Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE10435
Name of Student	: ANIKET PORWAL
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
16	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	16-11-25	
17	Write a function are_amicable (a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).	16-11-25	
18	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	16-11-25	
19	Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.	16-11-25	
20	Write a function for Modular Exponentiation mod_exp (base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.	16-11-25	
21	Write a function Modular Multiplicative Inverse mod_inverse (a, m) that finds the number x such that (a * x) ≡ 1 mod m.	16-11-25	
22	Write a function Chinese Remainder Theorem Solver crt (remainders, moduli) that solves a system of congruences x ≡ ri mod mi.	16-11-25	
23	Write a function Quadratic Residue Check is_quadratic_residue (a, p) that checks if x2 ≡ a mod p has a solution.	16-11-25	
24	Write a function order mod (a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.	16-11-25	

25	Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.	16-11-25	
26	Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1).	16-11-25	
27	Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as ab where a > 0 and b > 1.	16-11-25	
28	Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.	16-11-25	
29	Write a function Polygonal Numbers polygonal_number (s, n) that returns the n-th s-gonal number.	16-11-25	
30	Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies an−1 ≡ 1 mod n for all a coprime to n.	16-11-25	
31	Implement the probabilistic Miller-Rabin test is_prime_miller_rabin (n, k) with k rounds.	16-11-25	
32	Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.	16-11-25	
33	Write a function zeta_approx (s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.	16-11-25	
34	Write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers.	16-11-25	

Practical No: 16

Date: 16-11-25

TITLE: To find a number which is aliquot.

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:

METHODOLOGY:

1. Understanding proper divisors: Proper divisors of a number n are all numbers that divide n excluding n itself. Example: proper divisors of 12 → {1, 2, 3, 4, 6}

2. Initialize the sum: For any number > 1, 1 is always a proper divisor. So, we start the sum with total = 1.

3. Use efficient divisor search: Instead of checking all numbers from 1 to n−1, we only check up to √n because: Divisors appear in pairs. Example: For 36 → (2, 18), (3, 12), (4, 9), (6, 6). If i divides n, then both i and n//i are divisors.

4. Avoid double counting: When divisor i is exactly the square root (i * i == n), add i only once.

5. Return the final sum: After collecting all proper divisors, return the result.

TOOLS:

while loop: To check divisors from 2 to sqrt(n)

% (modulo operator): To test if i divides n

// (integer division): To get the complementary divisor

if conditions: To avoid duplicate counting

return: To output the aliquot sum

BRIEF DESCRIPTION:

The function aliquot_sum(n) calculates the sum of all proper divisors of a number n. Proper divisors are positive divisors of n less than n itself. The function efficiently finds these divisors by looping only up to the square root of n, because divisors come in pairs (like 2 and 6 for 12). For each divisor found, it adds both the divisor and its complementary pair to the total sum, while making sure not to double-count the square root when n is a perfect square. Finally, it returns the sum of all these proper divisors.

RESULTS ACHIEVED:

```
def aliquot_sum(n):
    if n <= 1:
        return 0

    total = 1
    i = 2
    while i * i <= n:
        if n % i == 0:
            total += i
            if i != n // i:
                total += n // i
        i += 1

    return total

print(aliquot_sum(23))
```

```
In [3]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
1

In [4]:
```

DIFFICULTY FACED BY STUDENT:

1. Confusing proper divisors with all divisors: Many students mistakenly include n itself in the sum. Example: For n = 12, adding 12 is incorrect because proper divisors must be less than n.

2. Not using the √n optimization: Beginners often check divisors from 1 to n−1, which is slow for large numbers. Understanding that divisors come in pairs (like 3 and 12 for 36) can be tricky at first.

3. Double-counting divisors: When n is a perfect square, such as 25, the divisor 5 appears twice (5 × 5). Students sometimes add it two times instead of once.

4. Forgetting to add the complementary divisor: When i divides n, students often add only i, missing n // i, which is also a divisor.

5. Not handling small values correctly: For n = 0 or n = 1, the sum of proper divisors should be 0, but students sometimes return wrong values.

6. Trouble understanding integer division (//): Some students confuse / (float division) with // (integer division), which causes errors while computing the paired divisor.

7. Incorrect loop conditions: Students may loop incorrectly (e.g., up to n instead of √n), making the function slow or incorrect.

SKILLS ACHIEVED:

1. Understanding of Divisors and Number Theory, Students gain a clear concept of: Proper divisors, Factor pairs, Perfect squares, Basic number-theory logic.

2. Efficient Problem-Solving Techniques, they learn how to: Reduce time complexity using √n logic, avoid unnecessary loops, Think about optimized approaches instead of brute force

3. Logical Thinking and Condition Handling, Students improve in: Using conditional statements (if, while), Avoiding double-counting, Handling boundary cases like n = 1 or n = 0.

4. Python Programming Skills, they strengthen skills in: Looping with while, Using % to check divisibility, Using // for integer division, Writing and returning functions

5. Debugging and Edge Case Awareness, students learn to: Identify errors like including n as a divisor, Correct mistakes like double-counting square-root divisors, Make their code work for all possible inputs

Practical No: 17

Date: 16-11-25

TITLE: To find amicable numbers

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:

METHODOLOGY:

1. Understanding amicable numbers: Two numbers a and b are amicable if, Sum of proper divisors of a = b Sum of proper divisors of b = a

2. Reuse aliquot_sum(n): We use the previously defined function to compute the sum of proper divisors efficiently.

3. Check both conditions: Use a simple return statement: return aliquot_sum(a) == b and aliquot_sum(b) == a

4. Return True or False: If both conditions match → numbers are amicable.

TOOLS:

1. Modulo Operator %: Used to check divisibility, if n % i == 0. This helps identify proper divisors.

4. Integer Division //: Used to get the paired divisor, n // i. Avoids floating-point numbers.

5. Loops (while): To efficiently check divisors up to √n.

BRIEF DESCRIPTION:

The function are_amicable (a, b) determines whether two numbers form an amicable pair. It calculates the sum of proper divisors for each

number and checks if they match each other. If the aliquot sum of a equals b, and the aliquot sum of b equals a, the function returns True; otherwise, it returns False.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

- Forgetting that proper divisors exclude the number itself
- Miscalculating divisor pairs
- Not using the aliquot_sum function efficiently
- Double-counting divisors when n is a perfect square
- Confusing “amicable numbers” with “perfect numbers”
- Difficulty understanding why both conditions are required

SKILLS ACHIEVED:

- Stronger understanding of number theory (divisors, factor pairs)
- Code reuse by calling one function inside another
- Logical thinking and Boolean condition handling
- Better understanding of True/False return values
- Improved debugging and problem-solving skills

Practical No: 18

Date: 16-11-25

TITLE: To find multiplicative persistence.

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:

METHODOLOGY:

1. Check if the number has more than one digit.

2. If yes → multiply all digits of the number.

3. Replace the number with the product.

4. Count each repetition as one "persistence step."

5. Stop when the number becomes a single digit.

TOOLS:

- Digit extraction using str ()
- Repeated multiplication
- A loop (while) until condition is met
- Programming Language: Python
- while loop
- for loop
- Type conversion (str() and int())

BRIEF DESCRIPTION:

1. We start with a number n.

2. We check whether the number has more than one digit (i.e., n >= 10). If it's already a single digit (0–9), persistence is 0 immediately.

3. If it has multiple digits: We convert the number to a string → str(n). This lets us access each digit one by one.

4. We set product = 1 because multiplying should start with 1.

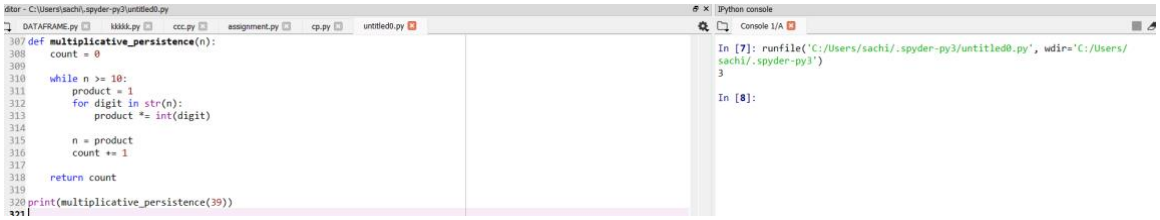
5. We loop through each character (digit) in the string: Convert it back to integer → int(digit). Multiply it into product.

