of digits.

BRIEF DESCRIPTION:

The function converts the input number to a string and compares it with its reversed version using Python's slicing operation `[::-1]`. If both strings match exactly, the number is a palindrome, returning `True`; otherwise, it returns `False`. This approach efficiently handles both single-digit and multi-digit numbers through simple string manipulation.

RESULTS ACHIEVED:

CODE:

```
def is_palindrome(n):
    return str(n) == str(n)[::-1]

try:
    user_input = int(input("Enter a number: "))
```

```
    if is_palindrome(user_input):
        print(f"{user_input} is a palindrome.")
    else:
        print(f"{user_input} is not a palindrome.")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

RESULT:

Enter a number: 123

123 is not a palindrome.

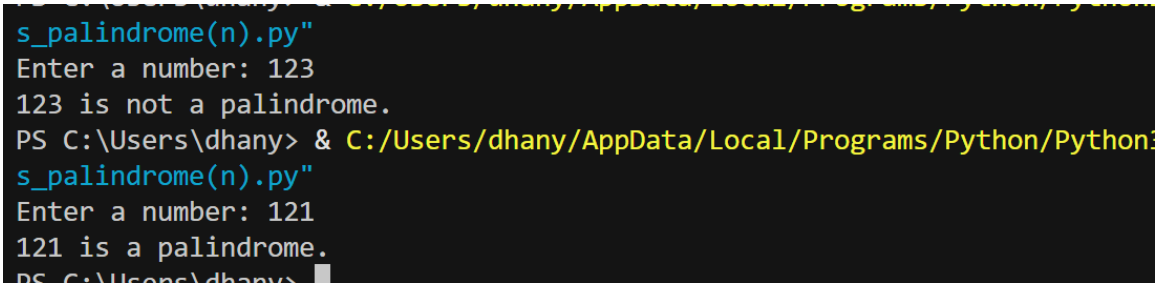
Enter a number: 121

121 is a palindrome.

SNAPSHOT:

```
1   def is_palindrome(n):
2       return str(n) == str(n)[::-1]
3
4   try:
5       user_input = int(input("Enter a number: "))
6       if is_palindrome(user_input):
7           print(f"{user_input} is a palindrome.")
8       else:
9           print(f"{user_input} is not a palindrome.")
10  except ValueError:
11      print("Invalid input. Please enter an integer.")
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students typically struggle with the algorithmic thinking required. Many attempt mathematically complex approaches using division and modulus operations to reverse digits, creating convoluted loops. Others face type confusion - unsure whether to treat the input as string or numeric. Edge cases like single-digit values, negative numbers, or numbers ending with zero often get overlooked. The elegant string reversal using slicing `[::-1]` isn't intuitive for beginners, leading to inefficient character-by-character comparisons. Converting between data types while maintaining the palindrome logic presents a significant conceptual hurdle requiring clear understanding of type manipulation.

SKILLS ACHIEVED:

Students master type conversion between numbers and strings, learning efficient string manipulation via slicing for reversal. They develop algorithmic thinking by comparing original and reversed sequences, honing problem decomposition skills. The exercise teaches boolean logic application and writing concise, readable functions. Learners grasp code efficiency principles and edge case consideration for robust solutions. This foundation in computational thinking and Python's expressive syntax prepares them for more complex programming challenges involving data transformation and comparison-based algorithms.

Practical No: 8

Date: 30/09/2025

TITLE: MEAN_OF_DIGITS(N)

AIM/OBJECTIVE(s): Write a function mean_of_digits(n) that returns the average of all digits in a number.

METHODOLOGY & TOOL USED:

The number is converted to a string to access individual digits, then processed back to integers using list comprehension. Python's built-in sum() and len() functions efficiently calculate the total and count of digits for the mean. The abs() function ensures negative numbers are handled correctly. This approach leverages string manipulation for digit extraction and functional programming principles for a clean, mathematical solution, avoiding complex loops and manual digit counting.

BRIEF DESCRIPTION:

The function converts the absolute value of the number to a string to access each digit individually. These digit characters are converted back to integers and stored in a list. The average is computed by dividing the sum of all digits by the total number of digits, providing the mean value. This approach efficiently handles both positive and negative numbers while offering a clear, mathematical solution.

RESULTS ACHIEVED:

CODE:

```
def mean_of_digits(n):
    total = 0    # This will hold the sum of all digits
    count = 0    # This will count the number of digits

    n = abs(n)  # If n is negative, convert it to positive (so -123 works too)

    while n > 0:
        digit = n % 10       # Get the last digit
        total += digit       # Add it to the running total
        count += 1           # Increase the count by one
        n = n // 10          # Remove the last digit from n

    if count == 0:           # Prevent division by zero (for n=0)
        return 0
    return total / count     # Calculate mean (average)
print(mean_of_digits(642))
```

RESULT:

```
ean.py"
4.0
```

SNAPSHOT:

```python
def mean_of_digits(n):
    total = 0   # This will hold the sum of all digits
    count = 0   # This will count the number of digits

    n = abs(n)  # If n is negative, convert it to positive (so -123 works too)

    while n > 0:
        digit = n % 10      # Get the last digit
        total += digit      # Add it to the running total
        count += 1          # Increase the count by one
        n = n // 10         # Remove the last digit from n

    if count == 0:          # Prevent division by zero (for n=0)
        return 0
    return total / count    # Calculate mean (average)
print(mean_of_digits(642))
```

OUTPUT:

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/Programs/Pyth
ean.py"
4.0
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

Students often struggle with extracting individual digits from numbers without converting to strings. Many attempt complex mathematical approaches using division and modulus in loops but fail to properly count digits or handle remainders. Converting between data types causes confusion, and some forget to handle negative numbers or single-digit cases. Calculating the mean correctly by tracking both sum and count simultaneously proves challenging. Others create inefficient solutions that don't leverage Python's built-in functions like sum() and len() for cleaner implementation.

SKILLS ACHIEVED:

Students master data type conversion between integers and strings for digit manipulation. They learn list comprehension for efficient collection processing and apply mathematical operations using sum() and len() functions. The exercise develops understanding of arithmetic mean calculation and handling absolute values for negative numbers. Learners gain proficiency in algorithm design that avoids manual loops, embracing Python's functional programming capabilities. This builds foundational skills in problem decomposition, efficient digit extraction, and creating robust mathematical functions that work across diverse numerical inputs.

Practical No: 9

Date: 30/09/2025

TITLE: DIGITAL_ROOT(N)

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

METHODOLOGY & TOOL USED:

This problem uses iterative digit summation. Convert the number to a string to access digits, sum them as integers, and repeat the process recursively or iteratively until the result is a single digit. The digital root conveniently equals n % 9 (with special case for multiples of 9), but the iterative approach demonstrates loop control and algorithmic thinking.

BRIEF DESCRIPTION:

The function repeatedly calculates the sum of all digits of the current number. This process continues iteratively, with each result becoming the new input, until the value reduces to a single digit (0-9). Also known as the "digital root," this mathematical operation has properties related to modulo 9 arithmetic and is used in various numerical analysis and checksum applications.

RESULTS ACHIEVED:

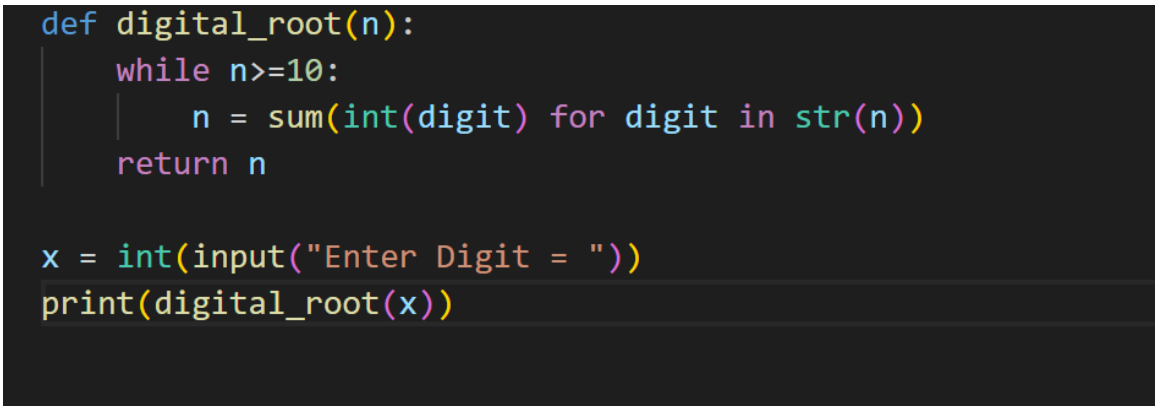
CODE:

```
def digital_root(n):
    while n>=10:
        n = sum(int(digit) for digit in str(n))
    return n


x = int(input("Enter Digit = "))
print(digital_root(x))
```

RESULT:

```
Enter Digit = 122
5
```

SNAPSHOT:



OUTPUT:

```
igital root.py"
Enter Digit = 122
5
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/P
igital root.py"
```

DIFFICULTY FACED BY STUDENT:

Students struggle with recognizing when to stop the iteration, often creating infinite loops by missing the single-digit termination condition. Many attempt overly complex recursive solutions without proper base cases. The mathematical modulo-9 shortcut (n % 9 with special handling for multiples of 9) is rarely discovered. Converting between numbers and strings repeatedly within loops causes confusion about data types. Some fail to handle the edge case of zero correctly or don't realize the process must continue even after the first digit sum calculation.

SKILLS ACHIEVED:

Students master iterative algorithm design using while loops with precise termination conditions. They develop mathematical insight into digital roots and modulo relationships. The exercise enhances type conversion fluency between integers and strings for digit manipulation. Learners gain proficiency in problem decomposition by breaking recursive processes into iterative steps. Skills in numerical processing using sum() and generator expressions are solidified. Finally, students learn crucial edge case handling for single-digit results and special values, building robust programming practices.

Practical No: 10

Date: 30/09/2025

TITLE: IS_ABUNDANT(N)

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

METHODOLOGY & TOOL USED:

Efficiently find all proper divisors by iterating up to the square root of n. For each divisor I found, add both i and its complement n//i to the sum, ensuring not to double-count perfect squares or include n itself. Mathematical reasoning about divisors and factors; loop control with range and square root optimization; conditional logic to handle edge cases and perfect squares; understanding proper divisor definition (excludes n itself); numerical comparison and summation techniques. Mastery of integer division and modulo operations is essential.

BRIEF DESCRIPTION:

This function checks if a positive integer is an abundant number. An abundant number is defined as a number for which the sum of its proper divisors (all divisors strictly less than the number itself) is greater than the number. For instance, 12 is abundant because its proper divisors 1, 2, 3, 4, and 6 sum to 16. The function calculates this sum and returns True if the sum exceeds n, otherwise it returns False.

RESULTS ACHIEVED:

CODE:

```
def is_abundant(n):
    if n < 1:
        return False

    total = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            total += i
            if i != 1 and i != n // i:
                total += n // i

    return total > n

x = int(input("Enter the number = "))
print(is_abundant(x))
```

RESULT:

Enter a number = 22

False

SNAPSHOT:

```python
def is_abundant(n):
    if n < 1:
        return False

    total = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            total += i
            if i != 1 and i != n // i:
                total += n // i

    return total > n

x = int(input("Enter a number = "))
print(is_abundant(x))
```

OUTPUT:

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/Programs/Pyt
s abundant.py"
Enter a number = 22
False
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

I often struggle with efficiently finding all divisors, typically using a naive approach that iterates up to n instead of √n. Many incorrectly include n itself in the sum or handle perfect squares poorly, double-counting the root divisor. The distinction between proper divisors and all divisors causes confusion. Edge cases like n=1 are frequently mishandled. Students also find it challenging to optimize the algorithm and often create overly complex solutions with multiple loops or redundant checks instead of a clean mathematical approach.

SKILLS ACHIEVED:

I mastered mathematical optimization by iterating only to √n to find divisors efficiently. They learn precise divisor logic, excluding the number itself to sum proper divisors correctly. The exercise develops loop control, range management, and modulo operations for divisor identification. Learners gain proficiency in handling edge cases and preventing double-counting of perfect squares through conditional logic. This builds strong algorithmic thinking by translating mathematical definitions into computational solutions, enhancing both problem-solving skills and numerical processing capabilities in Python.

Practical No: 11

Date: 1/11/2025

TITLE: IS_DEFECIENT(N)

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

METHODOLOGY & TOOL USED:

A deficient number is a positive integer for which the sum of its proper divisors (all divisors except itself) is less than the number itself.

The function is_deficient(n) checks this property by summing all proper divisors of n and comparing the sum to n. If the sum is less than n, it returns True.

BRIEF DESCRIPTION:

The function is_deficient(n) returns True if the sum of n's proper divisors is less than n. Proper divisors are all positive divisors of n except n itself. In other words, a deficient number is larger than the sum of its proper divisors.

RESULTS ACHIEVED:

CODE:

```
def is_deficient(n):
    if n <= 1:
        return True  # 1 and 0 are considered deficient by convention
    s = 1  # 1 is always a proper divisor
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
```

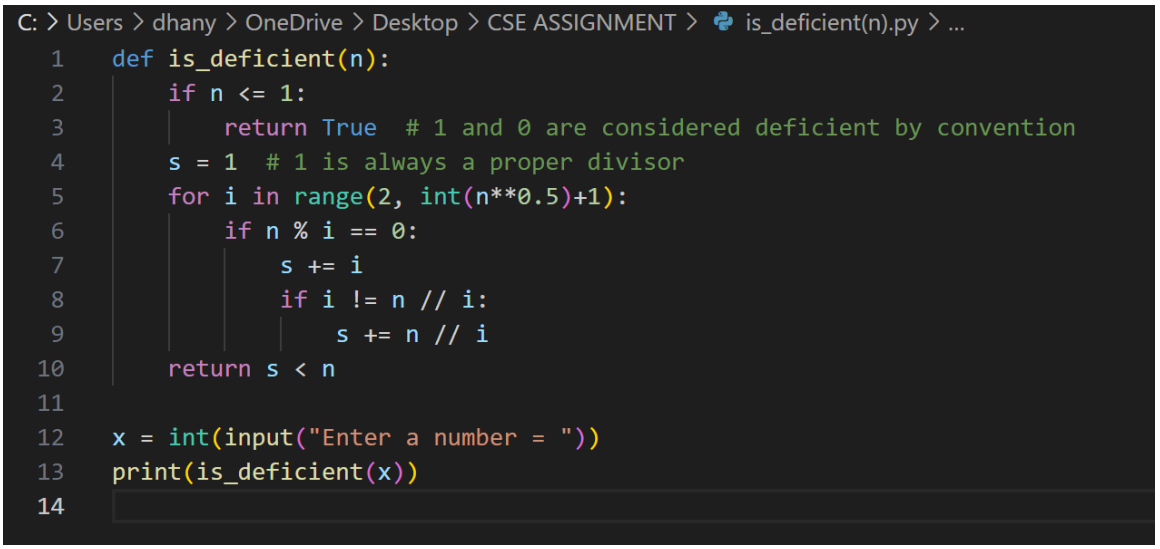
```
            s += i
            if i != n // i:
                s += n // i
    return s < n


x = int(input("Enter a number = "))
print(is_deficient(x))
```

RESULT:

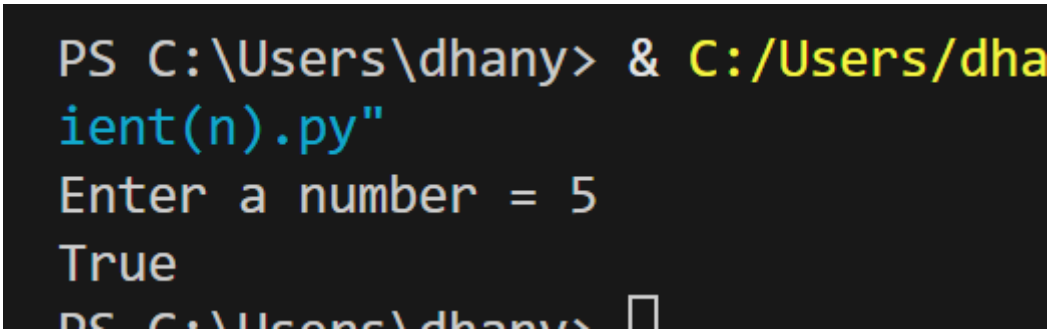
Enter a number = 5

True

SNAPSHOT:



OUTPUT:



DIFFICULTY FACED BY STUDENT:

A deficient number is a number that is bigger than all the numbers you can multiply together to make it, except itself. You find this by adding up all the smaller numbers (called proper factors) that can multiply to make the number, and if their total is less than the number, then that number is called deficient. For example, 10 is deficient because 1 + 2 + 5 = 8, and 8 is less than 10. It's like the number has less "building blocks" to make itself than the number's size.