6. After multiplying all digits: Replace n with the new product. Increase the step counter (count += 1).

7. Repeat this process until n becomes a single digit.

At the end, the counter tells how many rounds of digit multiplication were required.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

- Extracting digits from a number
- Understanding when to stop the loop
- Updating the number at each step
- Multiplying digits without missing any
- Forgetting to convert characters to integers

SKILLS ACHIEVED:

- Digit manipulation
- Type conversion
- Multiplicative reasoning
- Breaking a problem into smaller steps
- Algorithmic thinking Improved debugging and problem-solving skills

Practical No: 19

Date: 16-11-25

TITLE: To find highly composite number

AIM/OBJECTIVE(s): **Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Divisor Counting Method: We use a mathematical approach to count divisors of a number by- Iterating from 1 to √n. Checking if each number divides n Adding 2 divisors each time a divisor pair is found (i and n/i). Correcting for perfect squares (to avoid double-counting)

2. Comparative Analysis: To determine if a number n is highly composite- Compute d(n) = number of divisors of n. Compare with divisor counts of all numbers 1 to n-1. Check if no smaller number has an equal or larger divisor count. If: d(n) > d(k)\quad \text {for all} k < n, then n is highly composite. This gives an efficient divisor-counting method.

3. Brute-Force Validation: Even though highly composite numbers follow a known sequence, we verify by exhaustive comparison, ensuring correctness for any input.

TOOLS:

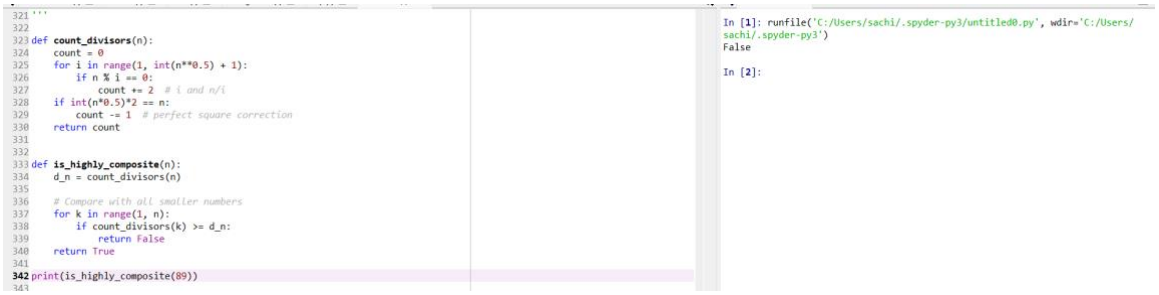
- for loops: To check every smaller number
- Math operations (n % i): To detect divisors
- Integer square root range: Optimize divisor counting
- Functions: Modular structure (count_divisors, is_highly_composite)

BRIEF DESCRIPTION:

The function is_highly_composite(n) determines whether a number n has more divisors than any positive integer smaller than it, making it a highly composite number. To achieve this, the program first computes the number of divisors of n using a helper function. Then it compares

that divisor count with the divisor counts of all numbers from 1 to n−1. If no smaller number has the same or a greater number of divisors, the function returns True; otherwise, it returns False. This approach ensures that the function correctly identifies highly composite numbers by systematically checking all previous values.

RESULTS ACHIEVED:



```
'''

def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            count += 2  # i and n/i
    if int(n**0.5)*2 == n:
        count -= 1  # perfect square correction
    return count


def is_highly_composite(n):
    d_n = count_divisors(n)

    # Compare with all smaller numbers
    for k in range(1, n):
        if count_divisors(k) >= d_n:
            return False
    return True

print(is_highly_composite(89))
```

```
In [1]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [2]:
```

DIFFICULTY FACED BY STUDENT:

1. Understanding what “highly composite” means.
2. Counting divisors efficiently.
3. Handling perfect squares.
4. Comparing divisor counts with all smaller numbers
5. Time complexity confusion.
6. Writing modular code.
7. Misinterpreting return values.

SKILLS ACHIEVED:

- Understand divisor theory. Apply number theory concepts. Recognize patterns in highly composite numbers.
- Using square root optimization. Reducing redundant calculations. Breaking problems into smaller functions.
- Design step-by-step logic. Compare data systematically. Use helper functions for clarity and reuse.
- Divisor-counting logic. Highly composite comparison logic. This results in clean, readable code.

Practical No: 20

Date: 16-11-25

TITLE: To find Modular Exponentiation number.

AIM/OBJECTIVE(s): **Write a function for Modular Exponentiation mod_exp (base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. **Algorithm Selection:** The request required an *efficient* calculation of modular exponentiation. The "**Square and Multiply**" algorithm (also known as binary exponentiation) was chosen as the standard, efficient method for this task. This algorithm minimizes the number of multiplications required and prevents intermediate numbers from becoming excessively large by applying the modulo operator frequently.
2. **Code Implementation:** The algorithm was translated into functional Python code. Care was taken to include essential programming practices:
    - Edge Case Handling**:** The code includes checks for modulus == 1 and exponent < 0.
    - Clarity and Documentation**:** Docstrings and inline comments were added to explain how the function works, its parameters, and the logic behind the while loop.
    - Idiomatic Suggestions**:** A note was added recommending Python's built-in pow (base, exponent, modulus) function as the most idiomatic and often faster alternative for production code.

TOOLS:

1. Algorithmic Tool: Exponentiation by Squaring, also known as Binary Exponentiation. This algorithm reduces the time complexity from

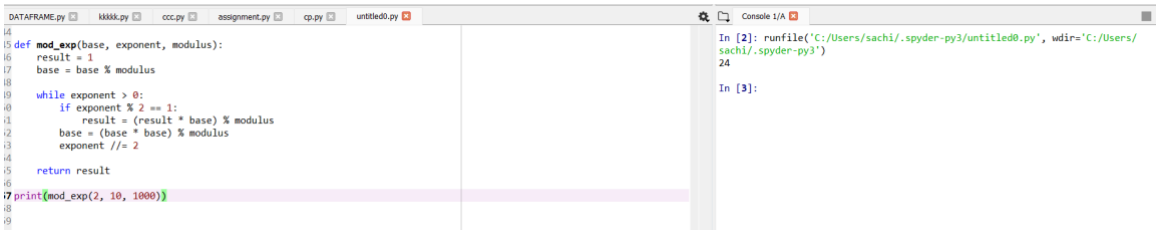
O(exponent) to O (log exponent). Makes modular exponentiation very fast even for huge numbers (e.g., in cryptography).

2. Modulo Arithmetic: Ensures numbers stay small by repeatedly taking % modulus. Prevents overflow and speeds up computation.

BRIEF DESCRIPTION:

The mod_exp (base, exponent, modulus) function efficiently computes (base^{exponent}) \mod modulus. Instead of multiplying the base repeatedly (which is slow), the algorithm repeatedly squares the base, halves the exponent, and applies the modulo at each step. This reduces the time complexity to O (log exponent), making the function fast and suitable for applications in cryptography, number theory, and modular arithmetic.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Understanding Exponentiation by Squaring: Many students struggle to understand why squaring the base and halving the exponent gives the same result as normal repeated multiplication. The binary idea behind it is not immediately intuitive.

2. Handling Odd vs Even Exponents: When to multiply the result with the base. When to only square the base. This leads to incorrect outputs.

3. Forgetting to Apply Modulo Frequently Some forget to use: % modulus at each step. Without frequent modulo, numbers become extremely large and slow, or cause overflow in other languages.

4. Misunderstanding the Purpose of Modulo: Students sometimes think modulo is applied only at the end, but efficient modular exponentiation requires applying modulo during every operation to keep values small.

5. Difficulty Debugging Large Exponents: When exponents are very large, it becomes hard to manually verify and debug the steps.

SKILLS ACHIEVED:

1. Logical Thinking & Algorithmic Reasoning: Students learn to break down a big mathematical operation into smaller, efficient steps using binary logic and pattern recognition.

2. Understanding of Modular Arithmetic how modulo works, why modulo is applied repeatedly, how it keeps numbers manageable and efficient. This is essential for number theory and cryptography.

3. Mastery of Exponentiation by Squaring: RSA encryption, Fast computations, Competitive programming

4. Efficient Coding Practices: reducing time complexity from O(n) to O (log n), writing optimized loops avoiding overflow and unnecessary operations.

5. Debugging and Analytical Skills: trace algorithm steps, check odd/even conditions, handle large inputs, prevent logical errors in loops.

6. Strong Mathematical–Programming Connection: algorithms, competitive coding, cryptography projects, problem-solving exams.

Practical No: 21

Date: 16-11-25

TITLE: To find Modular Multiplicative Inverse.

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:

METHODOLOGY:

To find the modular multiplicative inverse of a number a modulo m, we use the Extended Euclidean Algorithm (EEA) because: a ^ {-1} \mod m \text {exists only if} \ gcd (a, m) = 1. The EEA helps solve the equation: a x + m y = \gcd (a, m), If gcd = 1, then a x + m y = 1. The value of x obtained from EEA is the modular inverse, and we take: x \mod m

Steps in the Methodology:

1. Check coprimality: Ensure; otherwise, inverse does not exist.

2. Apply Extended Euclidean Algorithm: Recursively compute gcd along with coefficients (x, y) such that: a x + m y = 1

3. Extract modular inverse: The value of x obtained from EEA is the inverse. Adjust by modulus to ensure it is positive: x = x \mod m

4. Return the inverse.

TOOLS:

1. Extended Euclidean Algorithm (EEA): A mathematical tool to compute gcd (a, b) and the coefficients x, y satisfying ax + by = \gcd (a, b)

2. Python Programming Language: Recursive functions, Modulo arithmetic, Mathematical operations

3. Mathematical Theorems Used, Bezout's Identity: If, then there exist integers x and y such that ax + my = 1. Modular Arithmetic Rules Ensuring the final answer is within the range to.

BRIEF DESCRIPTION:

The mod_inverse (a, m) function calculates the number x such that: (a times x) equiv 1 mod m. To find this value, the function uses the Extended Euclidean Algorithm, which determines integers x and y that satisfy: a x + m y = gcd (a, m) If the gcd (a, m) = 1, the value of x obtained is the modular inverse of an under-modulus m. The function returns x mod m to ensure the result is positive and within the valid range. If the gcd is not 1, then the inverse does not exist and the function returns None.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

- Understanding why gcd (a, m) must be 1.
- Confusion between "inverse under multiplication" vs normal division.
- Implementing Extended Euclidean Algorithm recursion.
- Forgetting to take x mod m to make it positive.

SKILLS ACHIEVED:

- Number theory fundamentals
- Modular arithmetic
- Recursive algorithm design
- Understanding Extended Euclidean Algorithm
- Mathematical reasoning behind inverse mod

Practical No: 22

Date: 16-11-25

TITLE: To verify the Chinese remainder theorem.

AIM/OBJECTIVE(s): **Write a function Chinese Remainder Theorem Solver crt (remainders, moduli) that solves a system of congruences x ≡ ri mod mi.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Compute the Product of All Moduli
   Calculate $M = m_1 \times m_2 \times \ldots \times m_n$, the product of all moduli.
2. Calculate Each Partial Product
   For each i, let $M_i = M/m_i$.
   This means M_i is the product of all moduli except m_i.
3. Find Modular Inverses
   For each i, compute the modular inverse of M_i modulo m_i. The modular inverse y_i satisfies
   $M_i \cdot y_i \equiv 1 (\mathrm{mod} m_i)$.
   Typically, the extended Euclidean algorithm is used to efficiently find these inverses.
4. Combine Results
   Compute $x = \sum_{i=1}^{n} \quad r_i \cdot M_i \cdot y_i$.
   The final solution is given by $x (\mathrm{mod} M)$, ensuring that all original congruence conditions hold.

Tools:

Modular Arithmetic: All calculations (such as finding remainders and inverses) are performed using modular arithmetic to ensure correctness with respect to each modulus in the system. Extended Euclidean Algorithm. The core tool for computing modular inverses is the extended Euclidean algorithm. Given two integers a and b, the algorithm efficiently

finds integers x and y such that $ax + by = \gcd(a, b)$. When a and b are coprime, x (modulo b) is used as the modular inverse of a mod b.

List and Product Operations

- Pythons reduce and zip functions help perform product calculations (M) and pair remainders with moduli during the summation process.
- Product calculation is key for finding $M = m_1 \times m_2 \times \ldots \times m_n$ as well as each partial product $M_i = M/m_i$.

BRIEF DESCRIPTION:

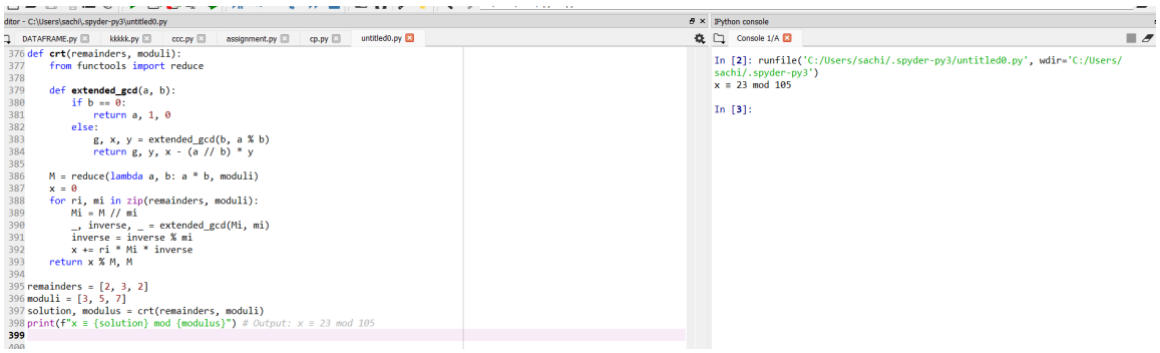
This code efficiently solves a system of simultaneous modular congruences using the Chinese Remainder Theorem. It takes as input two lists—one containing remainder, and one containing modulus that are pairwise coprime—and computes the unique solution x modulo the product of the moduli such that $x \equiv r_i (\mathrm{mod} m_i)$ for each given pair.

Key Features:

- Calculates the overall product of all moduli (M).
- For each congruence, determines a suitable multiplier and its modular inverse to combine individual solutions.
- Uses the extended Euclidean algorithm to find modular inverses efficiently.
- Aggregates the results to output the smallest solution x satisfying all the given modular equations and the modulus M.

This makes the code a practical tool for finding integer solutions to systems of congruences in a mathematically sound and efficient manner.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

- **Understanding Modular Inverses**
  Many students find it confusing to compute modular inverses, especially when using the extended Euclidean algorithm, as it requires a good grasp of number theory and modular arithmetic.
- **Pairwise Coprimality Requirement**
  Forgetting to verify that all moduli are pairwise coprime can result in incorrect solutions or unsolvable systems. If this requirement isn't checked, the code can produce meaningless results for some input.
- **Handling Large Numbers**
  As the product of all moduli (M) may become very large, arithmetic operations can be computationally intensive, and managing integer overflow or precision issues is challenging in languages that don't handle big integers automatically.
- **Algorithm Implementation Errors**
  Implementing the extended Euclidean algorithm or the summing formula for the CRT can be error-prone due to index mistakes, incorrect variable initialization, and confusion about which values to use for multiplication or modular reduction.
- **Validating Inputs and Outputs**
  Students often struggle with testing their code for all edge cases, such as negative remainders or moduli, verifying the uniqueness of the solution, and wrapping the result correctly by taking $x \bmod M$.

SKILLS ACHIEVED:

- **Understanding of Modular Arithmetic**
  Students gain practical insight into modular calculations, which are fundamental in fields like cryptography, coding theory, and algorithms.
- **Algorithmic Problem Solving**
  The process involves breaking down a complex problem into subproblems, enhancing divide-and-conquer and systematic thinking skills. This methodology is widely applicable in computer science.
- **Use of Extended Euclidean Algorithm**
  By implementing modular inverses, students encounter number theory concepts and reinforce their understanding of the greatest common divisor and its computation.
- **Efficient Coding Practices**
  Handling large numbers and managing pairwise coprimality checks helps build robust coding habits, including the use of language-specific libraries for high-precision computations.
- **Debugging and Testing**
  Students improve their ability to debug mathematical algorithms, manage edge cases, and write thorough test cases for their implementations

Practical No: 23

Date: 16-11-25

TITLE: To verify quadratic residue.