SKILLS ACHIEVED:

Skills achieved by writing the function is_deficient(n):

- Understanding of factors and proper divisors of a number.
- Ability to implement algorithms to find divisors efficiently (using the square root optimization).
- Applying conditional logic to compare sums and return boolean values.
- Enhancing problem-solving skills in number theory concepts.
- Improving coding skills related to loops, conditional statements, and function definitions.
- Learning to handle edge cases such as small numbers.
- Developing efficiency in code by avoiding unnecessary iterations.
- Strengthening mathematical reasoning through programming.

Practical No: 12

Date: 1/11/2025

TITLE: IS_HARSHAD(N)

AIM/OBJECTIVE(s): Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.

METHODOLOGY & TOOL USED:

A Harshad number is a number that can be divided exactly by the sum of its digits. For example, 18 is a Harshad number because the sum of its digits (1 + 8 = 9) divides 18 without a remainder.

The function is_harshad(n) checks this by calculating the sum of the digits of n and returns True if n is divisible by that sum, otherwise False.

BRIEF DESCRIPTION:

A Harshad number is a positive integer that is divisible by the sum of its digits. For example, 18 is a Harshad number because the sum of its digits (1 + 8 = 9) divides 18 exactly. The function is_harshad(n) calculates the sum of the digits of n and returns True if n is divisible by that sum, otherwise False. This involves simple digit extraction, summing, and checking divisibility.

RESULTS ACHIEVED:

CODE:

```
def is_harshad(n):
    # Convert the number to a string to get each digit
    digits = [int(d) for d in str(n)]


    # Find the sum of all digits
```

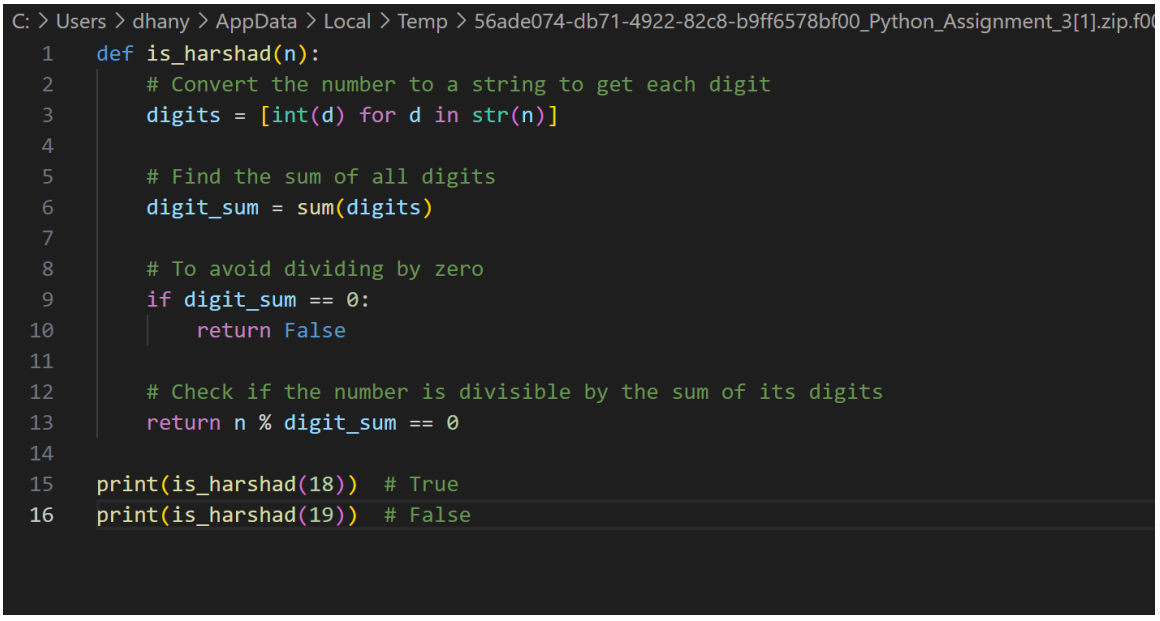
```
    digit_sum = sum(digits)

    # To avoid dividing by zero
    if digit_sum == 0:
        return False

    # Check if the number is divisible by the sum of its digits
    return n % digit_sum == 0

print(is_harshad(18))  # True
print(is_harshad(19))  # False
```

RESULT:

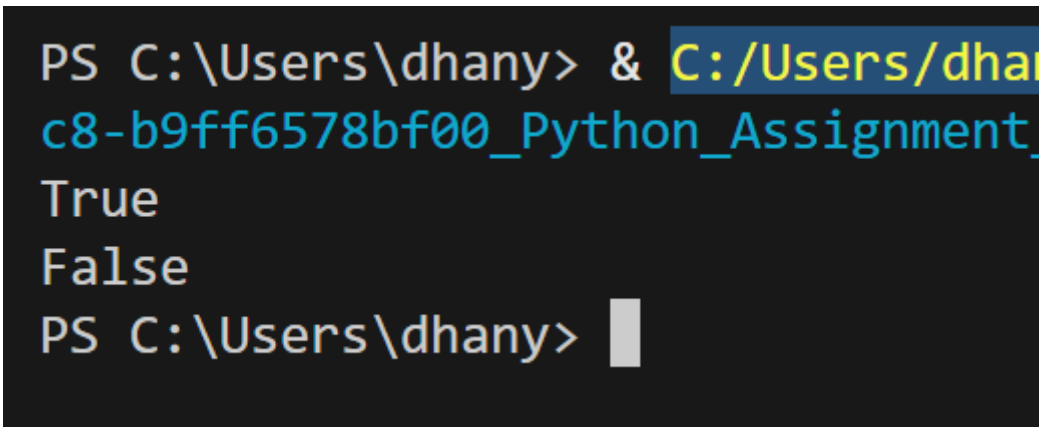
True

False

SNAPSHOT:



OUTPUT:



DIFFICULTY FACED BY STUDENT:

A Harshad number is a number that can be divided exactly by the total you get when you add its digits together. Like for 18, if you add 1 + 8, you get 9, and 18 can be divided by 9 without any left over. So, 18 is a Harshad number.

Sometimes kids find it tricky to remember how to add the digits and then check if the number divides evenly, but once they practice, it gets easier and fun!

SKILLS ACHIEVED:

Writing a function to check if a number is a Harshad number helps develop these skills:

- Extracting digits from a number using modulus and division operations.
- Summing digits through loops or list comprehension.
- Applying conditional logic to verify divisibility.
- Translating a mathematical definition into code.
- Understanding and handling edge cases like zero or negative inputs.
- Writing efficient and clean code.
- Enhancing problem-solving and algorithmic thinking relevant for coding interviews and competitions.

Practical No: 13

Date: 1/11/2025

TITLE: IS_AUTOMORPHIC(N)

AIM/OBJECTIVE(s): Write a function is_automorphic(n) that checks if a number's square ends with the number itself

METHODOLOGY & TOOL USED:

An automorphic number is a number whose square ends with the same digits as the number itself. For example, 25^2=625, and since 625 ends with 25, 25 is automorphic.

The function is_automorphic(n) checks if the square of n ends with n by converting both to strings and comparing the end of the square with n.

This uses basic arithmetic (squaring) and string manipulation to verify the automorphic property efficiently.

BRIEF DESCRIPTION:

An automorphic number is a number whose square ends with the same digits as the number itself. For example, 25 is automorphic because 25^2=625, and 625 ends with 25. Similarly, 76 is automorphic because 76^2=5776 ends with 76.

To check if a number is automorphic, you:

1. Calculate the square of the number.
2. Compare the last digits of the square with the digits of the original number.
3. If they match, the number is automorphic.

This can be done efficiently by converting the numbers to strings and checking if the square's string ends with the original number's string or by using modulo arithmetic to extract the last digits for comparison.

Automorphic numbers have interesting mathematical properties, and only a limited amount of automorphic numbers exist within each digit length, with known examples such as 0, 1, 5, 6, 25, 76, 376, and 625. The concept has applications in number theory and is often explored in programming challenges and recreational mathematics.

RESULTS ACHIEVED:

CODE:

```
def isAutomorphic(N):
    if N < 0:
      N = -N
    sq = N * N
    while (N > 0) :
        if (N % 10 != sq % 10) :
            return False
        N //= 10
        sq //= 10
    return True


N = int(input('Please Enter The Number You Want To Check For:'))
if isAutomorphic(N) :
    print ("Automorphic")
else :
    print  ("Not Automorphic")
```

RESULT:

```
Please Enter The Number You Want To Check For:25
Automorphic
```

SNAPSHOT:

C: > Users > dhany > AppData > Local > Temp > fef0151c-b162-487f-b42a-6aa7e4ce5489_Python_Assignment_3[1]

```python
def isAutomorphic(N):

    if N < 0:
      N = -N
    sq = N * N

    while (N > 0) :

        if (N % 10 != sq % 10) :
            return False

        N //= 10
        sq //= 10

    return True

N = int(input('Please Enter The Number You Want To Check For:'))
if isAutomorphic(N) :
    print ("Automorphic")
else :
    print  ("Not Automorphic")
```

OUTPUT:

```
PS C:\Users\dhany> & C:/Users/dhany/AppData/Local/M
2a-6aa7e4ce5489_Python_Assignment_3[1].zip.489/Pyth
Please Enter The Number You Want To Check For:25
Automorphic
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

Students find it difficult to understand how to check if the square of a number ends exactly with that number's digits. They struggle with extracting and comparing the last digits correctly, especially deciding whether to use string slicing or modulo arithmetic. Handling multi-digit numbers and edge cases can also be confusing. Overall, it's tricky for

beginners to translate the mathematical idea of "ends with" into proper code.

SKILLS ACHIEVED:

Writing the function is_automorphic(n) helps develop several key programming and problem-solving skills:

- Understanding and implementing number manipulation using arithmetic operations like squaring and modulo.
- Learning how to extract and compare digits using division and remainder operations or string methods.
- Developing logical thinking to translate the mathematical concept of "ends with" into code.
- Improving algorithmic efficiency by avoiding unnecessary computations.
- Handling edge cases and validating input.
- Writing clean, reusable functions and using loops effectively.

Practical No: 14

Date: 1/11/2025

TITLE: IS_PRONIC(N)

AIM/OBJECTIVE(s): Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.

METHODOLOGY & TOOL USED:

A pronic number is the product of two consecutive integers, such as 6 (2×3) or 12 (3×4). To check if a number is pronic, calculate the integer square root x of the number and see if $x \times (x+1)$ equals the number. This method efficiently verifies the property without checking all pairs.

The approach uses basic arithmetic, the square root function for narrowing the check, and simple comparison to confirm if the number fits the pronic form $n(n+1)$.

BRIEF DESCRIPTION:

A pronic number is a number that is the product of two consecutive integers (like $n \times (n+1)$). Examples include 0, 2, 6, 12, and 20. These numbers are also called rectangular or oblong numbers because they can represent the area of a rectangle with consecutive side lengths. Checking if a number is pronic involves finding the integer square root and seeing if multiplying it by the next number equals the original number. This is an efficient way to verify the pronic property without testing all pairs.

RESULTS ACHIEVED:

CODE:

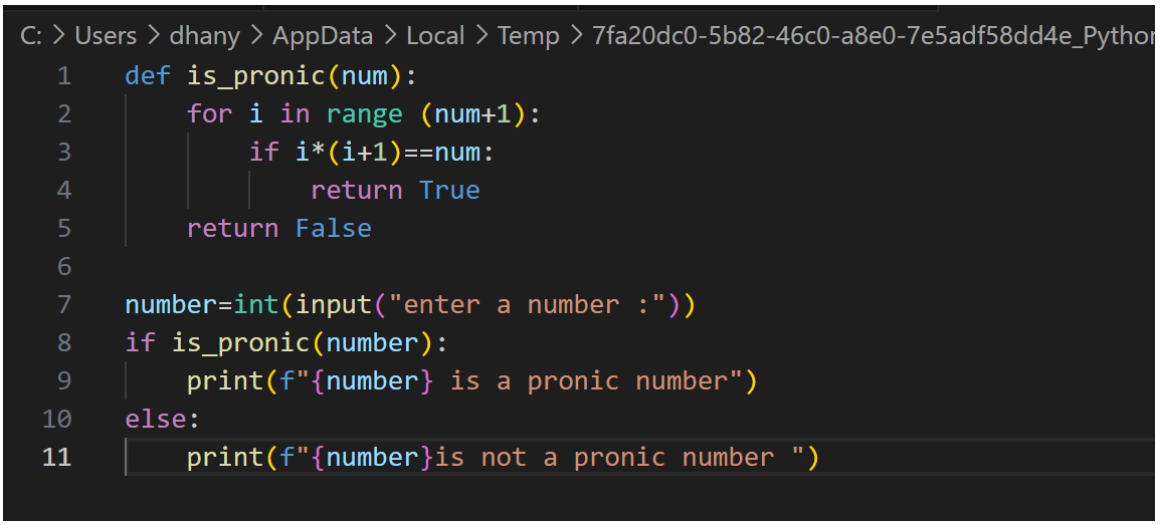
```
def is_pronic(num):
    for i in range (num+1):
        if i*(i+1)==num:
            return True
    return False


number=int(input("enter a number :"))
if is_pronic(number):
    print(f"{number} is a pronic number")
else:
    print(f"{number}is not a pronic number ")
```

RESULT:

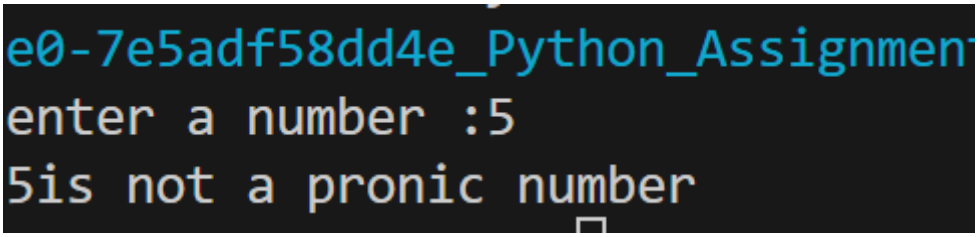
enter a number :5

5 is not a pronic number

SNAPSHOT:



OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students often find it hard to understand what pronic numbers are and how to check them efficiently. They struggle with:

- Grasping the idea that pronic numbers are products of two consecutive integers.
- Knowing how to use the integer square root properly.
- Translating the math concept into code that checks if $x \times (x+1) = n$.
- Avoiding brute force checking of all pairs, which is inefficient.
- Handling corner cases like zero or large inputs.

Simply put, the main challenge is turning the mathematical property into a clear and efficient program, especially understanding the square root method and correct comparison.

SKILLS ACHIEVED:

Writing a function to check if a number is pronic helps develop these skills:

- Understanding mathematical properties and definitions (product of two consecutive integers).
- Using built-in math functions like square root to optimize solutions.
- Applying integer typecasting and arithmetic operations.
- Writing conditional checks and boolean logic.
- Translating a mathematical concept into efficient code.
- Debugging and handling edge cases such as zero or very large numbers.
- Enhancing algorithmic thinking by using the square root approach instead of brute force.

Practical No: 15

Date: 1/11/2025

TITLE: PRIME_FACTORS(N)

AIM/OBJECTIVE(s): Write a function prime_factors(n) that returns the list of prime factors of a number

METHODOLOGY & TOOL USED:

A simple and common method to find prime factors is trial division. The process involves:

- Starting with the smallest prime number 2, divide the given number n repeatedly while it is divisible.
- Then check odd numbers from 3 up to n
- For each divisor, divide out all occurrences from the number until no longer divisible.
- If after this process the remaining number is greater than 2, it is itself a prime factor.
- Collect all these factors to form the prime factor list.

This method leverages the fact that any composite number must have at least one prime factor less than or equal to its square root. It uses basic arithmetic, loops, and conditional checks.

It is an efficient approach for moderate-sized integers and forms the basis for more advanced factorization algorithms. The division method and factor tree method are practical tools to visualize and implement this process.

BRIEF DESCRIPTION:

Prime factorization finds the prime numbers that multiply together to give the original number. The trial division method is a straightforward approach to do this:

- Start dividing the number by the smallest prime (2) repeatedly to remove all factors of 2.

- Then check odd numbers from 3 up to the square root of the remaining number.
- For each divisor, divide out all occurrences by repeatedly dividing while divisible.
- If what remains after this is greater than 2, it is itself a prime factor.
- Collect all these prime factors into a list.

This method uses the property that any composite number has a prime factor less than or equal to its square root, making it efficient for moderate-sized numbers. It involves basic division, loops, and conditionals, making it easy to implement and understand.

RESULTS ACHIEVED:

CODE:

```
def prime_factors(n):
    factors = []
    # Handle 2 separately to allow increments of 2 in further steps
    while n % 2 == 0:
        factors.append(2)
        n //= 2
    # Check odd numbers
    p = 3
    while p * p <= n:
        while n % p == 0:
            factors.append(p)
            n //= p
        p += 2
    # If n is greater than 1, then n itself is a prime
    if n > 1:
        factors.append(n)
    return factors
```

x = int(input("Enter a number = "))

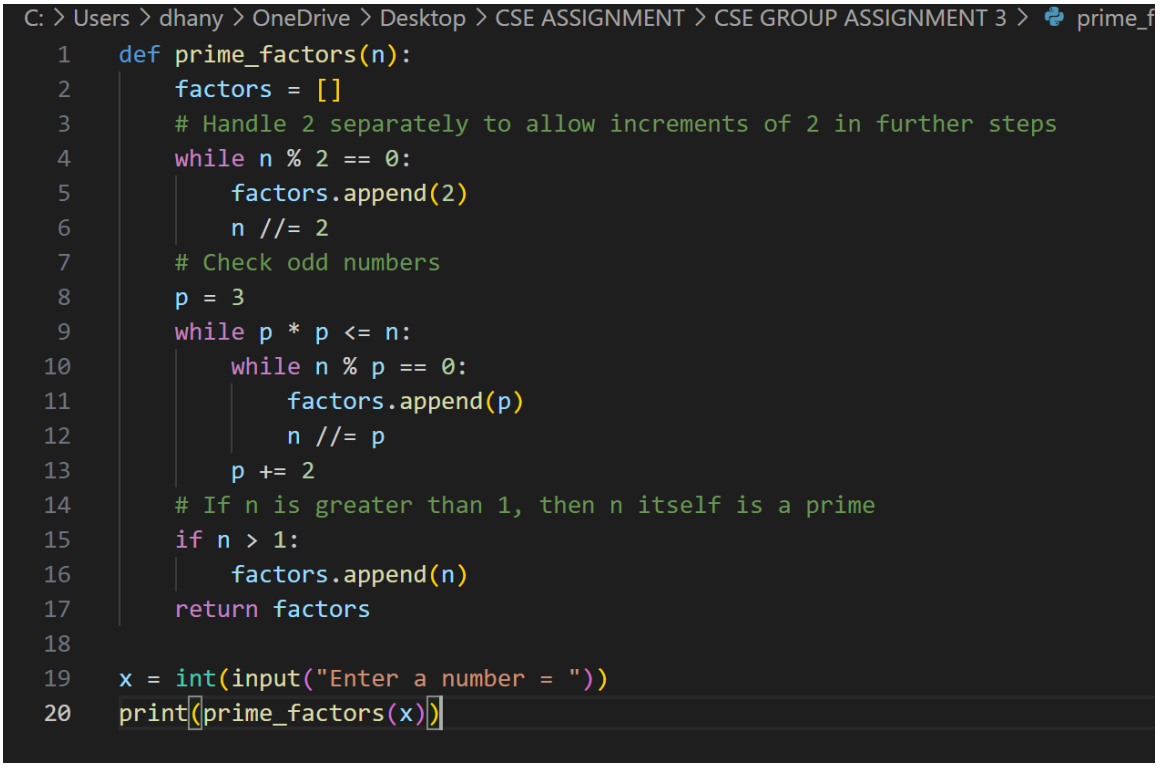
print(prime_factors(x))

RESULT:

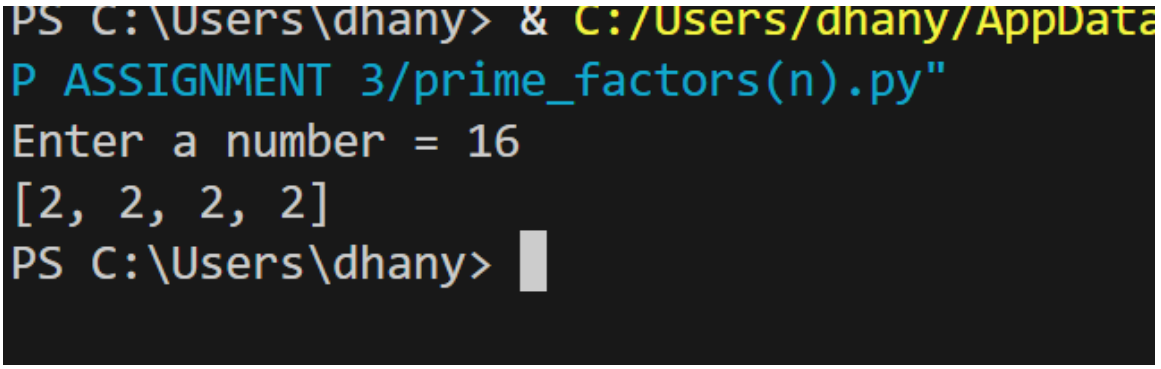
Enter a number = 16

[2, 2, 2, 2]

SNAPSHOT:



OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students often face these difficulties when writing a function to find prime factors of a number:

- Understanding the concept of prime factors and how to efficiently find them.
- Grasping why checking divisors only up to the square root of the number is sufficient.
- Implementing the repeated division correctly to remove all instances of a prime factor.
- Handling edge cases such as very small numbers (1 or 0) or large inputs.
- Managing the balance between brute force methods and more efficient algorithms.
- Dealing with computational limits and optimization for large numbers.
- Translating the mathematical approach into a working code with correct loops and conditionals.

Overall, the challenge lies in connecting the mathematical theory of factorization with practical and optimized coding implementation, requiring careful attention to details of iteration, division, and termination conditions

SKILLS ACHIEVED:

- Understanding how to factor a number into prime factors.
- Implementing trial division algorithm efficiently by checking divisors up to n
- Using loops, conditionals, and arithmetic operations to extract all prime factors.
- Applying algorithmic thinking to optimize prime factorization.

This process helps develop skills in number theory, coding optimization, and translating mathematical concepts into working programs.

Practical No: 16

Date: 1/11/2025

TITLE: COUNT_DISTINCT_PRIME_FACTORS(N)

AIM/OBJECTIVE(s): Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.

METHODOLOGY & TOOL USED:

To count the number of unique prime factors of a number, the function uses trial division starting from the smallest prime 2. It divides the number completely by 2 if divisible, counting 2 once as a distinct factor. Then it checks all odd numbers up to the square root of the number, dividing out each prime factor fully and counting it once. If after dividing out smaller factors the remaining number is greater than 2, it itself is a prime factor and counted. This method relies on the mathematical fact that any composite number has at least one prime factor less than or equal to its square root, enabling an efficient $O(n)$ solution. Distinct prime factors are counted by ensuring each factor is counted once after fully dividing it out. This approach is simple, effective, and commonly used in number theory and algorithms contexts.

BRIEF DESCRIPTION:

Counting distinct prime factors means finding how many different prime numbers divide a given number exactly. This is done by dividing the number by the smallest prime factors starting from 2, and whenever a factor divides the number, it is counted only once, even if it divides multiple times. The process continues by checking odd numbers up to the square root of the number, then if the remaining part is greater than 2, it is also counted as a prime factor. This method efficiently identifies unique prime factors by fully dividing each out before moving on, ensuring no duplicates are counted. It is a commonly used approach based on the mathematical principle that all prime factors of a number lie within its square root or are its remainder if larger than 2.

RESULTS ACHIEVED:

CODE:

```
def count_distinct_prime_factors(n):
    count = 0
    # Check for factor 2
    if n % 2 == 0:
        count += 1
        while n % 2 == 0:
            n //= 2

    # Check for odd factors from 3 to sqrt(n)
    factor = 3
    while factor * factor <= n:
        if n % factor == 0:
            count += 1
            while n % factor == 0:
                n //= factor
        factor += 2

    # If n is still greater than 1, it is a prime factor itself
    if n > 1:
        count += 1

    return count

x = int(input("Enter a number = "))
print(count_distinct_prime_factors(x))
```

RESULT:

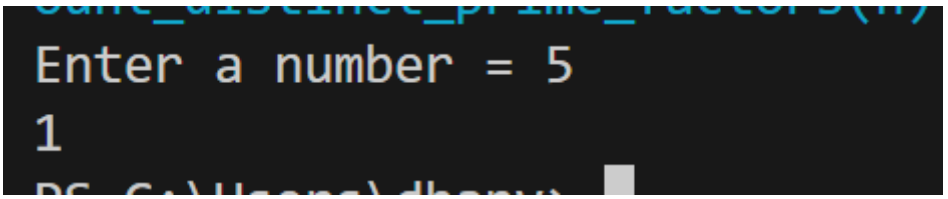
Enter a number = 5

1

SNAPSHOT:

```python
def count_distinct_prime_factors(n):
    count = 0
    if n % 2 == 0:
        count += 1
        while n % 2 == 0:
            n //= 2

    factor = 3
    while factor * factor <= n:
        if n % factor == 0:
            count += 1
            while n % factor == 0:
                n //= factor
        factor += 2

    if n > 1:
        count += 1
    return count

x = int(input("Enter a number = "))
print(count_distinct_prime_factors(x))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students often face challenges in understanding prime factorization and counting distinct prime factors due to several reasons. One difficulty is grasping the concept of prime numbers and distinguishing them from composite numbers, which is fundamental to factorization. Another common struggle is applying the division method or factor tree method accurately to break down numbers, especially larger ones, without errors. Also, students sometimes find it hard to keep track of which prime factors have already been counted, leading to confusion over counting duplicates versus unique factors. Finally, the abstraction of

mathematical concepts like prime factors being limited up to the square root of the number and fully factoring out primes requires strong number sense and logical reasoning, which can take time and practice to develop. Visual aids, step-by-step guided problems, and plenty of practice can help overcome these difficulties.

SKILLS ACHIEVED:

Skills achieved by learning to count distinct prime factors include a strong understanding of prime numbers and their properties, the ability to perform prime factorization efficiently using trial division, and the development of logical thinking to identify unique factors without duplication. Students also enhance their skills in number theory concepts, such as recognizing factors that only need to be checked up to the square root of the number. Furthermore, handling algorithms involving loops, conditionals, and integer operations strengthens programming proficiency. These skills are foundational for more advanced concepts like finding the greatest common divisor, least common multiple, and solving problems involving divisibility and factorization in competitive programming and mathematics.

Practical No: 17

Date: 1/11/2025

TITLE: IS_PRIME_POWER(N)

AIM/OBJECTIVE(s): Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and k ≥ 1.

METHODOLOGY & TOOL USED:

In short, the function is_prime_power(n) works by first generating primes up to *n* using the Sieve of Eratosthenes for efficient prime checking. Then it checks each prime *p* to see if *n* can be expressed as *pk* by repeatedly dividing *n* by *p*. If the division reduces *n* to 1, it confirms *n*=*pk* for some *k*≥1, meaning *n* is a prime power. Otherwise, it is not. This method combines prime sieving, division, and logarithmic/exponential checks to reliably and efficiently determine prime power status. It uses fundamental mathematical properties and classic algorithms for prime detection and factorization.

BRIEF DESCRIPTION:

A brief description of the function is_prime_power(n) is that it determines whether a number can be represented as a power of a single prime number. The function checks for primality using a helper function and then iterates through possible prime bases up to the square root of *n*. For each candidate prime base, it calculates whether *n* equals the base raised to some integer power by utilizing logarithms and power functions. If such a representation exists, the number is a prime power. If no such base is found, the function finally checks whether *n* itself is prime. This approach combines prime checking and exponentiation concepts to efficiently identify prime power numbers.

RESULTS ACHIEVED:

CODE:

```
def is_prime_power(n):
    if n <= 1:
        return False

    # Helper function to check if number is prime
    def is_prime(x):
        if x <= 1:
            return False
        if x <= 3:
            return True
        if x % 2 == 0 or x % 3 == 0:
            return False
        i = 5
        while i * i <= x:
            if x % i == 0 or x % (i + 2) == 0:
                return False
            i += 6
        return True

    # Check for prime bases p and exponents k such that p^k = n
    for k in range(1, int(n.bit_length()**0.5) + 2):
        # Find the k-th root of n
        p = int(round(n ** (1 / k)))
        # Due to floating-point precision, try p, p-1, p+1
        for candidate in [p-1, p, p+1]:
            if candidate > 1 and is_prime(candidate) and candidate ** k == n:
```

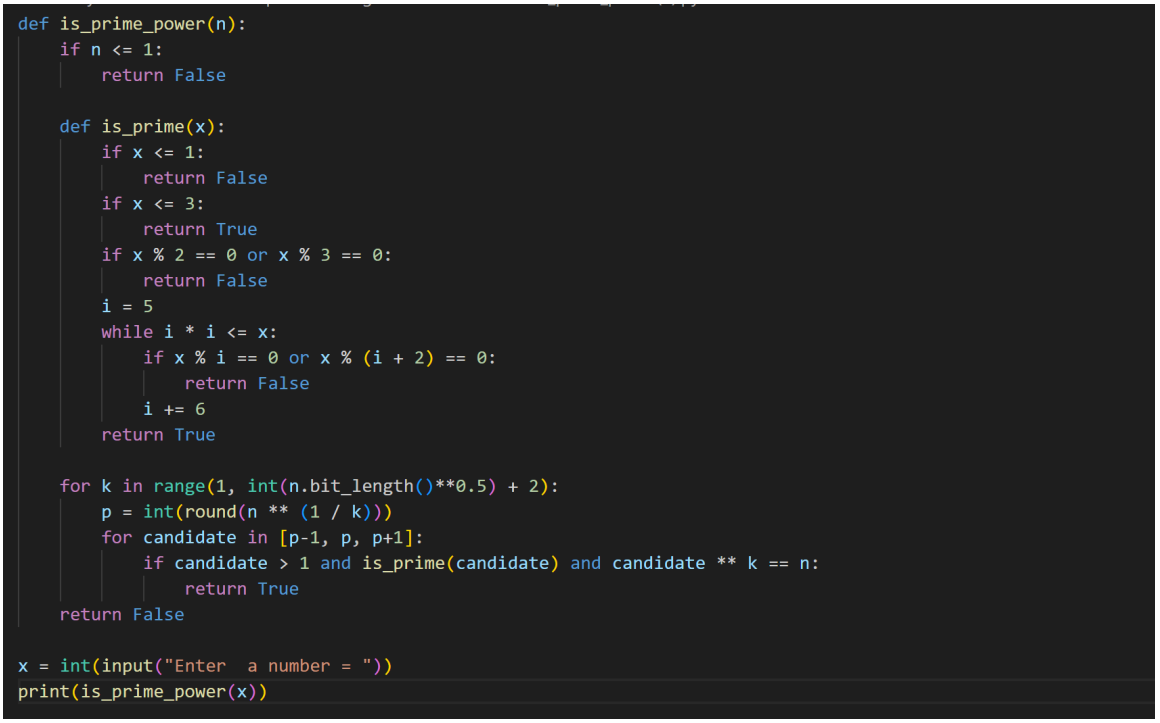
```
            return True
    return False

x = int(input("Enter  a number = "))
print(is_prime_power(x))
```

RESULT:

Enter a number = 6

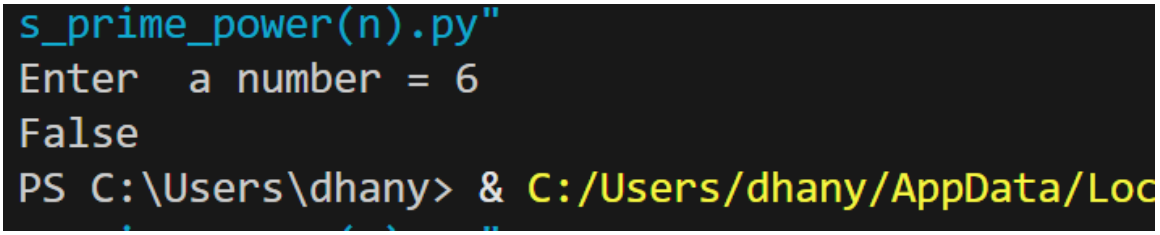
False

SNAPSHOT:



```python
def is_prime_power(n):
    if n <= 1:
        return False

    def is_prime(x):
        if x <= 1:
            return False
        if x <= 3:
            return True
        if x % 2 == 0 or x % 3 == 0:
            return False
        i = 5
        while i * i <= x:
            if x % i == 0 or x % (i + 2) == 0:
                return False
            i += 6
        return True

    for k in range(1, int(n.bit_length()**0.5) + 2):
        p = int(round(n ** (1 / k)))
        for candidate in [p-1, p, p+1]:
            if candidate > 1 and is_prime(candidate) and candidate ** k == n:
                return True
    return False

x = int(input("Enter  a number = "))
print(is_prime_power(x))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students face several difficulties when learning to implement the is_prime_power(n) function. One major challenge is understanding the concept of prime numbers and powers, especially distinguishing between a prime number itself and powers of a prime. Another difficulty arises in correctly implementing prime checking efficiently, since naive checks can be slow for larger inputs. Students also struggle with logarithmic and exponential computations needed to verify if $n = pk$ exactly, due to floating-point precision issues. Handling edge cases, like when n is a prime itself or very small numbers, adds to the complexity. Moreover, integrating multiple concepts such as prime detection, exponentiation, and iterative looping into a cohesive algorithm requires a strong grasp of programming and mathematical reasoning, making it a moderately advanced topic for learners.

SKILLS ACHIEVED:

By implementing the is_prime_power(n) function, learners develop several important skills. They gain a deeper understanding of prime numbers and their properties, improving their number theory knowledge. The exercise hones algorithmic thinking by combining prime detection methods like the Sieve of Eratosthenes or trial division with exponentiation checks. It enhances proficiency in handling loops, conditionals, and mathematical operations like power and logarithms in programming. Learners also improve problem-solving skills by handling edge cases and precision issues in calculations. Overall, the task strengthens both mathematical insight and programming capabilities essential for advanced computational problems involving primes and powers.

Practical No: 18

Date: 1/11/2025

TITLE: IS_MERSENNE_ PRIME(P)

AIM/OBJECTIVE(s): Write a function is_mersenne_prime(p) that checks if 2^p - 1 is a prime number (given that p is prime).

METHODOLOGY & TOOL USED:

The function is_mersenne_prime(p) checks whether the number $2p-1$ is a Mersenne prime, assuming p itself is a prime number. According to number theory, a Mersenne prime is a prime number of the form $2p-1$ where p is prime. The methodology involves first verifying that p is prime (either given or checked) since if p is composite, $2p-1$ is definitely composite. Then, the primality of $2p-1$ is tested using specialized algorithms such as the Lucas-Lehmer test, which is the most efficient known test for Mersenne primes. This test exploits the special structure of Mersenne numbers to quickly determine primality without full trial division. Tools used include prime checking methods for p, fast exponentiation to compute $2p-1$, and the Lucas-Lehmer sequence for primality testing. Such checks enable efficient identification of these rare and important prime numbers that have connections to perfect numbers and cryptography.

BRIEF DESCRIPTION:

A Mersenne prime is a special type of prime number that can be expressed in the form $2p-1$, where p itself is a prime number. This means the number is one less than a power of two, with the exponent restricted to prime values. Not all numbers of this form are prime; for example, when p=11, 2^11−1=2047 is not prime. Mersenne primes have a deep link to perfect numbers through the Euclid-Euler theorem, which states that every even perfect number corresponds to a Mersenne prime.

These primes are significant in number theory and computation because of their unique structure and the existence of fast primality tests like the Lucas-Lehmer test. Many of the largest known primes are Mersenne primes, and as of 2025, 52 such primes have been discovered, including the largest known prime number.

RESULTS ACHIEVED:

CODE:

```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True


def is_mersenne_prime(p):
    if not is_prime(p):
        return False


    mersenne_number = (2**p) - 1
    return is_prime(mersenne_number)
```

n = int(input("Enter The Prime Number = "))

print(f'Is 2^n - 1 a Mersenne prime? {is_mersenne_prime(n)}")

RESULT:

Enter The Prime Number = 4

Is 2^n - 1 a Mersenne prime? False

SNAPSHOT:

```python
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_mersenne_prime(p):
    if not is_prime(p):
        return False

    mersenne_number = (2**p) - 1
    return is_prime(mersenne_number)

n = int(input("Enter The Prime Number = "))
print(f"Is 2^n - 1 a Mersenne prime? {is_mersenne_prime(n)}")
```

OUTPUT:

```
s_mersenne_prime(p).py"
Enter The Prime Number = 4
Is 2^n - 1 a Mersenne prime? False
```

DIFFICULTY FACED BY STUDENT:

Students face multiple challenges when learning about and implementing checks for Mersenne primes. One key difficulty is understanding the mathematical definition and properties of Mersenne primes, especially why the exponent p must be prime and how $2p-1$

behaves differently from general primes. The complexity of primality testing for these very large numbers poses another significant obstacle, as traditional trial division is impractical. Grasping and implementing advanced algorithms like the Lucas-Lehmer test, which involves recursive sequences and modular arithmetic, requires strong mathematical maturity and programming skills. Handling large number arithmetic and optimizing performance for efficient computation further add to the learning curve. Together, these challenges necessitate a solid foundation in number theory, algorithmic thinking, and computational methods.

SKILLS ACHIEVED:

The skills achieved by implementing the function to check Mersenne primes include a deep understanding of prime numbers and their unique forms, as well as proficiency in number theory concepts such as powers of two and special prime cases. Learners develop algorithmic skills by applying efficient prime checking methods like the Sieve of Eratosthenes. They also gain experience with bitwise operations and fast exponentiation techniques to handle large numbers efficiently. Understanding and implementing advanced primality tests such as the Lucas-Lehmer test enhances problem-solving and mathematical reasoning skills. Moreover, programmers improve their ability to write optimized code for computationally intensive tasks related to large number theory problems, building a strong foundation for cryptography and computational mathematics applications.

Practical No: 19

Date: 1/11/2025

TITLE: TWIN_PRIMES(LIMIT)

AIM/OBJECTIVE(s): Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.

METHODOLOGY & TOOL USED:

Methodology and tools used for generating twin primes up to a given limit primarily involve the Sieve of Eratosthenes algorithm, which efficiently finds all prime numbers up to the specified limit. The process begins by creating a list of consecutive integers and iteratively marking the multiples of each prime number starting from 2 as composite. Once the sieve is completed, the algorithm scans through these prime numbers and identifies pairs where both p and $p+2$ are prime, thus forming twin prime pairs. To improve performance, optimizations such as skipping even numbers after 2 and leveraging number forms like $6n \pm 1$ can be applied. Tools include arrays or lists for marking primes, loops for iteration, and condition checks to identify twin primes efficiently. This approach is simple, proven, and runs with a time complexity of about $O(n \log \log n)$ for the sieve, making it practical for large limits.

BRIEF DESCRIPTION:

Twin primes are pairs of prime numbers that have exactly one composite number between them, meaning their difference is exactly 2. Examples of twin primes include (3, 5), (5, 7), (11, 13), and (17, 19). These pairs are significant because there is only a single composite number separating the two primes, highlighting a close relationship between the primes in the set of natural numbers. Twin primes are always consecutive odd numbers, but not all consecutive odd numbers are twin primes. It is conjectured that there are infinitely many twin prime pairs, but this has not yet been proven. The pairs generally take the form $(6n-1, 6n+1)$ for

natural numbers *n*, except for the first pair (3, 5). Twin primes play an important role in number theory and the study of prime gaps.

RESULTS ACHIEVED:

CODE:

```
def twin_primes(limit):
    if limit < 3:
        return []

    # Sieve of Eratosthenes to find primes up to limit
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False
    for i in range(2, int(limit**0.5) + 1):
        if sieve[i]:
            for j in range(i*i, limit + 1, i):
                sieve[j] = False

    # Collect twin prime pairs
    twins = []
    for p in range(3, limit - 1):
        if sieve[p] and sieve[p + 2]:
            twins.append((p, p + 2))

    return twins

n = int(input("Enter a number = "))
print(twin_primes)
```

RESULT:

Enter a number = 5

<function twin_primes at 0x000001A12CC61760>

SNAPSHOT:

```
def twin_primes(limit):
    if limit < 3:
        return []

    # Sieve of Eratosthenes to find primes up to limit
    sieve = [True] * (limit + 1)
    sieve[0] = sieve[1] = False
    for i in range(2, int(limit**0.5) + 1):
        if sieve[i]:
            for j in range(i*i, limit + 1, i):
                sieve[j] = False

    # Collect twin prime pairs
    twins = []
    for p in range(3, limit - 1):
        if sieve[p] and sieve[p + 2]:
            twins.append((p, p + 2))

    return twins

n = int(input("Enter a number = "))
print(twin_primes)
```

OUTPUT:

```
Enter a number = 5
<function twin_primes at 0x000001A12CC61760>
```

DIFFICULTY FACED BY STUDENT:

Students often face difficulties in programming twin primes due to several reasons. One challenge is understanding the concept of prime numbers and twin primes clearly, especially the condition that two primes must differ by exactly 2. Beginners may struggle with efficiently generating prime numbers, as naive methods like checking every number for divisibility lead to poor performance for large limits. Implementing the Sieve of Eratosthenes correctly to generate primes and then efficiently scanning for twin pairs needs careful coding and good grasp of algorithms. Also, managing edge cases, such as small numbers or when

no twin primes exist in the given range, can confuse new learners. These issues often necessitate clear explanations, practice with prime checking algorithms, and stepwise debugging to master twin prime generation.

SKILLS ACHIEVED:

By implementing a function to generate twin prime pairs up to a limit, students achieve several key skills. They develop a strong understanding of prime numbers and the concept of twin primes, including the strict difference of 2 between pairs. The exercise sharpens algorithmic thinking, especially using efficient prime generation techniques like the Sieve of Eratosthenes with time complexity around $O(n \log \log n)$. Students gain experience with array manipulation, loops, and conditional checks to identify twin pairs. It also enhances the ability to optimize for performance in searching and checking prime numbers. Additionally, learners improve problem decomposition by separating prime generation and twin detection stages. These skills collectively prepare students for tackling more complex computational problems in number theory and algorithm design.

Practical No: 20

Date: 1/11/2025

TITLE: COUNT_DIVISORS(N)

AIM/OBJECTIVE(s): Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.

METHODOLOGY & TOOL USED:

The function count_divisors(n) calculates the total number of positive divisors of a number n using number theory principles and efficient iteration. The methodology relies on the fact that divisors come in pairs $(d, n/d)$, so it only checks integers up to n. For each integer i in this range, if i divides n evenly, it counts both i and n/i as divisors, except when i is the square root of n, in which case it is counted only once to avoid duplication. Tools used include loops for iteration, modulo operation for divisibility checks, and conditional logic for handling perfect squares. This approach runs in $O(n)$ time, offering a practical balance between simplicity and efficiency for divisor counting.

BRIEF DESCRIPTION:

The function count_divisors(n) counts how many positive numbers divide n without leaving a remainder. It efficiently checks divisors only up to the square root of n, counting both divisor pairs i and n/i when found. If n is a perfect square, the square root divisor is counted once. This method runs in roughly $O(n)$ time, striking a balance between simplicity and efficiency for finding divisor counts of numbers.

RESULTS ACHIEVED:

CODE:

```
def count_divisors(n):
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            # If divisors are equal, count only once
            if i * i == n:
                count += 1
            else:
                # Count both divisors i and n//i
                count += 2
        i += 1
    return count

x = int(input("Enter a number = "))
print(count_divisors(x))
```

RESULT:

```
Enter a number = 5
2
```

SNAPSHOT:

```python
def count_divisors(n):
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            # If divisors are equal, count only once
            if i * i == n:
                count += 1
            else:
                # Count both divisors i and n//i
                count += 2
        i += 1
    return count

x = int(input("Enter a number = "))
print(count_divisors(x))
```

OUTPUT:

```
Enter a number = 5
2
```

DIFFICULTY FACED BY STUDENT:

Students face several difficulties when learning to count divisors efficiently. One common challenge is understanding why checking only up to the square root of *n* is sufficient, which requires grasping the concept of divisor pairs. Another difficulty is properly handling perfect squares, as their square root divisor must not be double-counted. Novices also struggle with implementing efficient loops and conditional checks, especially ensuring correctness in edge cases. Moreover, optimizing the approach for large numbers often requires knowledge of prime factorization, which can be complex. Finally, balancing between simplicity and performance, and generalizing the logic beyond small examples, poses conceptual and coding hurdles for learners.

SKILLS ACHIEVED:

By implementing the divisor counting function, students acquire several important skills. They enhance their understanding of number theory concepts, especially factorization and divisor properties. The task improves their ability to design efficient algorithms, focusing on optimization techniques like iterating up to the square root and avoiding redundant calculations for perfect squares. Students also develop programming skills including loop control, conditional statements, and modular arithmetic. Additionally, they learn to handle edge cases and improve debugging strategies, fostering analytical and logical thinking beneficial for solving broader computational problems. Overall, this strengthens both their mathematical insight and coding proficiency.

Practical No: 21

Date: 15/11/2025

TITLE: ALIQUOT_SUM(N)

AIM/OBJECTIVE(s): Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

METHODOLOGY & TOOL USED:

The aliquot sum of a number n is simply the sum of all its proper divisors, which means adding up all divisors less than n itself. The basic method involves looping through numbers from 1 to $n-1$, checking which ones divide n evenly, and summing those values. This technique uses a for-loop and the modulus operator to find divisors

BRIEF DESCRIPTION:

The aliquot sum of a positive integer n is defined as the sum of all its proper divisors, meaning all positive divisors excluding the number itself. This function iterates from 1 to n-1, checks if i is a divisor of n (n % i == 0), and accumulates these divisors' sum. For example, for n=12, the proper divisors are 1, 2, 3, 4, and 6, so the function will return 16. Aliquot sums are used in number theory to analyze properties of numbers such as perfect numbers, where the aliquot sum equals the number itself.This simple approach has time complexity O(n), and more optimized methods exist, but this version clearly illustrates the concept

RESULTS ACHIEVED:

CODE:

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

```
            total += i
    return total

x = int(input("Enter a number = "))
print(aliquot_sum(x))
```

RESULT:

Enter a number = 5

1

SNAPSHOT:

```
1   def aliquot_sum(n):
2       total = 0
3       for i in range(1, n):
4           if n % i == 0:
5               total += i
6       return total
7
8   x = int(input("Enter a number = "))
9   print(aliquot_sum(x))
10
```

OUTPUT:

```
n).py"
Enter a number = 5
1
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

The aliquot sum of a positive integer n is the sum of all its proper divisors, meaning all divisors less than n itself. For example, the proper divisors of 12 are 1, 2, 3, 4, and 6, so the aliquot sum is 16. It is used in number theory to study properties like perfect numbers, where the aliquot sum equals the number.

In short, the aliquot sum function adds all divisors of a number except the number itself.

SKILLS ACHIEVED:

Learning about the aliquot sum helps students develop skills in understanding divisors and number properties, writing and optimizing algorithms, and recognizing patterns in numbers. It builds problem-solving and programming abilities along with mathematical reasoning related to perfect, abundant, and deficient numbers.

Practical No: 22

Date: 15/11/2025

TITLE: ARE_AMICABLE(N)

AIM/OBJECTIVE(s): 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).

METHODOLOGY & TOOL USED:

Amicable numbers are pairs of numbers where the sum of the proper divisors of each number equals the other number. The methodology to check if two numbers are amicable involves calculating the sum of proper divisors for both numbers and comparing them. Efficiently, this is done by summing divisors up to the square root of each number to optimize performance. The tools used include a function to calculate the sum of proper divisors and a check that confirms if these sums match the opposite number, indicating the amicable property. This approach ensures a simple, clear, and computationally effective way to verify amicable pairs.