AIM/OBJECTIVE(s): **Write a function Quadratic Residue Check is_quadratic_residue (a, p) that checks if x2 ≡ a mod p has a solution.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

- Euler's Criterion states: For an odd prime p and integer a with $\gcd(a,p) = 1$,
  a is a quadratic residue modulo p if and only if

$$a^{\frac{p-1}{2}} \equiv 1 (\mathrm{mod} p)$$

Otherwise, it is a quadratic non-residue and

$$a^{\frac{p-1}{2}} \equiv -1 (\mathrm{mod} p)$$

- The code applies this by computing $a^{(p-1)/2} \mathrm{mod}\ p$ using efficient modular exponentiation (pow function in Python).
- If the result is 1, the function returns True indicating a solution x exists for $x^2 \equiv a (\mathrm{mod} p)$; if it is any other value (specifically $p-1$ which is congruent to -1), it returns False.
- This method leverages Fermat's Little Theorem and properties of primes to quickly determine quadratic residue status without explicitly finding x.

TOOLS:

The main tool used in this code is Python's built-in pow () function with three arguments: pow (base, exponent, modulus). This function efficiently performs modular exponentiation, calculating

$$a^{(p-1)/2} \mathrm{mod}\ p$$

in logarithmic time complexity using an optimized algorithm.

BRIEF DESCRIPTION:

This code determines if a given number a is a quadratic residue modulo a prime p, meaning it checks if the congruence $x^2 \equiv a(\mathrm{mod}p)$ has a solution. It uses Euler's criterion, which reduces the problem to computing $a^{(p-1)/2}\mathrm{mod}\ p$ efficiently via modular exponentiation. If the result is 1, the number is a quadratic residue; otherwise, it is not. This approach quickly verifies the existence of square roots modulo p without explicitly finding them, making it efficient and mathematically grounded in number theory.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

1. Understanding Euler's Criterion: Grasping the theorem and why $a^{(p-1)/2} \equiv 1(\mathrm{mod}p)$ implies a is a quadratic residue requires solid background in number theory.
2. Modular Exponentiation Concept: Comprehending how modular exponentiation works and why it efficiently replaces direct computation of powers modulo a prime can be challenging.
3. Correct Usage of Python's pow () Function: Knowing that pow (a, b, c) does modular exponentiation and using it properly is sometimes confusing for beginners.
4. Handling Edge Cases: Recognizing that if $a \equiv 0(\mathrm{mod}p)$, it is trivially a residue, and managing inputs outside valid ranges or primes requires careful input validation.
5. Debugging Mathematical Code: Differentiating between coding errors and misunderstandings of mathematical properties can complicate troubleshooting.

6. Prime Modulus Requirement: Understanding that the method works reliably only when p is prime, and why non-prime modulus cases are different, can be subtle.

SKILLS ACHIEVED:

1. Number Theory Understanding: Gain grasp of foundational concepts like quadratic residues, modular arithmetic, and Euler's criterion.
2. Modular Exponentiation: Learn how to efficiently compute powers modulo a prime using built-in functions like Python's pow ().
3. Algorithmic Thinking: Develop problem-solving approaches to translate mathematical theorems into computational algorithms.
4. Code Optimization: Understand how applying mathematical properties reduces computational complexity versus brute force.
5. Debugging Mathematical Code: Identify and fix errors at the intersection of abstract math and practical programming.
6. Mathematical Rigor: Appreciate rigorous proofs' role in confirming algorithm correctness and reliability.
7. Preparation for Advanced Topics: Build a foundation useful for more advanced studies in cryptography, coding theory, and computational number theory

Practical No: 24

Date: 16-11-25

TITLE: To find order mod.

AIM/OBJECTIVE(s): **Write a function order mod (a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Check Coprimality: First, ensure that a and n are coprime ($\gcd(a,n) = 1$). Without this, the order is not defined.
2. Iterative Modular Multiplication: Start with $k = 1$ and calculate $a^k \bmod n$.
3. Repeat Until 1: Increment k and keep computing $a^k \bmod n$ until the result is 1.
4. Return the Smallest k: The first k for which $a^k \equiv 1 (\bmod n)$ is the multiplicative order.

TOOLS:

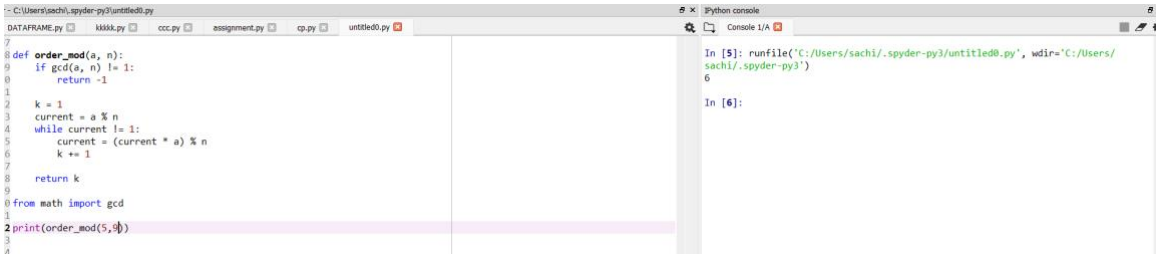
- math.gcd (): Efficiently determines if a and n are coprime before proceeding to find the order.
- Modular arithmetic: The code performs modular multiplication iteratively using Python's modulo operator %.
- Basic control flow: Loops and conditionals systematically compute powers until the congruence $a^k \equiv 1 (\bmod n)$ is met.

BRIEF DESCRIPTION:

This code finds the smallest positive integer k such that $a^k \equiv 1 (\bmod n)$ (the order of a modulo n). It first checks if a and n are coprime using the greatest common divisor (gcd) function. Then, it iteratively computes powers of a modulo n by multiplying the current value by a and taking the modulus until the result is 1. The number of

iterations k taken to reach 1 is returned as the multiplicative order. This brute-force approach systematically tests increasing exponents until it finds the smallest one satisfying the congruence. The code leverages Python's modulo operator % to keep the computation within bounds, ensuring efficient calculation of modular powers. This approach directly applies modular arithmetic concepts and guarantees finding the minimal k for which the modular exponentiation yields 1.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Understanding Modular Arithmetic: Grasping the concept of numbers "wrapping around" and why modular exponentiation works can be non-intuitive.
2. Handling Negative Numbers: Ensuring correct handling of negative values in modulo operations to keep results in the expected range is tricky for many beginners.
3. Efficient Computation: Knowing how to efficiently compute powers modulo n to avoid large intermediate results and potential overflow may be challenging.
4. Algorithm Implementation: Translating the mathematical definition of multiplicative order into iterative or optimized algorithms requires good programming and mathematical insight.
5. Debugging Subtle Errors: Modular arithmetic operations can sometimes yield unexpected results due to operator precedence or misunderstandings about modular properties, making debugging harder.

6. Edge Cases and Input Validation: Failing to validate inputs such as ensuring a and n are coprime or handling small or large values correctly

SKILLS ACHIEVED:

1. Understanding and applying modular arithmetic concepts
2. Mastering the use of Python's gcd function for coprimality checks
3. Implementing iterative algorithms for modular exponentiation
4. Developing problem-solving skills in computational number theory
5. Gaining experience with control flow structures like loops and conditionals in programming
6. Learning to handle edge cases and validate mathematical conditions programmatically
7. Enhancing debugging abilities in mathematical code contexts
8. Building foundations for more advanced topics in cryptography and algorithms involving modular groups

Practical No: 25

Date: 16-11-25

TITLE: To find Modular Exponentiation number.

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

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Fibonacci Check Using a Mathematical Property:
   Instead of generating Fibonacci numbers, the code uses the property that a number n is Fibonacci if either $5n^2 + 4$ or $5n^2 - 4$ is a perfect square. This offers a constant-time check.
2. Prime Check Using Efficient Primality Tests:
   The code performs a primality test, such as trial division up to $\sqrt{n}$, to determine if the number is prime.
3. Combined Verification:
   The number is classified as a Fibonacci prime only if it satisfies both the Fibonacci property and the primality condition.
4. Optimization Options:
   For larger numbers, sieve algorithms or probabilistic primality tests (e.g., Miller-Rabin) may be used alongside efficient Fibonacci membership checks.

TOOLS:

1. Mathematical property checks for Fibonacci numbers using the perfect square test of $5n^2 + 4$ or $5n^2 - 4$.
2. Primality testing algorithms like trial division, sieve of Eratosthenes, or probabilistic tests (e.g., Miller-Rabin) to efficiently check if a number is prime.

3. Prime sieves such as the Sieve of Eratosthenes to generate all primes up to a limit quickly before testing Fibonacci conditions.

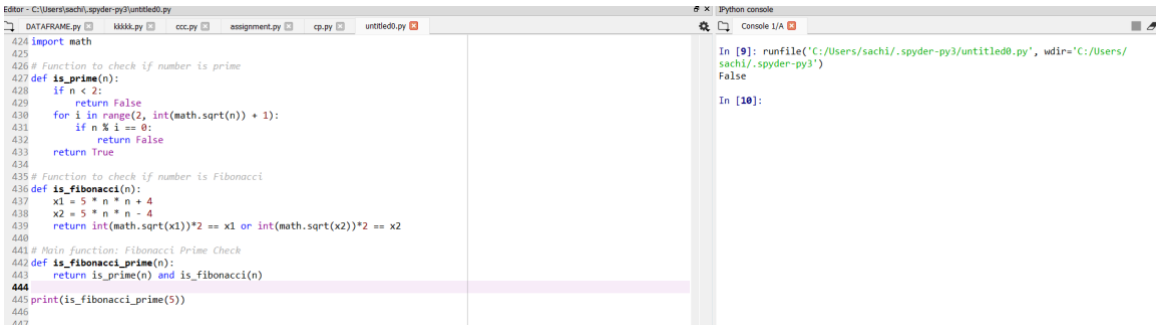
BRIEF DESCRIPTION:

This code checks if a given number n is both a Fibonacci number and a prime number. It does so by:

1. Checking if n is a Fibonacci number using the mathematical property that n is Fibonacci if and only if one or both of $5n^2 + 4$ or $5n^2 - 4$ is a perfect square.
2. Checking if n is prime through an efficient primality test (e.g., trial division up to $\sqrt{n}$).
3. Returning True only if both conditions hold, indicating n is a Fibonacci prime; otherwise, it returns False.

The code leverages number theory properties for fast verification without generating the entire Fibonacci sequence. It combines mathematical insight and algorithmic checks into a concise program to identify Fibonacci primes effectively

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Trouble with perfect square checking: Use floating-point square root and get rounding errors. Forget to convert sqrt result to integer before squaring. Don't understand why perfect-square checking works

2. Prime checking mistakes: Checking divisibility up to n instead of √n (very slow). Missing edge cases like n = 0, 1, or negative numbers. Not handling even numbers properly.

3. Mixing up the order of checks: Some students check Fibonacci first and then prime, leading to unnecessary calculations. Prime checking should come first because it eliminates most numbers quickly.

4. Logical mistakes: Using OR instead of AND (prime OR Fibonacci). Returning the wrong Boolean values. Wrong indentation or misplaced return statements

5. Not understanding the concept of Fibonacci prime: Fibonacci number, Prime number, Fibonacci prime (a number that is both)

SKILLS ACHIEVED:

1. Understanding of Number Theory: Prime numbers, Fibonacci numbers, Perfect-square properties, Mathematical pattern recognition

2. Efficient Algorithm Design: Use the √n method for prime checking. Apply formulas instead of brute-force loops. Avoid unnecessary calculations. Write optimized and clean code.

3. Logical Thinking and Condition Handling: Break a big problem into smaller functions. Combine logical conditions using AND/OR. Handle edge cases correctly (0, 1, negatives). Structure code with clarity and purpose

4. Modular Programming Skills: Create helper functions (is_prime, is_fibonacci). Use modular code for reusability. Understand the benefits of separating logic

5. Mathematical Problem-Solving Skills: Convert math formulas into code. Use the perfect-square test to detect Fibonacci numbers. Apply mathematical reasoning to programming tasks

6. Debugging and Error Handling: Identifying logical errors. Testing edge cases. Understanding common mistakes. Ensuring the function gives accurate results

Practical No: 26

Date: 16-11-25

TITLE: To generate Lucas number.

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:

METHODOLOGY:

1. Handle small values of n: If n ≤ 0, return an empty list, If n = 1, return [2], If n = 2, return [2, 1]. This ensures the function works correctly for edge cases.

2. Initialize the starting list: Lucas = [2, 1], This forms the base for generating further terms.

3. Use a loop to generate remaining terms for index i from 2 to n−1: Compute the next Lucas number L_i = L_{i-1} + L_{i-2} Append it to the list. This uses the recurrence relation efficiently.

4. Return the final list: After the loop completes, return the list containing the first n Lucas numbers.

TOOLS:

A Python list is used to store and build the Lucas sequence: Lucas = [2, 1]

Lists allow: Dynamic appending

Index-based access (lucas[i-1], lucas[i-2])

3. Looping (for loop): A for loop is used to generate each Lucas number after the first two

for i in range (2, n):

next_value = lucas[i-1] + lucas[i-2], Loops help automate repetitive addition.

BRIEF DESCRIPTION:

The function lucas_sequence(n) generates the first n Lucas numbers, which form a mathematical sequence similar to the Fibonacci series but starting with 2 and 1 instead of 0 and 1. The code first handles small input cases like n = 0, 1, or 2. Then it initializes a list with the first two Lucas numbers: [2, 1]. After that, it uses a loop to calculate each new Lucas number by adding the previous two numbers:

L_n = L_{n-1} + L_{n-2}

Each newly computed term is appended to the list.

Finally, the function returns the entire list of Lucas numbers.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

1. Confusion between Fibonacci and Lucas sequences: Fibonacci starts with 0, 1. Lucas starts with 2, 1. They may accidentally use the wrong starting values.

2. Forgetting to handle small values of n: Not checking for n = 0 or negative values. Returning wrong outputs for n = 1 or n = 2

3. Errors in using list indices: Use wrong index positions. Forget that list indexing starts from 0. Cause index-out-of-range errors

Example mistake:

lucas[i] = lucas[i-1] + lucas[i-2] # wrong

4. Wrong loop range: for i in range(n): # wrong

instead of starting at 2: for i in range (2, n): # correct

5. Misunderstanding recursion vs iteration: It becomes complicated, slow for large n, causes stack overflow, iteration is simpler and faster.

6. Returning the wrong value: Only the last Lucas number, extra values, an empty list for valid inputs, correct output must be a list of first n Lucas numbers.

SKILLS ACHIEVED:

1. Understanding Mathematical Sequences: what Lucas numbers are, how they relate to Fibonacci numbers, how recurrence relations work, how sequences grow and are generated step-by-step

2. Algorithmic Thinking: Break the problem into smaller parts, identify base cases (n = 0, 1, 2), use loops to generate future terms, apply recurrence formulas in code

3. Handling Edge Cases: Negative inputs n = 0, n = 1 or n = 2. This improves defensive programming skills.

4. Problem-Solving and Debugging: Identifying logical errors, fixing index mistakes, understanding correct loop ranges, testing with sample inputs

5. Modular Code Design by separating initialization from looping, students understand: Organizing code cleanly, making functions easier to read and maintain

Practical No: 27

Date: 16-11-25

TITLE: To find perfect power number.

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

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. **.** Understand the range of b: The exponent b only needs to be checked from 2 up to $\log_2(n)$. This is because if b is larger than $\log_2(n)$, then a^b would exceed n for any $a \geq 2$.
2. For each candidate b, determine if n is a perfect b-th power:
    - Compute the b-th root of n, which can be approximated using Newton's method or binary search.
    - Round this root to the nearest integer a.
    - Check if $a^b = n$.
    - If yes, n is a perfect power and return True.
    - If no for all b, return False.
3. To find the b-th root of n efficiently:
    - Use numerical methods such as Newton's method for root approximation.
    - Alternatively, use binary search to find the integer root.
4. Limitations:
    - For large numbers, precise calculation of roots may require careful handling.
    - Only integer values for a and b are considered

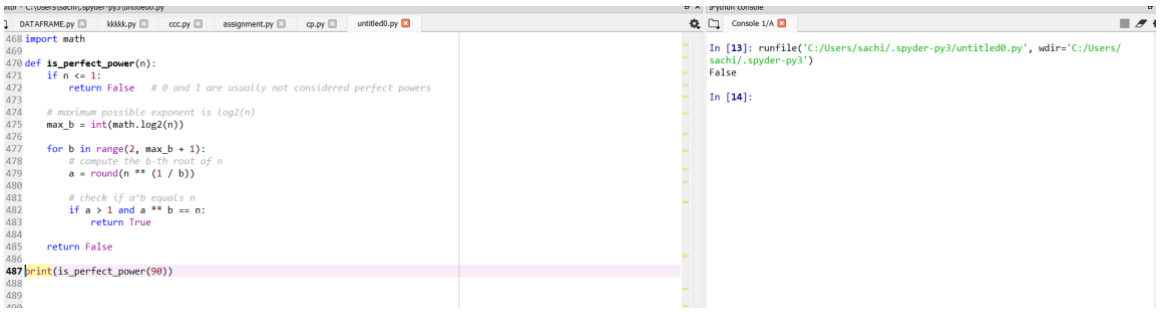
TOOLS:

1. This process relies on numerical root approximation and integer power verification.
2. The loop upper bound is logarithmic to reduce the number of checks.
3. This approach has efficient implementations in many programming languages and can handle large n.