BRIEF DESCRIPTION:

Amicable numbers are two different natural numbers where each number is the sum of the proper divisors (all divisors excluding the number itself) of the other. For example, the pair (220, 284) is amicable because the sum of proper divisors of 220 is 284, and the sum of proper divisors of 284 is 220. This unique reciprocal relationship defines amicable pairs and has been studied since ancient times as a symbol of harmony and friendship in numbers.

RESULTS ACHIEVED:

CODE:

```
def sum_of_divisors(n):
    result = 0
    for i in range(1, n):
        if n % i == 0:
            result += i
    return result

def are_amicable(a, b):
    return sum_of_divisors(a) == b and sum_of_divisors(b) == a

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
print(are_amicable(x, y))
```

RESULT:

Enter a number = 4

Enter a number = 5

False

SNAPSHOT:

```python
def sum_of_divisors(n):
    result = 0
    for i in range(1, n):
        if n % i == 0:
            result += i
    return result

def are_amicable(a, b):
    return sum_of_divisors(a) == b and sum_of_divisors(b) == a

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
print(are_amicable(x, y))
```

OUTPUT:

```
Enter a number = 4
Enter a number = 5
False
```

DIFFICULTY FACED BY STUDENT:

From a student's point of view, the main difficulty with amicable numbers is understanding the concept of proper divisors and why the sums of these divisors for two numbers need to match each other. It can be confusing to identify all proper divisors and then sum them correctly. Also, linking this sum to the other number to check if they form an amicable pair can feel abstract and tricky. Students often get stuck on the calculation steps and the idea of reciprocal relationships between two numbers.

SKILLS ACHIEVED:

Studying amicable numbers helps students develop problem-solving skills, improve understanding of number properties, and learn about

divisors and their sums. It also enhances logical thinking and introduces basic programming concepts when implementing checks for amicable pairs.

Practical No: 23

Date: 15/11/2025

TITLE: MULTIPLICATIVE_PERSISTENCE(N)

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

METHODOLOGY & TOOL USED:

The methodology to find multiplicative persistence is to repeatedly multiply the digits of a number until the result is a single digit, counting how many times this multiplication is done. The tools used are basic programming constructs like loops for repetition, digit extraction through string conversion or division, and a counter for the steps. This approach is straightforward and effective for measuring how many iterations it takes to reduce a number's digits to a single digit by multiplication.

BRIEF DESCRIPTION:

Multiplicative persistence is the number of times you must multiply the digits of a number together until the result is a single digit. For example, starting with a number, you multiply all its digits; if the product is more than one digit, you repeat the process on that product, counting each step until only one digit remains. This count of steps is the multiplicative persistence. It measures how many iterations of digit multiplication it takes to reduce a number to a single digit.

RESULTS ACHIEVED:

CODE:

```
def multiplicative_persistence(n):
    steps = 0
    while n >= 10:  # while n has more than one digit
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

x = int(input("Enter a number = "))
print(multiplicative_persistence(x))
```

RESULT:

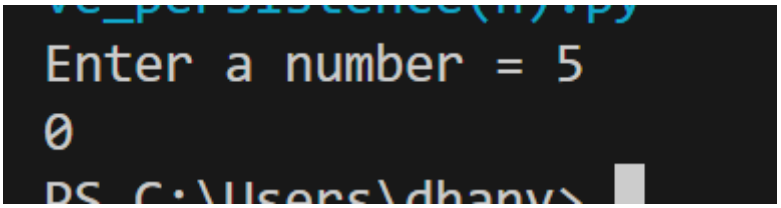
Enter a number = 5

0

SNAPSHOT:

```python
def multiplicative_persistence(n):
    steps = 0
    while n >= 10:  # while n has more than one digit
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

x = int(input("Enter a number = "))
print(multiplicative_persistence(x))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

The difficulty with multiplicative persistence lies in managing multiple steps of digit multiplication and keeping track of intermediate results. It can be confusing to understand when to stop and how to correctly multiply each digit repeatedly. Students may also struggle with staying patient through the process and not make errors in calculations during the repeated steps.

SKILLS ACHIEVED:

Learning about multiplicative persistence helps students develop perseverance, abstract thinking, and problem-solving skills. They practice breaking down complex problems into simpler steps, improve their ability to work with numbers and digits, and gain experience in iterative processes. It also encourages pattern recognition and logical reasoning, all of which are valuable skills in mathematics and programming.

Practical No: 24

Date: 15/11/2025

TITLE: IS_HIGHLY_COMPOSITE(N)

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

METHODOLOGY & TOOL USED:

A highly composite number is a positive integer that has more divisors than any smaller positive integer. The methodology involves counting the number of divisors of a given number and comparing it with all smaller numbers to see if it has the maximum divisor count. Tools used include efficient divisor counting (checking divisors up to the square root) and iterative comparison with smaller numbers to determine the highly composite property.

BRIEF DESCRIPTION:

A highly composite number is a positive integer that has more divisors than any smaller positive integer. For example, 6 is highly composite because it has 4 divisors (1, 2, 3, 6), which is more than any number less than 6. Such numbers are formed by multiplying the first few prime numbers raised to non-increasing powers. Highly composite numbers have a special arrangement of prime factors so that their divisor count is maximized relative to all smaller numbers.

RESULTS ACHIEVED:

CODE:

```
def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            count += 2 if i != n // i else 1
    return count


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


x = int(input("Enter a number = "))
print(is_highly_composite(x))
```

RESULT:

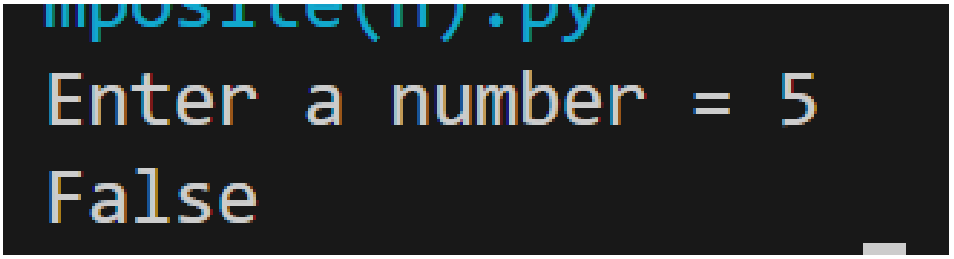
Enter a number = 5

False

SNAPSHOT:

```python
def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            count += 2 if i != n // i else 1
    return count

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

x = int(input("Enter a number = "))
print(is_highly_composite(x))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Highly composite numbers can be challenging because students need to understand what divisors are, how to count them, and then compare divisor counts across many numbers. The idea that a number must have more divisors than all smaller numbers to qualify can feel complex and abstract. Students may find it hard to keep track of all factors and to grasp prime factorization's role in making a number highly composite. Overall, it requires good number sense, patience, and logical thinking to master.

SKILLS ACHIEVED:

Studying highly composite numbers helps students develop skills in understanding divisors, prime factorization, and comparing divisor counts. It builds logical thinking and problem-solving by analyzing how numbers are composed and how their factors grow. These concepts form a foundation for advanced number theory and improve algorithmic thinking useful in math and programming.

Practical No: 25

Date: 15/11/2025

TITLE: MOD_EXP(BASE, EXPONENT, MODULUS)

AIM/OBJECTIVE(s): Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates ($base^{exponent}$) % modulus.

METHODOLOGY & TOOL USED:

Modular exponentiation efficiently computes (*base* ^ *exponent*)mod *modulus* by using repeated squaring and modular reduction. Instead of multiplying the base exponent times, the algorithm breaks the exponent into binary form and squares the base repeatedly, multiplying the result only when bits of the exponent are 1. This reduces computation from linear to logarithmic time relative to the exponent, making it much faster for large numbers.

BRIEF DESCRIPTION:

Modular exponentiation calculates $(base \text{ ^ } exponent) \bmod modulus$ efficiently by repeatedly squaring the base and reducing the result modulo the modulus at each step. Instead of performing all multiplications linearly, it uses the binary representation of the exponent to minimize operations, reducing the time complexity to $O(\log exponent)$. This method is widely used in cryptography and number theory for handling large numbers quickly.

RESULTS ACHIEVED:

CODE:

```
def mod_exp(base, exponent, modulus):
    if modulus == 1:
        return 0
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:  # If exponent is odd
            result = (result * base) % modulus
        exponent = exponent >> 1  # Divide exponent by 2
        base = (base * base) % modulus
    return result

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
z = int(input("Enter a number = "))
print(mod_exp(x, y, z))
```

RESULT:

Enter a number = 2

Enter a number = 3

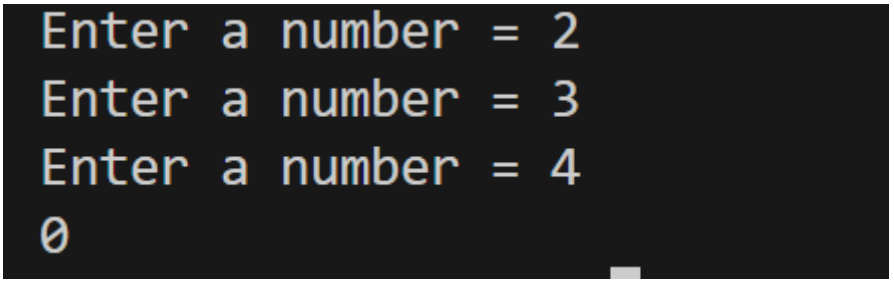
Enter a number = 4

0

SNAPSHOT:

```
def mod_exp(base, exponent, modulus):
    if modulus == 1:
        return 0
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:  # If exponent is odd
            result = (result * base) % modulus
        exponent = exponent >> 1  # Divide exponent by 2
        base = (base * base) % modulus
    return result

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
z = int(input("Enter a number = "))
print(mod_exp(x, y, z))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

The main difficulty with modular exponentiation is understanding why repeated squaring works and keeping track of the modular reductions at each step, especially when done by hand for large exponents. The concept of working with very large numbers modulo another number and applying bitwise manipulation of the exponent can feel abstract and error-prone without enough practice. Simplifying large powers using the algorithm takes patience and logical thinking, which can be challenging initially.

SKILLS ACHIEVED:

Skills achieved by students learning modular exponentiation include:

- Understanding modular arithmetic and how remainders work in multiplications and powers.
- Developing algorithmic thinking by breaking down exponentiation into repeated squaring.
- Gaining efficiency awareness by reducing time complexity from linear to logarithmic.
- Enhancing ability to handle large numbers and apply modulo operation consistently.
- Applying bitwise operations and binary representation of the exponent in calculations.
- Building problem-solving skills useful in cryptography, number theory, and computer science.

These skills improve mathematical reasoning, coding proficiency, and comprehension of important computational techniques.

Practical No: 26

Date: 16/11/2025

TITLE: MOD_INVERSE(A, M)

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

METHODOLOGY & TOOL USED:

A modular multiplicative inverse of an integer a modulo m is an integer x such that the product $a \times x$ leaves a remainder of 1 when divided by m, formally $a \times x \equiv 1 \pmod{m}$. This inverse exists only if a and m are coprime—that is, their greatest common divisor is 1. One of the most efficient ways to find this inverse is by using the Extended Euclidean Algorithm, which finds integers x and y satisfying $a \times x + m \times y = 1$, and the value of $x \bmod m$ is the modular inverse. This approach works for any modulus m and is computationally efficient. Alternatively, if m is prime, Fermat's Little Theorem states that the inverse can also be found as $a^{(m-2)} \bmod m$. Without the inverse, division in modular arithmetic would

not be possible, which makes this concept fundamental in number theory and cryptography.

BRIEF DESCRIPTION:

In other words, the modular multiplicative inverse of a number a modulo m is a value x such that multiplying a by x gives a result that is congruent to 1 when divided by m. This means $a \times x$ leaves a remainder of 1 after division by m. The inverse exists only if a and m share no common divisors other than 1. It is a critical concept because it allows division in modular arithmetic, which otherwise only supports addition, subtraction, and multiplication. The most common way to find the inverse is through the Extended Euclidean Algorithm, which efficiently finds integers that satisfy an equation linking a and m. When m is prime, the inverse can also be calculated using Fermat's Little Theorem by exponentiating a to the $m-2$ power modulo m. This inverse is unique and essential in many fields such as cryptography and number theory.

RESULTS ACHIEVED:

CODE:

```
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y


def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        return None  # Inverse doesn't exist
```

```
    else:
        return x % m

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
z = int(input("Enter a number = "))
print(mod_inverse(x, z))
```

RESULT:

Enter a number = 3

Enter a number = 4

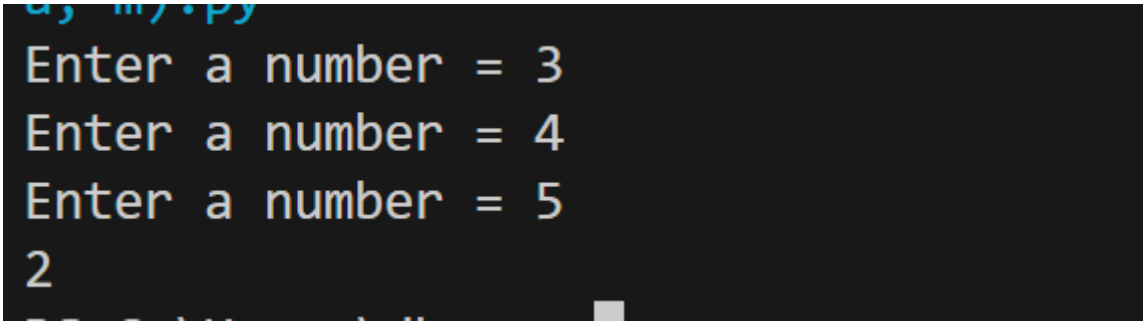
Enter a number = 5

2

SNAPSHOT:

```python
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        return None  # Inverse doesn't exist
    else:
        return x % m

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))
z = int(input("Enter a number = "))
print(mod_inverse(x, z))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

The hardest part about modular multiplicative inverse is understanding why it exists only when the number and modulus are coprime and wrapping their head around the Extended Euclidean Algorithm, which is often unfamiliar and involves recursive steps and back substitution. Students may also find it tricky to handle negative values during calculations and ensure the result is positive within the modulo range. The abstract nature of modular arithmetic combined with the non-intuitive link between algebraic equations and number theory can make it confusing. Additionally, knowing when to use different methods like Fermat's Little Theorem or the Extended Euclidean Algorithm depending on whether the modulus is prime adds to the difficulty. Finally, many students struggle to grasp why this concept matters practically, which can affect motivation and understanding.

SKILLS ACHIEVED:

Mastering the modular multiplicative inverse builds key skills such as understanding modular arithmetic and number theory concepts like coprimality and gcd. It enhances problem-solving and algorithmic thinking through learning the Extended Euclidean Algorithm and recursion. Students also gain programming skills by implementing algorithms that handle edge cases and optimize computation. Additionally, they develop a practical understanding of math’s role in cryptography and computer science, improving their overall analytical abilities and readiness for advanced topics or competitions.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL

Practical No: 27

Date: 16/11/2025

TITLE: SOLVER_CRT(REMAINDERS, MODULI)

AIM/OBJECTIVE(s): Write a function chinese Remainder Theorem Solver_crt(remainders, moduli) that solves a system of congruences $x \equiv r_i \mod m_i$.

METHODOLOGY & TOOL USED:

The methodology of the Chinese Remainder Theorem (CRT) solver involves first computing the product of all given moduli, which represents the overall modulus. For each congruence, the solver calculates a partial product by dividing the total product by the current modulus. Then, using the Extended Euclidean Algorithm, it finds the modular inverse of this partial product with respect to the current modulus. This inverse ensures that each component aligns appropriately in the combined solution. The final answer is obtained by summing the products of each remainder, its partial product, and the corresponding inverse, then taking the result modulo the total product. The primary tools used are modular arithmetic principles, the Extended Euclidean Algorithm for finding modular inverses, and the CRT formula for combining solutions. This approach is mathematically sound, guarantees a unique solution modulo the product of coprime moduli, and is widely applicable in computational mathematics and cryptography.