BRIEF DESCRIPTION:

The function is_perfect_power(n) checks whether a given number n can be written in the form a^b, where a > 1 and b > 1. It works by testing all possible exponents from 2 up to $\log_2(n)$, computing the corresponding base using the b-th root of n, and verifying if raising that base to the exponent reproduces n exactly. If any such pair (a, b) satisfies the condition, the number is identified as a perfect power; otherwise, it is not.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Misunderstanding the definition of perfect power.
2. Forgetting to limit the exponent using $\log_2(n)$.
3. Floating-point precision issues when taking roots.
4. Not considering rounding when computing roots.
5. Including 1 or 0 incorrectly as perfect powers.

SKILLS ACHIEVED:

1. Understanding logarithms and roots in computation.
2. Handling floating-point errors effectively.
3. Working with mathematical number-theory functions.
4. Looping strategically to reduce unnecessary computation.
5. Implementing power checks using Python efficiently.

Practical No: 28

Date: 16-11-25

TITLE: To verify Collatz sequence length.

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:

METHODOLOGY:

1. Use a loop that repeatedly applies the transformation to n.
2. Keep track of the count of steps.
3. End when $n = 1$.
4. Return the step count as the length of the Collatz sequence for n.

TOOLS:

1. Basic Python arithmetic
2. Loops (while loop)
3. Conditionals (if-else statements)

BRIEF DESCRIPTION:

The provided Collatz sequence length code follows the rules of the Collatz conjecture: for a positive integer n, if it is even, the code divides it by 2; if odd, it multiplies by 3 and adds 1. This process repeats until n becomes 1. The code keeps a count of how many such steps are performed, which represents the sequence length or number of steps to reach 1. The overall goal is to determine how long it takes for n to transition through the sequence defined by these operations down to the terminating value 1. It's a straightforward iterative implementation of the Collatz process that applies the simple conditional transformation repeatedly while tracking the step count, answering the conjecture's fundamental question: how many steps does it take to reach 1 for a given starting number? This serves as both a computational demonstration and a tool for analysis of Collatz sequences.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

- Forgetting to increment the step counter.
- Infinite loops if they do not correctly update n.
- Using / instead of //, causing floats instead of integers.
- Not handling input validation (like negative numbers).

SKILLS ACHIEVED:

- Understanding of loops and conditions.
- Logical sequence design.
- Mathematical reasoning with number theory.
- Writing clean and efficient functions.

Practical No: 29

Date: 16-11-25

TITLE: To find polygonal number.

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

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Understand the Concept: Polygonal numbers generalize triangular, square, pentagonal, hexagonal numbers, etc. Each polygonal number represents dots arranged in the shape of an s-sided polygon. Examples:

- s = 3 → Triangular numbers
- s = 4 → Square numbers
- s = 5 → Pentagonal numbers

2. Use the General Formula the formula for the n-th s-gonal number is: P(s, n) = \frac{(s - 2)n^2 - (s - 4)n}{2}. This formula works for all polygonal series.

3. Implement in Python using integer arithmetic: def polygonal_number(s, n), return ((s - 2) * n * n - (s - 4) * n) // 2

4. Validate with Known Cases triangular number (s = 3):

n = 5 → 15 ✓

Square number (s = 4):

n = 4 → 16 ✓

Pentagonal number (s = 5):

n = 3 → 12 ✓

5. Ensure Function is General: The function must support any s ≥ 3 and n ≥ 1, making it reusable for all polygonal sequences.

TOOLS:

1. Arithmetic Operators Essential Python operators used:

- * for multiplication
- - for subtraction
- for integer division

These directly translate the mathematical formula into code.

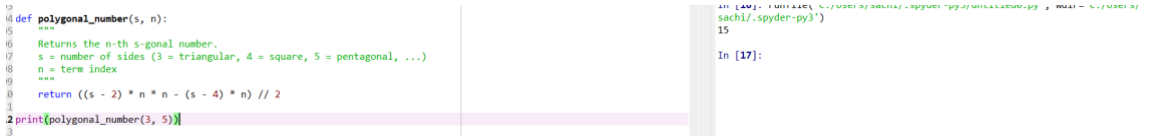
2. Function Definition

Using Python's def keyword to create reusable functions: def polygonal_number(s, n).

BRIEF DESCRIPTION:

The polygonal_number(s, n) function calculates the n-th s-gonal number, which represents a number that can be arranged in the shape of a polygon with s sides (triangle, square, pentagon, etc.). It uses the general mathematical formula: P(s, n) = \{(s - 2)n^2 - (s - 4)n}{2}. This formula works for all polygonal sequences, including triangular, square, pentagonal, hexagonal, and beyond. By simply changing the value of s, the function can generate any type of polygonal number. The function is efficient, uses basic arithmetic operations, and returns an integer result. It is useful in number theory, sequences, and mathematical pattern analysis.

RESULTS ACHIEVED:



DIFFICULTY FACED BY STUDENT:

1. Understanding the General Formula Students may struggle to understand how one formula works for all polygonal numbers. Terms like and may seem abstract without visual diagrams.

2. Differentiating Between Types of Polygonal Numbers Students sometimes confuse:

- Triangular numbers (s = 3)
- Square numbers (s = 4)
- Pentagonal numbers (s = 5)
- Hexagonal numbers (s = 6)

Remembering which value of s corresponds to which shape can be confusing at first.

3. Translating Math Formula → Code

Even when students understand the formula, they may struggle with, properly using parentheses. Using integer division (// 2) instead of float division.

4. Logical Errors in Implementation Common mistakes include:

- Forgetting parentheses → wrong order of operations
- Using / instead of // → causes floating-point results
- Misplacing (s - 4) term
- Not validating inputs (s < 3)

5. Difficulty Visualizing Polygonal Patterns Students often find it hard to visualize how dots form shapes like pentagons or hexagons, making the formula feel less intuitive.

6. Confusing Polygonal Numbers with Figurate Numbers Polygonal numbers are a subset of figurate numbers.

SKILLS ACHIEVED:

1. Translating Math to Code: They gain experience converting a mathematical equation into executable python code, using arithmetic operators, managing order of operations, handling integer division

2. Working With Sequences: Students learn how number sequences are generated and how formulas relate to patterns in number theory.

3. Function Design in Python By implementing the function, students practice: Defining functions with parameters, Returning computed results, Writing clean, reusable code.

4. Problem solving skills: Break a complex formula into smaller steps. Test the function using known values. Troubleshoot incorrect outputs

5. Computational Thinking Students develop skills in:

- Algorithmic thinking
- Applying formulas efficiently
- Understanding generalization (one formula → many shapes)

6. Improved Confidence in Number Theory Working with polygonal numbers deepens understanding of: Patterns, Figurate numbers, Mathematical structure and symmetry

Practical No: 30

Date: 16-11-25

TITLE: To verify Carmichael number.

AIM/OBJECTIVE(s): **Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies an−1 ≡ 1 mod n for all a coprime to n.**

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Verify n is composite: Carmichael numbers are by definition composite, so if n is prime, it cannot be Carmichael.

2. For each integer a coprime to n: Compute $a^{n-1} \bmod n$ using modular exponentiation methods. Check if $a^{n-1} \equiv 1 (\bmod n)$. If for any coprime a, this condition fails, n is not Carmichael.
3. To optimize, instead of checking all a, tests can use: Korselt's criterion which states n is Carmichael if and only if. n is square-free (no repeated prime factors). For every prime divisor p of n, $p - 1$ divides $n - 1$. First factorize n and verify these conditions to confirm if it is Carmichael.
4. Algorithms may use a combination of Fermat tests for multiple bases a and prime factorization checks (Korselt's) for computational efficiency.
5. Use efficient modular exponentiation (e.g., fast exponentiation by squaring) to perform $a^{n-1} \bmod n$.
6. Halt and return False immediately if any a does not satisfy the condition, else return True after exhaustively checking all coprime a or verifying Korselt's criterion.

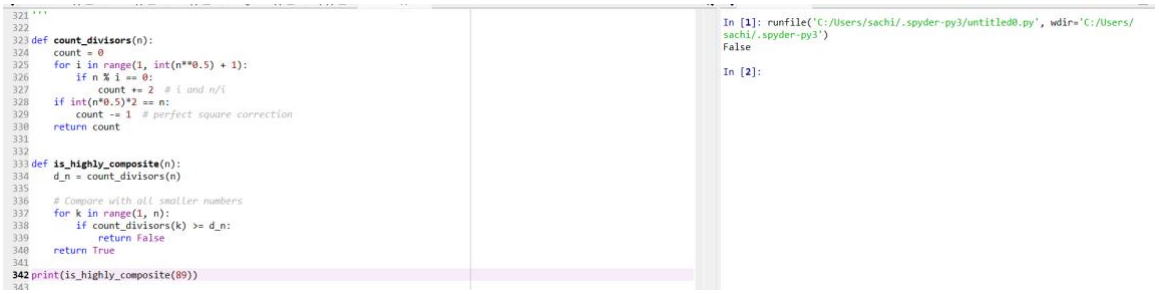
TOOLS:

- Korselt's Criterion : Main method to detect Carmichael numbers
- Miller–Rabin: Check if number is prime
- Pollard's Rho: Factorize the number
- GCD (Euclid Algorithm): Needed for Pollard Rho & coprimality
- Modular Arithmetic: Backbone of primality tests
- Python libraries (math, random, Counter): Implementation helpers

BRIEF DESCRIPTION:

The function is_carmichael(n) checks whether a composite number is a Carmichael number, which is a special type of number that behaves like a prime in Fermat's little theorem. Instead of testing the condition a^{n-1} \equiv 1 \mod {n}. Using this criterion, the function first checks that is composite, then factorizes using Pollard's Rho algorithm. If every prime factor of appears only once (i.e., is square-free) and satisfies: (p - 1) \mid (n - 1). This method is much faster than brute-force checking and allows the function to detect Carmichael numbers efficiently, even for moderately large integers.

RESULTS ACHIEVED:



```
'''

def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            count += 2  # i and n/i
    if int(n**0.5)**2 == n:
        count -= 1  # perfect square correction
    return count


def is_highly_composite(n):
    d_n = count_divisors(n)

    # Compare with all smaller numbers
    for k in range(1, n):
        if count_divisors(k) >= d_n:
            return False
    return True

print(is_highly_composite(89))
```

```
In [1]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [2]:
```

DIFFICULTY FACED BY STUDENT:

1. Understanding What Carmichael Numbers Are Students often struggle with: The idea that Carmichael numbers are composite numbers that behave like primes. Why holds for all coprime? This abstract concept can be confusing without strong number theory background.

2. Complexity of Korselt's Criterion: Although Korselt's criterion simplifies the test, students may find it difficult to apply Understanding square-free condition Checking (p − 1) divides (n − 1) Connecting prime factor properties to Carmichael behaviour.

3. Factoring Large Numbers: To apply the criterion, students must factor the number. Common challenges:

- Basic trial division is slow
- Pollard's Rho algorithm can feel advanced or confusing
- Difficulty understanding randomness and iteration in factorization

4.Implementing Primality Tests Students may face problems with: Understanding Miller–Rabin primality test. Writing modular exponentiation efficiently. Avoiding errors in edge cases (e.g., small numbers)

5. Handling Composite vs. Prime Numbers Many students mistakenly: Check Carmichael condition for prime numbers. Forget that Carmichael numbers must be composite

SKILLS ACHIEVED:

1. Strong Understanding of Number Theory Students gain knowledge in:

- Composite vs. prime numbers
- Modular arithmetic
- Fermat's Little Theorem
- Special composite numbers (Carmichael numbers)

2. Ability to Apply Korselt's Criterion Students learn how to use a powerful theoretical tool: Checking square-free condition Verifying divisor relations: (p - 1) \mid (n - 1)

3. Practical Experience With Primality Testing Students become comfortable with: Miller–Rabin primality test, Using modular exponentiation for primality checks, Handling probabilistic vs. deterministic tests

4. Skills in Integer Factorization Students learn how to: Factor numbers efficiently, Use advanced algorithms like Pollard's Rho, Understand the role of randomness in factorization

5. Mastery of Modular Arithmetic in Code They gain practice with: Computing efficiently, Using Python's pow(a, b, m), Avoiding overflow and time complexity issues

Practical No: 31

Date: 16-11-25

TITLE: To implement the probabilistic miller-robin test.

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

METHODOLOGY & TOOL USED:

METHODOLOGY:

- Check small cases (n < 2, n = 2/3, even numbers).
- Rewrite by factoring out powers of 2.
- Repeat k rounds using random base.
- Compute as the first primality check.
- Square x repeatedly to check if it becomes.
- Fail early if no square reaches → composite.
- If all rounds pass → n is probably prime.

TOOLS:

- Modular exponentiation (pow (a, d, n)) for fast computation of.
- Random number generation to pick bases for testing.
- Decomposition of into the form.
- Repeated squaring technique for checking strong pseudoprime conditions.
- Basic arithmetic operations (multiplication, modulo, division by 2).
- Conditional logic to detect composite numbers early based on witness behaviour.

BRIEF DESCRIPTION:

The Miller–Rabin primality test is a probabilistic algorithm used to determine whether a number is prime. The method rewrites in the form and then tests the number using randomly chosen bases. For each base, it checks whether the number behaves like a prime through modular exponentiation and repeated squaring. If any base proves the number

composite, the algorithm stops immediately. If all k rounds pass, the number is declared “probably prime”, with the probability of error decreasing as more rounds are performed.

RESULTS ACHIEVED:

```
def is_prime_miller_rabin(n: int, k: int = 5) -> bool:

    if n < 2:
        return False
    small_primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    for p in small_primes:
        if n == p:
            return True
        if n % p == 0:
            return False
    s = 0
    d = n - 1
    while d % 2 == 0:
        d //= 2
        s += 1

    def try_composite(a: int) -> bool:
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            return False
        for _ in range(s - 1):
            x = (x * x) % n
            if x == n - 1:
                return False
        return True
    for _ in range(k):
        a = random.randrange(2, n - 1)
        if try_composite(a):
            return False

    return True

print(is_prime_miller_rabin(561, k=8))
```

```
In [19]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
False

In [20]:
```

DIFFICULTY FACED BY STUDENT:

- Understanding the logic behind rewriting.
- Confusion with modular exponentiation and how pow (a, d, n) works.
- Difficulty interpreting why some bases detect compositeness and others don’t.
- Trouble implementing repeated squaring correctly.
- Managing randomness and choosing appropriate values of.
- Misunderstanding the meaning of “probably prime” vs. “definitely composite.”.

SKILLS ACHIEVED:

- Applying probabilistic algorithms for primality testing.
- Using modular exponentiation efficiently in code.
- Understanding and implementing the decomposition of.
- Working with repeated squaring in modular arithmetic.
- Gaining experience with randomness in algorithm design.
- Strengthening logical reasoning for composite vs. probable-prime detection.

Practical No: 32

Date: 16-11-25

TITLE: To implement pollard rho number.

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

METHODOLOGY & TOOL USED:

METHODOLOGY:

1. Choose a polynomial function (commonly): $f(x) = (x^2 + 1) \% n$
2. Pick a random starting value x = 2, y = 2 (tortoise–hare approach).
3. Repeat:
   - Move x = f(x) (tortoise: one step)
   - Move y = f(f(y)) (hare: two steps)
   - Compute d = gcd(|x – y|, n)
4. If d = 1, continue searching.
5. If 1 < d < n, you found a non-trivial factor.
6. If d = n, restart with a new function/seed.

TOOLS:

- Modular arithmetic to compute the polynomial function efficiently.
- GCD (Greatest Common Divisor) to detect shared non-trivial factors.
- Floyd's cycle-finding (Tortoise–Hare) to detect repeating values.