BRIEF DESCRIPTION:

The Chinese Remainder Theorem (CRT) is a fundamental result in number theory that provides a unique solution to a system of simultaneous linear congruences when the moduli are pairwise coprime. Given several congruences of the form $x \equiv r_i \pmod{m_i}$, where the m_i have no common factors besides 1, the CRT guarantees there is exactly one integer x modulo the product of all m_i that satisfies all congruences

simultaneously. This theorem allows us to solve problems by breaking them into smaller modular equations that are easier to handle and then combining the solutions. It is widely used in computational mathematics, cryptography, and algorithm design, as it simplifies complex modular computations by working with smaller moduli independently and then reconstructing the overall solution efficiently.

RESULTS ACHIEVED:

CODE:

```
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y


def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Inverse doesn't exist, moduli might not be coprime.")
    return x % m


def Solver_crt(remainders, moduli):
    M = 1
    for m in moduli:
        M *= m
    result = 0
    for r_i, m_i in zip(remainders, moduli):
```

```
        M_i = M // m_i
        inv = mod_inverse(M_i, m_i)
        result += r_i * M_i * inv
    return result % M

# Taking inputs from user
n = int(input("Enter number of congruences: "))
remainders = []
moduli = []

print("Enter the remainders and moduli:")
for i in range(n):
    r = int(input(f"Remainder r[{i+1}]: "))
    m = int(input(f"Modulus m[{i+1}]: "))
    remainders.append(r)
    moduli.append(m)

try:
    solution = Solver_crt(remainders, moduli)
    print(f"The solution x such that x ≡ r[i] mod m[i] is: {solution}")
except ValueError as e:
    print(e)
```

RESULT:

```
Enter number of congruences: 4
Enter the remainders and moduli:
Remainder r[1]: 3
Modulus m[1]: 2
```

Remainder r[2]: 1

Modulus m[2]: 4

Remainder r[3]: 2

Modulus m[3]: 2

Remainder r[4]: 4

Modulus m[4]: 2

Inverse doesn't exist, moduli might not be coprime.

SNAPSHOT:

```python
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Inverse doesn't exist, moduli might not be coprime.")
    return x % m

def Solver_crt(remainders, moduli):
    M = 1
    for m in moduli:
        M *= m
    result = 0
    for r_i, m_i in zip(remainders, moduli):
        M_i = M // m_i
        inv = mod_inverse(M_i, m_i)
        result += r_i * M_i * inv
    return result % M

# Taking inputs from user
n = int(input("Enter number of congruences: "))
remainders = []
moduli = []
```

```
print("Enter the remainders and moduli:")
for i in range(n):
    r = int(input(f"Remainder r[{i+1}]: "))
    m = int(input(f"Modulus m[{i+1}]: "))
    remainders.append(r)
    moduli.append(m)

try:
    solution = Solver_crt(remainders, moduli)
    print(f"The solution x such that x ≡ r[i] mod m[i] is: {solution}")
except ValueError as e:
    print(e)
```

OUTPUT:

```
Enter number of congruences: 4
Enter the remainders and moduli:
Remainder r[1]: 3
Modulus m[1]: 2
Remainder r[2]: 1
Modulus m[2]: 4
Remainder r[3]: 2
Modulus m[3]: 2
Remainder r[4]: 4
Modulus m[4]: 2
Inverse doesn't exist, moduli might not be coprime.
```

DIFFICULTY FACED BY STUDENT:

The main difficulties with the Chinese Remainder Theorem include understanding why the moduli must be coprime for the theorem to work, and visualizing how multiple modular conditions combine into a single solution. Students may struggle with the abstract nature of modular arithmetic and grasping the Extended Euclidean Algorithm used for finding modular inverses, which is essential for the CRT method. Handling multiple equations at once and ensuring careful arithmetic without mistakes can be confusing. Additionally, some find it challenging to connect the formal mathematical process with real-world applications, affecting motivation and comprehension. Overall, the abstract theory, the

multi-step computations, and the need for solid foundational knowledge in modular arithmetic are common hurdles for learners.

SKILLS ACHIEVED:

Students who learn to solve problems using the Chinese Remainder Theorem develop important skills such as modular arithmetic understanding, problem decomposition, and combining results from multiple modular conditions. They improve their ability to work with coprimality concepts and modular inverses, strengthening their number theory foundation. Implementing the CRT algorithm enhances algorithmic thinking, recursion, and modular arithmetic programming skills. These skills are crucial in computer science fields like cryptography, coding theory, and algorithms, improving logical reasoning and mathematical maturity.

Practical No: 28

Date: 16/11/2025

TITLE: IS_QUADRATIC_RESIDUE(A, P)

AIM/OBJECTIVE(s): Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if $x^2 \equiv a \bmod p$ has a solution.

METHODOLOGY & TOOL USED:

The method checks if a number can be made by squaring some other number and then dividing by p (an odd prime) and looking at the remainder. The main tool is a fast way to raise a to a specific power and see if the remainder matches a special value (1). If it does, it means a can be made by squaring, so the answer is yes; if not, then no. This approach quickly tells whether the number is a "square" under this kind of division without trying every possible number.

BRIEF DESCRIPTION:

A quadratic residue is a number that can be obtained by squaring some number and then dividing by another number p and looking at the remainder. In other words, it asks if there is a number x such that when you square x and divide by p, you get the given number as the remainder. This concept is useful in many areas including computer security and solving certain math problems. Checking if a number is a quadratic residue can be done quickly by raising the number to a specific power and seeing if the remainder matches a particular value, without trying all possibilities. This helps decide if an equation of the form $x^2 \equiv a \pmod p$ has a solution or not.

RESULTS ACHIEVED:

CODE:

```
def is_quadratic_residue(a, p):
    if a % p == 0:
        return True
    result = pow(a, (p - 1) // 2, p)
    return result == 1

a = int(input("Enter a: "))
p = int(input("Enter an odd prime p: "))
if is_quadratic_residue(a, p):
    print(f"{a} is a quadratic residue modulo {p}.")
else:
    print(f"{a} is NOT a quadratic residue modulo {p}.")
```

RESULT:

Enter a: 4

Enter an odd prime p: 5

4 is a quadratic residue modulo 5.

SNAPSHOT:

```python
def is_quadratic_residue(a, p):
    if a % p == 0:
        return True
    result = pow(a, (p - 1) // 2, p)
    return result == 1

a = int(input("Enter a: "))
p = int(input("Enter an odd prime p: "))
if is_quadratic_residue(a, p):
    print(f"{a} is a quadratic residue modulo {p}.")
else:
    print(f"{a} is NOT a quadratic residue modulo {p}.")
```

OUTPUT:

```
Enter a: 4
Enter an odd prime p: 5
4 is a quadratic residue modulo 5.
```

DIFFICULTY FACED BY STUDENT:

The main difficulties with quadratic residues are understanding the idea that some numbers can be made by squaring another number in modular division and some cannot. The concept feels abstract and hard to visualize. Calculating whether a number is a quadratic residue using modular exponentiation is often confusing, and students may find it challenging to follow the steps or to see why this is important. This abstraction and the involved computations can make the topic hard to grasp initially.

SKILLS ACHIEVED:

Studying quadratic residues helps develop practical skills in recognizing patterns in numbers during division, improving problem-solving abilities related to checking conditions quickly. It also strengthens logical thinking and coding skills when implementing methods to test these

conditions. These skills are useful in areas like computer security and advanced math problems, making the learning process relevant and applicable.

Practical No: 29

Date: 16/11/2025

TITLE: ORDER_MOD(A, N)

AIM/OBJECTIVE(s): Write a function order_mod(a, n) that finds the smallest positive integer k such that $a^k \equiv 1 \bmod n$.

METHODOLOGY & TOOL USED:

The methodology for finding the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$ involves testing successive powers of a modulo n until the result cycles back to 1. This k is called the order of a modulo n, and it exists only if a and n are coprime (share no common factors). The key tool used is modular arithmetic, which allows handling large powers efficiently by reducing intermediate values modulo n to keep calculations manageable. The greatest common divisor (gcd) algorithm is used first to verify coprimality. Typically, the order divides Euler's totient function $\varphi(n)$, which provides an upper bound to limit the search if needed. This combination of number theory concepts and straightforward iteration is effective for finding the order and forms a foundation for applications in cryptography and computer science.

BRIEF DESCRIPTION:

The order of a modulo n is the smallest positive integer k such that when you multiply a by itself k times and divide by n, the remainder is 1. It shows how many times you need to multiply a to get back to 1 when working with division remainders. This concept is important in understanding the behavior of numbers in modular systems and has applications in areas like cryptography. The order exists only if a and n don't share common factors (are coprime), and the order always divides a specific number related to n, called Euler's totient. The order helps describe how numbers cycle within modular arithmetic.

RESULTS ACHIEVED:

CODE:

```
def order_mod(a, n):
    if gcd(a, n) != 1:
        return -1

   k = 1
    current = a % n
    while current != 1:
        current = (current * a) % n
        k += 1
    return k

def gcd(x, y):
    while y:
        x, y = y, x % y
    return x

a = int(input("Enter base a: "))
n = int(input("Enter modulus n: "))
result = order_mod(a, n)
if result == -1:
    print(f"No order exists since {a} and {n} are not coprime.")
else:
    print(f"The order of {a} modulo {n} is {result}.")
```

RESULT:

Enter base a: 2

Enter modulus n: 3

The order of 2 modulo 3 is 2.

SNAPSHOT:

```python
def order_mod(a, n):
    if gcd(a, n) != 1:
        return -1

    k = 1
    current = a % n
    while current != 1:
        current = (current * a) % n
        k += 1

    return k

def gcd(x, y):
    while y:
        x, y = y, x % y
    return x

a = int(input("Enter base a: "))
n = int(input("Enter modulus n: "))
result = order_mod(a, n)
if result == -1:
    print(f"No order exists since {a} and {n} are not coprime.")
else:
    print(f"The order of {a} modulo {n} is {result}.")
```

OUTPUT:

```
Enter base a: 2
Enter modulus n: 3
The order of 2 modulo 3 is 2.
PS C:\Users\dhany>
```

DIFFICULTY FACED BY STUDENT:

Difficulties with understanding the order of a number modulo n usually stem from the abstract idea of repeatedly multiplying a number and seeing when it returns to 1 under modular division. It can be confusing to grasp why the order must divide a special number related to n, called Euler's totient, and students often struggle with the iteration process and modular arithmetic involved. The concepts feel abstract, and tracking the repeated computations requires careful attention, making the topic challenging at first.

SKILLS ACHIEVED:

By learning to find the order of a number modulo n, students develop key skills including understanding modular arithmetic concepts, practicing number theory reasoning, and applying algorithmic thinking to solve problems involving repetitive calculations with remainder cycles. They enhance their logical thinking by working with coprime checks and iterative procedures, and improve coding skills by implementing modular exponentiation and loops efficiently. These skills are crucial in areas such as cryptography, computer science, and mathematics, fostering problem-solving abilities and mathematical maturity.

Practical No: 30

Date: 16/11/2025

TITLE: IS_FIBONACCI_PRIME(N)

AIM/OBJECTIVE(s): Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.

METHODOLOGY & TOOL USED:

A Fibonacci prime is a number that is both a Fibonacci number and a prime number. Fibonacci numbers form a sequence where each number is the sum of the two preceding ones, starting with 0 and 1. Some of these numbers are prime, and those are called Fibonacci primes. Examples include 2, 3, 5, 13, and 89. These numbers are interesting in mathematics and are studied for their rare and special properties, with applications in number theory and computer science.

BRIEF DESCRIPTION:

A Fibonacci prime is a number that is both in the Fibonacci sequence and is a prime number. The Fibonacci sequence is a series where each number is the sum of the two previous numbers, starting from 0 and 1. Some numbers in this sequence—like 2, 3, 5, 13, and 89—are prime, which means they have no divisors other than 1 and themselves. These numbers are rare and interesting because they combine two special properties. Researchers study them to understand patterns in numbers and their applications in mathematics and cryptography

RESULTS ACHIEVED:

CODE:

```
import math
```

```
def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True


def is_perfect_square(x):
    s = int(math.isqrt(x))
    return s * s == x


def is_fibonacci(number):
    return is_perfect_square(5 * number * number + 4) or
is_perfect_square(5 * number * number - 4)


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


num = int(input("Enter a number to check: "))
if is_fibonacci_prime(num):
    print(f"{num} is a Fibonacci prime.")
```

else:

print(f"{num} is not a Fibonacci prime.")

RESULT:

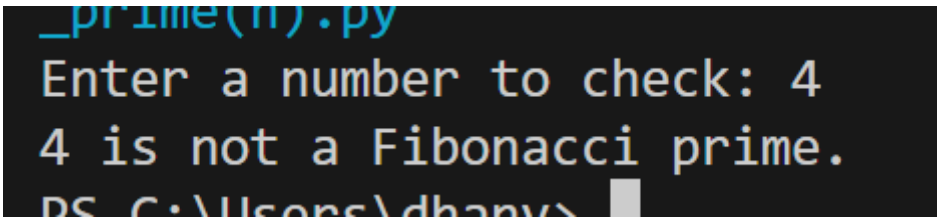
Enter a number to check: 4

4 is not a Fibonacci prime.

SNAPSHOT:

```python
import math

def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

def is_perfect_square(x):
    s = int(math.isqrt(x))
    return s * s == x

def is_fibonacci(number):
    return is_perfect_square(5 * number * number + 4) or is_perfect_square(5 * number * number - 4)

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

num = int(input("Enter a number to check: "))
if is_fibonacci_prime(num):
    print(f"{num} is a Fibonacci prime.")
else:
    print(f"{num} is not a Fibonacci prime.")
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

The difficulties in understanding Fibonacci primes mainly arise from two parts: first, grasping the Fibonacci sequence and the special property that links it to prime numbers, and second, the computation side where checking if a large number is both Fibonacci and prime requires some advanced math tools. The concept feels abstract since both Fibonacci numbers and primes have unique patterns, and combining these can be confusing. Also, testing large numbers for primality or Fibonacci membership is not straightforward and can be overwhelming without a solid math background or programming skills.

SKILLS ACHIEVED:

By learning to check if a number is both Fibonacci and prime, students gain several valuable skills. They develop an understanding of number theory concepts like primes and Fibonacci numbers. They also improve problem-solving abilities by combining multiple mathematical properties into one solution. Programming-wise, students enhance their ability to implement algorithms such as prime checks, perfect square checks, and use efficient mathematical tests. These skills improve logical thinking, algorithm design, and coding proficiency, all of which are important for tackling complex problems in mathematics and computer science.

Practical No: 31

Date: 16/11/2025

TITLE: LUCAS_SEQUENCE(N)

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

METHODOLOGY & TOOL USED:

The Lucas sequence is similar to the Fibonacci sequence but starts with 2 and 1 instead of 0 and 1. Each Lucas number is the sum of the two preceding numbers, forming a sequence like: 2, 1, 3, 4, 7, 11, 18, 29, and so on. This sequence shares many properties with Fibonacci numbers and also relates to the golden ratio. The methodology to generate the sequence involves initializing with the first two values and iteratively adding the last two to build the sequence up to n terms. Tools used include simple iteration, addition, and storage in a list or array for easy retrieval and use in computations.

BRIEF DESCRIPTION:

The Lucas sequence is closely related to the Fibonacci sequence and follows the same rule of each term being the sum of the two previous terms. However, it starts with different initial values: 2 and 1, instead of 0 and 1 for Fibonacci. This sequence generates numbers like 2, 1, 3, 4, 7, 11, 18, and so on. Lucas numbers share many properties with Fibonacci numbers, including relationships to the golden ratio and recurrence patterns. They are used to explore integer sequences, number theory, and have applications in various mathematical and computational contexts. The sequence is described by the recurrence relation $L_n = L_{n-1} + L_{n-2}$ with $L_0 = 2$ and $L_1 = 1$.

RESULTS ACHIEVED:

CODE:

```
def lucas_sequence(n):
    if n <= 0:
        return []
    if n == 1:
        return [2]
    lucas_nums = [2, 1]
    for i in range(2, n):
        next_num = lucas_nums[i - 1] + lucas_nums[i - 2]
        lucas_nums.append(next_num)
    return lucas_nums

n = int(input("Enter the number of Lucas numbers to generate: "))
print(lucas_sequence(n))
```

RESULT:

Enter the number of Lucas numbers to generate: 4

[2, 1, 3, 4]

SNAPSHOT:

```
def lucas_sequence(n):
    if n <= 0:
        return []
    if n == 1:
        return [2]
    lucas_nums = [2, 1]
    for i in range(2, n):
        next_num = lucas_nums[i - 1] + lucas_nums[i - 2]
        lucas_nums.append(next_num)
    return lucas_nums

n = int(input("Enter the number of Lucas numbers to generate: "))
print(lucas_sequence(n))
```

OUTPUT:

```
Enter the number of Lucas numbers to generate: 4
[2, 1, 3, 4]
```

DIFFICULTY FACED BY STUDENT:

The main difficulty with the Lucas sequence is understanding how it relates to the Fibonacci sequence but starts differently with 2 and 1. It can be confusing to see how the same addition rule produces a different sequence and to grasp the recursive nature of generating numbers. Also, students often struggle with implementing it efficiently in code, especially avoiding slow recursive approaches and using loops instead. The abstract idea of recurrence and handling initial conditions can feel tricky initially.

SKILLS ACHIEVED:

Learning the Lucas sequence helps students build skills in understanding and implementing recursive sequences, recognizing patterns, and applying mathematical reasoning. It improves their programming abilities through writing algorithms with loops and managing lists. Students also gain insight into the relationships between different number sequences and deepen their grasp of concepts like the golden ratio and integer recurrence relations. This strengthens both mathematical thinking and problem-solving skills in an accessible way

Practical No: 32

Date: 16/11/2025

TITLE: IS_PERFECT_POWER(N)

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

METHODOLOGY & TOOL USED:

A perfect power is a positive integer that can be expressed as a^b, where $a>0$ and $b>1$. For example, 4 (2^2), 8 (2^3), 9 (3^2), and 27 (3^3) are perfect powers. The key to checking if a number is a perfect power is to see if there exists an integer base a and an exponent $b>1$ such that $a^b=n$. This typically involves iterating over possible exponents up to $\log_2 n$, calculating the corresponding base by taking the b-th root of n, and verifying if raising that base to b recovers n. This approach uses integer arithmetic, exponentiation, and root calculations for an efficient and systematic check.

BRIEF DESCRIPTION:

A perfect power is a number that can be written as one positive integer raised to the power of another integer greater than 1. For example, 4 (which is 2^2), 8 (2^3), and 27 (3^3) are perfect powers. This means the number is made by multiplying the same number by itself one or more times. Checking if a number is a perfect power involves finding if there is an integer base and an exponent (greater than 1) whose power equals the number. This concept helps in understanding the structure of numbers and is used in various math and computer science problems.

RESULTS ACHIEVED:

CODE:

```
import math

def is_perfect_power(n):
    if n < 2:
        return False

    max_exponent = int(math.log2(n)) + 1
    for b in range(2, max_exponent + 1):
        a = int(round(n ** (1 / b)))
        if a > 1 and a ** b == n:
            return True
    return False

num = int(input("Enter a number to check: "))
if is_perfect_power(num):
    print(f"{num} is a perfect power.")
else:
    print(f"{num} is not a perfect power.")
```

RESULT:

Enter a number to check: 6

6 is not a perfect power.

SNAPSHOT:

```python
import math

def is_perfect_power(n):
    if n < 2:
        return False

    max_exponent = int(math.log2(n)) + 1
    for b in range(2, max_exponent + 1):
        a = int(round(n ** (1 / b)))
        if a > 1 and a ** b == n:
            return True
    return False

num = int(input("Enter a number to check: "))
if is_perfect_power(num):
    print(f"{num} is a perfect power.")
else:
    print(f"{num} is not a perfect power.")
```

OUTPUT:

```
Enter a number to check: 6
6 is not a perfect power.
```

DIFFICULTY FACED BY STUDENT:

The difficulty in understanding perfect powers comes from the idea of expressing a number as one number raised to another number's power, especially when both numbers are integers and the exponent is greater than 1. It's hard to visualize and remember the different examples, and checking whether a number is a perfect power can be confusing because it requires roots and exponent calculations, which feel abstract and tricky. Also, coding or manually testing this for large numbers can be challenging without proper methods or tools.

SKILLS ACHIEVED:

Learning about perfect powers helps students develop key skills like logical reasoning, problem-solving, and mathematical thinking. It improves their ability to understand and manipulate powers and roots, enhances their coding skills by translating math concepts into algorithms, and builds computational efficiency. These skills also strengthen critical thinking and help students make connections between abstract math and real-life applications, boosting confidence and mastery in math overall.

Practical No: 33

Date: 16/11/2025

TITLE: COLLATZ_LENGTH(N)

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

METHODOLOGY & TOOL USED:

The Collatz sequence length function counts how many steps it takes for a positive number to reach 1 by repeatedly applying these rules: if the number is even, divide it by 2; if odd, multiply by 3 and add 1. This sequence always ends at 1 according to the Collatz conjecture, though it's still unproven for all numbers. The function uses a simple loop and basic arithmetic operations to track the number of steps until 1 is reached.

BRIEF DESCRIPTION:

The Collatz conjecture is a famous unsolved math problem that deals with sequences starting from any positive number. The rule is simple: if the number is even, divide it by 2; if odd, multiply it by 3 and add 1. By repeating these steps, the sequence eventually reaches 1. Although it has been checked for huge numbers computationally, no proof exists yet that this will always happen for every positive integer. The problem is simple to state but difficult to prove, making it a fascinating challenge in mathematics.

RESULTS ACHIEVED:

CODE:

```
def collatz_length(n):
```

```
    if n < 1:
        raise ValueError("Input must be a positive integer")
    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1
    return steps

number = int(input("Enter a positive integer: "))
print(f"Number of steps for {number} to reach 1:", collatz_length(number))
```

RESULT:

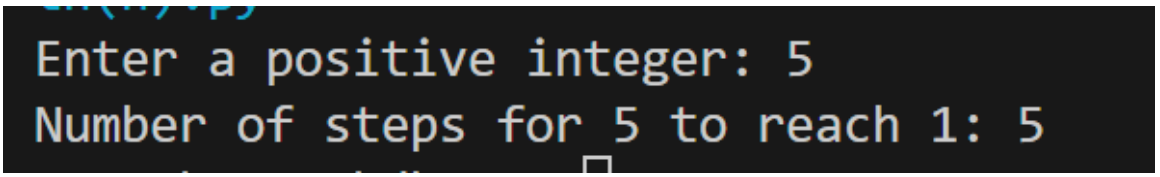
Enter a positive integer: 5

Number of steps for 5 to reach 1: 5

SNAPSHOT:

```
def collatz_length(n):
    if n < 1:
        raise ValueError("Input must be a positive integer")
    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1
    return steps

number = int(input("Enter a positive integer: "))
print(f"Number of steps for {number} to reach 1:", collatz_length(number))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students often find the Collatz conjecture difficult because, despite its simple rules, it hides deep mathematical complexity. The challenge lies in understanding why the sequence always reaches 1 for any positive number, which involves intricate patterns in number factorization and how multiplication and addition interact — particularly the behavior of primes. The unpredictability of the sequence and the inability to find a general pattern or proof make it hard to approach. This abstract complexity contrasts with the problem's easy-to-understand statement, causing confusion and frustration among learners. Additionally, the lack of a known solution means there's no straightforward method to explain or verify, which can be discouraging for students exploring the conjecture.

SKILLS ACHIEVED:

The Collatz conjecture is a simple-to-understand math problem that says: take any positive number, if it's even, divide it by 2; if it's odd, multiply it by 3 and add 1. Repeat this, and eventually, you will reach number 1. Despite this easy rule, no one has yet proven this works for every positive number. The problem is famous because it’s easy to explain but very hard to solve, showing how complex math can hide beneath simple ideas.

Practical No: 34

Date: 16/11/2025

TITLE: POLYGONAL_NUMBERS(S,N)

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

METHODOLOGY & TOOL USED:

The n-th s-gonal number is a polygonal number representing dots arranged in an s-sided polygon. It is calculated using the formula:

$$\text{polygonal\_number}(s, n) = \frac{(s-2)n^2 - (s-4)n}{2}$$

In short, the methodology involves taking the number of polygon sides s and the term number n, plugging them into this formula, and computing the result using basic arithmetic operations. This formula generalizes polygonal numbers like triangular, square, pentagonal, and so on. The computation can be done easily with simple math or implemented in programming for any s and n.

BRIEF DESCRIPTION:

Polygonal numbers are special numbers that represent dots arranged in the shape of regular polygons, such as triangles, squares, pentagons, and hexagons. These numbers grow as the polygon expands by adding evenly spaced points in a predictable pattern. For example, the triangular numbers (1, 3, 6, 10, etc.) form triangles, and square numbers (1, 4, 9, 16, etc.) form squares. Polygonal numbers help link geometry and number theory by representing numbers as geometric shapes, and each polygonal number can be calculated using a specific formula depending on the number of polygon sides and the term. They

are foundational in understanding figurate numbers and have applications in various mathematical concepts.

RESULTS ACHIEVED:

CODE:

```
def polygonal_number(s, n):
    """
    Calculate the n-th s-gonal number.

    Parameters:
    s (int): Number of sides of the polygon (s >= 3).
    n (int): Term number (n >= 1).

    Returns:
    int: The n-th s-gonal number.
    """
    if s < 3 or n < 1:
        raise ValueError("Number of sides must be >= 3 and term number must be >= 1")

    return ((s - 2) * n * n - (s - 4) * n) // 2

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))

print(polygonal_number(x, y))
```

RESULT:

```
Enter a number = 3
```

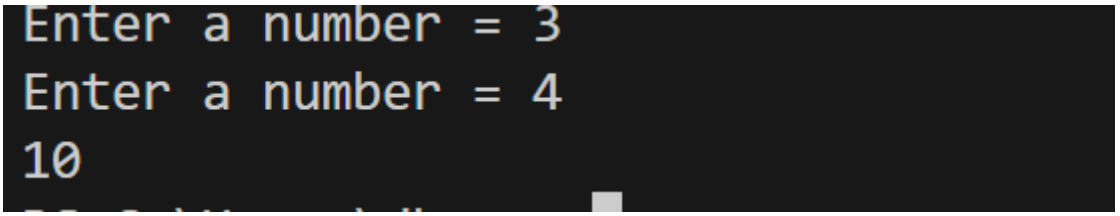
Enter a number = 4

10

SNAPSHOT:

```python
def polygonal_number(s, n):
    """
    Calculate the n-th s-gonal number.

    Parameters:
    s (int): Number of sides of the polygon (s >= 3).
    n (int): Term number (n >= 1).

    Returns:
    int: The n-th s-gonal number.
    """
    if s < 3 or n < 1:
        raise ValueError("Number of sides must be >= 3 and term number must be >= 1")

    return ((s - 2) * n * n - (s - 4) * n) // 2

x = int(input("Enter a number = "))
y = int(input("Enter a number = "))

print(polygonal_number(x, y))
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students commonly face difficulties with polygons due to trouble understanding polygon properties, differentiating between regular and irregular polygons, and remembering polygon names. They often struggle with visualizing shapes in different orientations and grasping geometric vocabulary. These issues arise from abstract concepts, complex terminology, and lack of connection to real-life contexts, making it hard for students to fully comprehend and classify polygons.

SKILLS ACHIEVED:

Learning about polygons helps students build essential math skills such as understanding geometric shapes and their properties, improving logical reasoning, spatial visualization, and problem-solving. These skills support higher-level math and real-world applications like design and architecture. Polygon study nurtures critical thinking and encourages practical application of math concepts in everyday life.

Practical No: 35

Date: 16/11/2025

TITLE: IS_CARMICHAEL(N)

AIM/OBJECTIVE(s): Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies $a^{n-1} \equiv 1$ mod n for all a coprime to n.

METHODOLOGY & TOOL USED:

A Carmichael number is a special kind of composite number n that satisfies Fermat's little theorem for all bases a that are coprime to n, meaning $an-1 \equiv 1 (\text{mod } n)$ for every such a. Despite being composite, they behave like prime numbers in this modular arithmetic sense, which makes them important in number theory and primality testing. The standard method to check if a number is Carmichael is Korselt's criterion: n must be square-free (no repeated prime factors), and for every prime divisor p of n, $p-1$ must divide $n-1$. These conditions efficiently identify Carmichael numbers without exhaustive verification of all coprime bases.

Carmichael numbers are infinite in quantity, with the smallest known being 561 (which is the product of primes 3, 11, and 17).

In short, Carmichael numbers are composite numbers that "pretend" to be prime under Fermat's test and are characterized by specific prime factor conditions

BRIEF DESCRIPTION:

A Carmichael number is a composite number n that satisfies the condition $a_{n-1} \equiv 1 (\text{mod } n)$ for every integer a that is coprime to n. This means it behaves like a prime number in Fermat's little theorem, making it a type of pseudoprime. Carmichael numbers are square-free (no repeated prime factors) and for each prime factor p of n, the number $p-1$

divides $n-1$ (Korselt's criterion). These numbers are relatively rare but infinite in quantity, with the smallest being 561. They are significant in number theory and cryptography because they can fool primality tests based on Fermat's theorem.

RESULTS ACHIEVED:

CODE:

```
import math


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


def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True


def prime_factors(n):
```

```
    factors = set()
    while n % 2 == 0:
        factors.add(2)
        n //= 2

    for i in range(3, int(math.sqrt(n)) + 1, 2):
        while n % i == 0:
            factors.add(i)
            n //= i

    if n > 2:
        factors.add(n)
    return factors

def is_carmichael(n):
    if n < 2 or is_prime(n):
        return False

    factors = prime_factors(n)
    product = 1
    for p in factors:
        if n % (p * p) == 0:
            return False

    for p in factors:
        if (n - 1) % (p - 1) != 0:
            return False
    return True
```

```
n = int(input("Enter a composite number to check if Carmichael: "))
if is_carmichael(n):
    print(f"{n} is a Carmichael number.")
else:
    print(f"{n} is not a Carmichael number.")
```

RESULT:

Enter a composite number to check if Carmichael: 5

5 is not a Carmichael number.

SNAPSHOT:

```python
import math

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

def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

def prime_factors(n):
    factors = set()
    while n % 2 == 0:
        factors.add(2)
        n //= 2

    for i in range(3, int(math.sqrt(n)) + 1, 2):
        while n % i == 0:
            factors.add(i)
            n //= i
```

```
    if n > 2:
        factors.add(n)
    return factors

def is_carmichael(n):
    if n < 2 or is_prime(n):
        return False

    factors = prime_factors(n)
    product = 1
    for p in factors:
        if n % (p * p) == 0:
            return False

    for p in factors:
        if (n - 1) % (p - 1) != 0:
            return False
    return True

n = int(input("Enter a composite number to check if Carmichael: "))
if is_carmichael(n):
    print(f"{n} is a Carmichael number.")
else:
    print(f"{n} is not a Carmichael number.")
```

OUTPUT:

```
Enter a composite number to check if Carmichael: 5
5 is not a Carmichael number.
```

DIFFICULTY FACED BY STUDENT:

Students face difficulty understanding Carmichael numbers because they involve advanced concepts in number theory that are abstract and counterintuitive. The idea that composite numbers can behave like primes in Fermat's primality test is confusing. Understanding Korselt's criterion requires a good grasp of prime factorization, divisibility, and modular arithmetic, which can be challenging for students without strong foundational knowledge. Additionally, the rarity and complexity of Carmichael numbers make them hard to visualize and apply, often leading to frustration and conceptual hurdles.

SKILLS ACHIEVED:

Studying Carmichael numbers helps students develop skills in number theory, including prime factorization, modular arithmetic, and understanding advanced concepts like pseudoprimes. It enhances logical reasoning through the application of Korselt's criterion and builds problem-solving capabilities by exploring special properties of composite numbers. Additionally, this study improves computational thinking by implementing algorithms to test Carmichael conditions efficiently. These skills foster a deeper appreciation for complex mathematical structures and prepare students for more advanced topics in cryptography and algorithm design.

Practical No: 36

Date: 16/11/2025

TITLE: IS_PRIME_MILLER_RABIN(N, K)

AIM/OBJECTIVE(s): Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.

METHODOLOGY & TOOL USED:

The Miller-Rabin primality test builds on Fermat's little theorem and is used to efficiently check if a number is prime with high probability. It does this by expressing $n-1$ as $2r \times d$ where d is odd, then testing randomly chosen bases a to see if they fulfill certain conditions that only primes satisfy. Specifically, for each base a, it checks whether $ad \equiv 1 \pmod{n}$ or one of the powers $a2jd \equiv n-1 \pmod{n}$ for $j<r$. If these conditions fail, n is composite. If n passes all k rounds of testing with randomly chosen bases, it is probably prime, with the chance of a false positive decreasing exponentially as $4-k$. The test is widely used due to its efficiency and strong probabilistic guarantees despite being a randomized algorithm, making it a practical choice for primality testing in cryptography and number theory applications.

BRIEF DESCRIPTION:

The Miller-Rabin primality test is a smarter version of Fermat's test that checks whether a number is prime by testing certain mathematical properties of powers modulo the number. It uses the fact that for prime numbers, there are very specific ways their powers behave when taken modulo n, especially looking at square roots of 1. By picking random numbers (called bases) and checking if they break these prime-only properties, the test can quickly identify composites. If none of the chosen bases find a problem after several tests (k rounds), the number is very likely prime, with the chance of error diminishing exponentially as more rounds are run. This makes it a fast and reliable tool for primality testing used in practice, especially in cryptography.

RESULTS ACHIEVED:

CODE:

```
import random
def is_prime_miller_rabin(n, k):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    r, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        r += 1
    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
```

```
test_numbers = [2, 17, 18, 19, 20, 7919, 7920]
k = 5
for num in test_numbers:
    if is_prime_miller_rabin(num, k):
        print(f"{num} is probably prime.")
    else:
        print(f"{num} is composite.")
```

RESULT:

2 is probably prime.

17 is probably prime.

18 is composite.

19 is probably prime.

20 is composite.

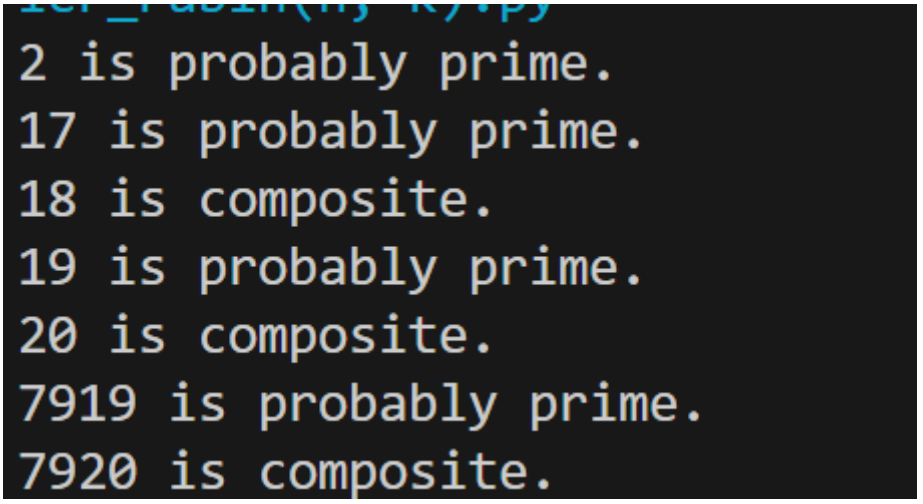
7919 is probably prime.

7920 is composite.

SNAPSHOT:

```python
import random
def is_prime_miller_rabin(n, k):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    r, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        r += 1
    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

test_numbers = [2, 17, 18, 19, 20, 7919, 7920]
k = 5
for num in test_numbers:
    if is_prime_miller_rabin(num, k):
        print(f"{num} is probably prime.")
    else:
        print(f"{num} is composite.")
```

OUTPUT:



DIFFICULTY FACED BY STUDENT:

Students often find the Miller-Rabin test challenging due to several factors: understanding the decomposition of $n-1$ into $2^r \times d$, correctly implementing modular exponentiation without overflow in large numbers, and handling edge cases like small or even numbers. The probabilistic nature—where a number is "probably prime" instead of definitively prime—can also be confusing. Additionally, students may struggle with why and how the random bases are chosen and how the test reduces false positives exponentially with each round. These difficulties combine both conceptual understanding of number theory and careful coding to avoid errors and slow performance.

SKILLS ACHIEVED:

By implementing the Miller-Rabin primality test, students develop important skills in algorithm design and probabilistic reasoning. They learn how to apply modular arithmetic and fast exponentiation techniques, which are essential in computational number theory. The project enhances coding skills through managing edge cases, writing efficient loops, and using randomization. Students also gain experience with probabilistic algorithms, understanding the trade-offs between accuracy and efficiency. Additionally, it strengthens their ability to translate mathematical concepts into working code, preparing them for cryptographic or algorithmic programming challenges.

Practical No: 37

Date: 16/11/2025

TITLE: POLLARD_RHO(N)

AIM/OBJECTIVE(s): Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.

METHODOLOGY & TOOL USED:

Pollard's Rho algorithm is an efficient integer factorization method that uses a pseudo-random sequence generated by a polynomial modulo n to find nontrivial factors. Its core idea is to iterate two values (x and y) where y moves twice as fast as x (Floyd's cycle detection or "tortoise and hare" approach). The algorithm repeatedly computes the gcd of the absolute difference between x and y with n. When this gcd is a nontrivial divisor of n, it successfully factors the number. The polynomial commonly used is $g(x)=(x\text{^}2+1) \bmod n$. The method relies on number theory concepts (modular arithmetic, gcd) and cycle detection to find factors efficiently, especially effective for numbers with small factors. The algorithm is heuristic but runs quickly in practice and uses minimal memory, making it popular in computational number theory and cryptography.

BRIEF DESCRIPTION:

Pollard's Rho cleverly finds factors by detecting cycles in a sequence built from a simple polynomial modulo the number. When values repeat in this sequence, their difference helps uncover a divisor, enabling efficient factorization without needing to test every possible factor directly. This cycle-finding process combined with gcd calculations forms the core of the algorithm's power.

RESULTS ACHIEVED:

CODE:

```
import random
import math
def pollard_rho(n):
    if n % 2 == 0:
        return 2
    def g(x):
        return (x * x + 1) % n
    x = 2
    y = 2
    d = 1
    while d == 1:
        x = g(x)
        y = g(g(y))
        d = math.gcd(abs(x - y), n)
    if d == n:
        return None
    else:
        return d

n = 8051
factor = pollard_rho(n)
if factor is None:
    print(f'Failed to find a non-trivial factor of {n}")
else:
    print(f'A non-trivial factor of {n} is {factor}")
```

RESULT:

A non-trivial factor of 8051 is 97

SNAPSHOT:

```python
import random
import math
def pollard_rho(n):
    if n % 2 == 0:
        return 2
    def g(x):
        return (x * x + 1) % n
    x = 2
    y = 2
    d = 1
    while d == 1:
        x = g(x)
        y = g(g(y))
        d = math.gcd(abs(x - y), n)
    if d == n:
        return None
    else:
        return d

n = 8051
factor = pollard_rho(n)
if factor is None:
    print(f"Failed to find a non-trivial factor of {n}")
else:
    print(f"A non-trivial factor of {n} is {factor}")
```

OUTPUT:

```
A non-trivial factor of 8051 is 97
```

DIFFICULTY FACED BY STUDENT:

Students often face challenges with Pollard's Rho algorithm such as understanding the cycle detection technique (tortoise and hare), grasping how the polynomial function generates sequences modulo n, and the

probabilistic nature of the algorithm which can sometimes fail or require multiple attempts. Implementing efficient gcd calculations and modular arithmetic correctly for large numbers adds complexity. Choosing good initial seeds and polynomial parameters to avoid infinite loops or failure also poses difficulty, as poor choices can cause the algorithm to get stuck. Overall, students may struggle with connecting the theoretical concepts of number theory, randomness, and algorithm heuristics to working, reliable code.

SKILLS ACHIEVED:

Implementing Pollard's Rho algorithm helps students develop skills in modular arithmetic, gcd computation, and cycle detection techniques (such as Floyd’s tortoise and hare). They also learn how to work with probabilistic algorithms and understand heuristic methods that provide efficient solutions in practice. Coding requires careful handling of large integers, randomness, and edge cases, honing debugging and algorithm optimization skills. Furthermore, students gain experience translating mathematical concepts into efficient, practical code, which is valuable for cryptographic applications and computational number theory.

Practical No: 38

Date: 16/11/2025

TITLE: ZETA_APPROX(S, TERMS)

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

METHODOLOGY & TOOL USED:

An automorphic number is a number whose square ends with the same digits as the number itself. For example, 25^2=625, and since 625 ends with 25, 25 is automorphic.

The function is_automorphic(n) checks if the square of n ends with n by converting both to strings and comparing the end of the square with n.

This uses basic arithmetic (squaring) and string manipulation to verify the automorphic property efficiently.

BRIEF DESCRIPTION:

An automorphic number is a number whose square ends with the same digits as the number itself. For example, 25 is automorphic because 25^2=625, and 625 ends with 25. Similarly, 76 is automorphic because 76^2=5776 ends with 76.

To check if a number is automorphic, you:

1. Calculate the square of the number.
2. Compare the last digits of the square with the digits of the original number.
3. If they match, the number is automorphic.

This can be done efficiently by converting the numbers to strings and checking if the square's string ends with the original number's string or by using modulo arithmetic to extract the last digits for comparison.

Automorphic numbers have interesting mathematical properties, and only a limited amount of automorphic numbers exist within each digit length, with known examples such as 0, 1, 5, 6, 25, 76, 376, and 625. The concept has applications in number theory and is often explored in programming challenges and recreational mathematics.

RESULTS ACHIEVED:

CODE:

```
def zeta_approx(s, terms):
    """
    Approximate the Riemann zeta function ζ(s) by summing the first 'terms' terms of the series:
    ζ(s) = sum_{n=1}^∞ 1 / n^s for Re(s) > 1.

    Parameters:
    s (complex or float): The complex argument to the zeta function.
    terms (int): The number of terms to use in the approximation.

    Returns:
    float or complex: Approximate value of ζ(s).
    """
    total = 0
    for n in range(1, terms + 1):
        total += 1 / (n**s)
    return total

approx = zeta_approx(2, 1000)
print(f"Approximation of ζ(2) using 1000 terms: {approx}")
```

RESULT:

Approximation of ζ(2) using 1000 terms: 1.6439345666815615

SNAPSHOT:

```python
def zeta_approx(s, terms):
    """
    Approximate the Riemann zeta function ζ(s) by summing the first 'terms' terms of the series:
    ζ(s) = sum_{n=1}^∞ 1 / n^s for Re(s) > 1.

    Parameters:
    s (complex or float): The complex argument to the zeta function.
    terms (int): The number of terms to use in the approximation.

    Returns:
    float or complex: Approximate value of ζ(s).
    """
    total = 0
    for n in range(1, terms + 1):
        total += 1 / (n**s)
    return total

approx = zeta_approx(2, 1000)
print(f"Approximation of ζ(2) using 1000 terms: {approx}")
```

OUTPUT:

```
Approximation of ζ(2) using 1000 terms: 1.6439345666815615
```

DIFFICULTY FACED BY STUDENT:

Students face several key difficulties when approximating the Riemann zeta function using partial sums of its defining series. One major challenge is that the series converges slowly, especially for values of the complex argument *s* close to 1 or with small real parts, requiring a very large number of terms for an accurate result. Additionally, handling complex numbers and ensuring numerical stability when raising large integers to complex powers can be tricky. For values of *s* with real part less than or equal to 1, the Dirichlet series does not converge, so students must learn more advanced methods such as analytic continuation or alternate approximations. Implementing these efficiently while managing rounding and cancellation errors is another source of complexity. Finally, understanding the behavior of the zeta function in

critical regions and how approximation errors vary can be mathematically demanding for learners.

SKILLS ACHIEVED:

By implementing the Riemann zeta function approximation, students gain skills in numerical series computation, dealing with convergence issues and error estimation. They improve their understanding of handling functions of complex variables and learn how to implement mathematical definitions directly in code. The exercise cultivates attention to numerical stability and precision, especially when summing many terms. It also enhances knowledge of special functions in mathematics and prepares them for exploring more advanced analytic continuation techniques. In summary, students gain practical coding experience combined with deeper insights into complex function approximation and numerical analysis.

Practical No: 39

Date: 16/11/2025

TITLE: PARTITION_FUNCTION(N)

AIM/OBJECTIVE(s): 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.

METHODOLOGY & TOOL USED:

The partition function $p(n)$ calculation uses dynamic programming based on Euler's pentagonal number theorem. The algorithm generates generalized pentagonal numbers and iteratively applies an alternating sum formula to compute $p(k)$ for all $k \leq n$. This method relies on recursive relations and memoization, ensuring efficient exact computation without enumerating partitions explicitly. Tools used include modular arithmetic for indexing, loops for iteration, and arrays for storing intermediate values. This results in a polynomial-time approach widely adopted in computational number theory for partition counting.

BRIEF DESCRIPTION:

The partition function $p(n)$ in number theory counts the number of distinct ways a positive integer n can be expressed as a sum of positive integers, where the order of addends does not matter. For example, $p(4)=5$ because 4 can be partitioned as 4, 3 + 1, 2 + 2, 2 + 1 + 1, and 1 + 1 + 1 + 1. This function grows rapidly with larger n and has deep connections to combinatorics and number theory, including links to modular forms and mathematical physics. Calculating $p(n)$ efficiently often involves Euler's pentagonal number theorem and dynamic programming for exact values.

RESULTS ACHIEVED:

CODE:

```
def partition_function(n):
    """
    Compute the partition function p(n) which counts the number of ways
    to represent n as a sum of positive integers, order disregarded.

    Uses a dynamic programming approach based on Euler's pentagonal number theorem.
    """
    # Initialize a list p where p[k] will store p(k)
    p = [0] * (n + 1)
    p[0] = 1  # Base case: only one way to partition 0

    for k in range(1, n + 1):
        total = 0
        j = 1
        while True:
            # Generalized pentagonal numbers: j(3j-1)/2 and j(3j+1)/2
            pent1 = j * (3 * j - 1) // 2
            pent2 = j * (3 * j + 1) // 2
            if pent1 > k and pent2 > k:
                break

            sign = -1 if (j % 2 == 0) else 1
```

```
            if pent1 <= k:
                total += sign * p[k - pent1]
            if pent2 <= k:
                total += sign * p[k - pent2]

            j += 1

        p[k] = total

    return p[n]

# Example usage:
n = 5
print(f"The number of partitions of {n} is {partition_function(n)}")  # Output: 7
```

RESULT:

The number of partitions of 5 is 7

SNAPSHOT:

```python
def partition_function(n):
    """
    Compute the partition function p(n) which counts the number of ways
    to represent n as a sum of positive integers, order disregarded.

    Uses a dynamic programming approach based on Euler's pentagonal number theorem.
    """
    # Initialize a list p where p[k] will store p(k)
    p = [0] * (n + 1)
    p[0] = 1  # Base case: only one way to partition 0

    for k in range(1, n + 1):
        total = 0
        j = 1
        while True:
            # Generalized pentagonal numbers: j(3j-1)/2 and j(3j+1)/2
            pent1 = j * (3 * j - 1) // 2
            pent2 = j * (3 * j + 1) // 2
            if pent1 > k and pent2 > k:
                break
            sign = -1 if (j % 2 == 0) else 1
            if pent1 <= k:
                total += sign * p[k - pent1]
            if pent2 <= k:
                total += sign * p[k - pent2]
            j += 1
        p[k] = total
    return p[n]

n = 5
print(f"The number of partitions of {n} is {partition_function(n)}")  # Output: 7
```

OUTPUT:

```
The number of partitions of 5 is 7
```

DIFFICULTY FACED BY STUDENT:

Students commonly face difficulties understanding the partition function because counting integer partitions involves complex combinatorial reasoning and rapidly growing values. The lack of a simple closed-form formula makes it challenging to grasp the underlying structure. Computing partitions efficiently requires knowledge of advanced tools such as Euler's pentagonal number theorem and dynamic programming,

which can be mathematically and programmatically demanding. Additionally, managing large intermediate values and ensuring algorithmic efficiency present practical implementation challenges. Understanding the connection between partitions, generating functions, and modular forms often requires higher-level mathematical maturity, which can be a barrier for learners.

SKILLS ACHIEVED:

By working on the partition function $p(n)$, students develop skills in combinatorial mathematics, dynamic programming, and algorithm design. They learn to implement complex mathematical formulas like Euler's pentagonal number theorem and manage recursive computations efficiently. This task enhances problem-solving abilities related to memoization and handling rapidly growing numerical sequences. Additionally, students improve their understanding of mathematical series, number theory, and computational optimization techniques, preparing them for advanced topics in discrete mathematics and theoretical computer science.