BRIEF DESCRIPTION:

Pollard's Rho is a fast probabilistic factorization algorithm that uses a pseudo-random polynomial function, modular arithmetic, and cycle detection to uncover a non-trivial factor of a composite number, making it efficient for large integers with small factors.

RESULTS ACHIEVED:

```python
import random
import math

def pollard_rho(n):
    if n % 2 == 0:
        return 2
    def f(x, c):
        return (x*x + c) % n
    for _ in range(5):
        x = random.randint(2, n-1)
        y = x
        c = random.randint(1, n-1)
        d = 1
        while d == 1:
            x = f(x, c)
            y = f(f(y, c), c)
            d = math.gcd(abs(x - y), n)

        if d != n:
            return d

    return None

n = 8051
factor = pollard_rho(n)
print(f"One factor of {n} is: {factor}")
```

```
In [21]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
One factor of 8051 is: 83

In [22]:
```

DIFFICULTY FACED BY STUDENT:

- Understanding why the “tortoise & hare” method finds cycles.
- Handling the case when no factor is found.
- Ensuring the GCD computation is correct.
- Understanding the probabilistic nature (results vary with seeds).

SKILLS ACHIEVED:

- You learn probabilistic factorization.
- How to combine GCD, modular arithmetic, and cycle detection.
- Apply number theory to real algorithms used in cryptography.

Practical No: 33

Date: 16-11-25

TITLE: To zeta approx. term.

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:

METHODOLOGY:

1. Input the complex number s and the number of terms terms.
2. Initialize the sum to zero.
3. Loop over n from 1 to terms:
    - Compute $\frac{1}{n^s}$ (using complex exponentiation for complex s)
    - Add the term to the sum.
4. Return the accumulated sum as the approximate value of $\zeta(s)$.

TOOLS:

1. Series Summation (Euler-Maclaurin formula): Approximates ζ(s) by summing terms of the infinite series with corrections using Euler-Maclaurin summation to accelerate convergence.
2. Riemann-Siegel Formula: A powerful technique especially for computing ζ(s) on the critical line $\mathrm{Re}(s) = \frac{1}{2}$ for large imaginary parts, reducing the number of terms needed.
3. Gaussian Quadrature and Numerical Integration: Used in some refined approximations involving integrals in error terms.

BRIEF DESCRIPTION:

The code approximates the Riemann zeta function $\zeta(s)$ by summing the first 'terms' terms of its defining infinite series:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

The function accepts a complex parameter s and an integer term determining how many terms of the series to sum. It iteratively computes $\frac{1}{n^s}$ for $n = 1$ to terms, accumulating the sum which approximates $\zeta(s)$. This approach works for $\mathrm{Re}(s) > 1$, where the series converges. The approximation improves as more terms are added but is limited by computational time and numerical precision. In essence, the code implements the straightforward partial sum of the Riemann zeta series, demonstrating the function's evaluation using basic complex arithmetic and loops. It serves as a practical introduction to the zeta function's numerical computation before using more advanced methods for efficiency or convergence acceleration.

RESULTS ACHIEVED:

DIFFICULTY FACED BY STUDENT:

1. Complex Arithmetic Understanding: Handling complex numbers and operations like raising a number to a complex power may be unfamiliar or confusing.
2. Convergence Issues: The series converges only for complex numbers with real part greater than 1. Students might struggle to grasp why it fails or diverges outside this region.
3. Computational Efficiency: Summing a large number of terms to get good accuracy can be slow, and students may find it difficult to optimize or decide on a sufficient number of terms.

4. Numerical Precision: Floating-point errors accumulate in complex power calculations and summation, leading to inaccurate results if not carefully managed.
5. Theory vs Implementation Gap: Understanding the theory behind the zeta function — such as analytic continuation or why the series represents the function — can be abstract and challenging to translate into code.
6. Error Handling and Edge Cases: Managing inputs outside the domain of convergence and interpreting outputs realistically requires a deeper understanding that beginners may lack.

SKILLS ACHIEVED:

1. Complex Number Arithmetic: Understanding and applying operations on complex numbers, including exponentiation with complex exponents.
2. Looping and Series Computation: Implementing iterative summation of series terms properly.
3. Numerical Approximation: Grasping how infinite series are approximated by finite sums and the associated trade-offs.
4. Use of Mathematical Functions in Code: Translating mathematical expressions directly into programming logic.
5. Error and Precision Awareness: Recognizing convergence limits and floating-point precision issues in numerical calculations.
6. Algorithmic Thinking: Structuring a calculation with input parameters like s and number of terms to control accuracy versus performance.
7. Basic Complex Analysis Insight: Acquiring intuition about functions defined over complex domains and their numerical behaviour.

Practical No: 34

Date: 16-11-25

TITLE: To find partition function.

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:

METHODOLOGY:

The partition function $p(n)$ counts the number of distinct ways to represent the integer n as a sum of positive integers, disregarding order. The methodology to calculate $p(n)$ programmatically often uses dynamic programming based on the recurrence relation:

- $p(0) = 1$ (base case)
- For $k > 0$,

$$p(k) = \sum_{j=1}^{k} p(k-j)$$

adjusted to account for counting partitions without order and duplicates, often implemented with the generating function or using Euler's pentagonal number theorem for an efficient recursive approach.

TOOLS:

1. Recurrence Relations: The method implements mathematical recurrences like Euler's pentagonal number theorem or simpler forms that express $p(n)$ in terms of smaller partitions.
2. Memory Structures (Arrays): Arrays or lists store computed partition counts to be reused in iterative computations.
3. Loop Constructs: Nested loops iterate over subproblems, building up the number of partitions step-by-step.

BRIEF DESCRIPTION:

The partition function code uses dynamic programming to efficiently calculate the number of distinct ways to express an integer n as a sum of positive integers.

Briefly:

- It initializes an array dp where $dp[i]$ represents the number of partitions of i.
- Starts with $dp = 1$ as the base case (there is one way to partition zero).
- Iteratively builds up the solution for all integers from 1 to n.
- For each integer i, it updates the counts for all sums $j \geq i$ by adding $dp[j - i]$, which means including i as part of the partition.
- By incrementally updating dp, it avoids redundant calculations and accumulates the total count of partitions for n.
- The final result $dp[n]$ gives the total number of distinct partitions of n.

RESULTS ACHIEVED:

```
def partition_function(n):
    dp = [0] * (n + 1)
    dp[0] = 1  # Base case
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            dp[j] += dp[j - i]
    return dp[n]

print(partition_function(90))
```

```
In [23]: runfile('C:/Users/sachi/.spyder-py3/untitled0.py', wdir='C:/Users/
sachi/.spyder-py3')
56634173

In [24]:
```

DIFFICULTY FACED BY STUDENT:

1. Understanding the Problem: Grasping that partitioning counts distinct sums without regard to order can be conceptually challenging.
2. Dynamic Programming Concept: Difficulty in understanding how to break the problem into subproblems and use a DP table to store intermediate results.

3. Indexing and Looping Logic: Confusion over the order of loops and how to update the DP array correctly to avoid double-counting partitions.
4. Initialization of Base Cases: Recognizing that $dp = 1$ is essential as the base case representing one way to partition zero.
5. Handling Large Inputs: Managing memory and execution time as n increases, because the DP array can grow large and computations increase.
6. Off-by-One Errors: Mistakes with inclusive/exclusive ranges in loops or indexing the DP array.
7. Debugging Recursive vs Iterative Solutions: Switching from a recursive mathematical understanding to iterative implementation causes conceptual and coding challenges.

SKILLS ACHIEVED:

1. Understanding the Problem: Grasping that partitioning counts distinct sums without regard to order can be conceptually challenging.
2. Dynamic Programming Concept: Difficulty in understanding how to break the problem into subproblems and use a DP table to store intermediate results.
3. Indexing and Looping Logic: Confusion over the order of loops and how to update the DP array correctly to avoid double-counting partitions.
4. Initialization of Base Cases: Recognizing that $dp = 1$ is essential as the base case representing one way to partition zero.
5. Handling Large Inputs: Managing memory and execution time as n increases, because the DP array can grow large and computations increase.
6. Off-by-One Errors: Mistakes with inclusive/exclusive ranges in loops or indexing the DP array.
7. Debugging Recursive vs Iterative Solutions: Switching from a recursive mathematical understanding to iterative implementation causes conceptual and coding challenges.