Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BAI11344
Name of Student	: AMAN KUMAR BEHERA
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCAI
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Write a function 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 ged(n, k) = 1).	16/11/2025	
2	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. 1(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.	16/11/2025	
3	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 o(n).	16/11/2025	
4	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.	16/11/2025	
5	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 a 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.	16/11/2025	
6	Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).	16/11/2025	
7	Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.	16/11/2025	
8	Write a function mean_of_digits(n) that returns the average of all digits in a number.	16/11/2025	

9	Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	16/11/2025	
10	Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n	16/11/2025	
11	Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.	16/11/2025	
12	Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.	16/11/2025	
13	Write a function is_automorphic(n) that checks if a number's square ends with the number itself.	16/11/2025	
14	Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.	16/11/2025	
15	Write a function prime_factors(n) that returns the list of prime factors of a number.	16/11/2025	
16	Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.	16/11/2025	
17	Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and $k \geq 1$.	16/11/2025	
18	Write a function is_mersenne_prime(p) that checks if $2^p - 1$ is a prime number (given that p is prime).	16/11/2025	
19	Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.	16/11/2025	
20	Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.	16/11/2025	
21	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	16/11/2025	
22	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).	16/11/2025	
23	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	16/11/2025	
24	Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.	16/11/2025	

25	Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates ($base^{exponent}$) % modulus.	16/11/2025	
26	Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.	16/11/2025	
27	Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences $x \equiv r_i \bmod m_i$	16/11/2025	
28	Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if $x^2 \equiv a \bmod p$ has a solution.	16/11/2025	
29	Write a function order_mod(a, n) that finds the smallest positive integer k such that $a^k \equiv 1 \bmod n$.	16/11/2025	
30	Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.	16/11/2025	
31	Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1).	16/11/2025	
32	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.	16/11/2025	
33	Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.	16/11/2025	
34	Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.	16/11/2025	
35	Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies $a^{n-1} \equiv 1 \bmod n$ for all a coprime to n.	16/11/2025	
36	Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.	16/11/2025	
37	Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.	16/11/2025	
38	Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.	16/11/2025	

39	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.	16/11/2025	

Practical No: 1

Date: 16/11/2025

TITLE : Euler's Totient Function (φ(n))

AIM/OBJECTIVE(s): To implement a Python function **euler_phi(n)** that computes Euler's Totient φ(n), which counts the number of integers ≤ n that are coprime to n.

METHODOLOGY & TOOL USED: Number-theoretic formula for φ(n) using prime factorization; Python.

BRIEF DESCRIPTION: Euler's Totient Function φ(n) counts positive integers ≤ n that are coprime to n.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def euler_phi(n):
    if n <= 0:
        return 0
    result = n
    x = n
    p = 2
    while p * p <= x:
        if x % p == 0:
            while x % p == 0:
                x //= p
            result -= result // p
        p += 1
    if x > 1:
        result -= result // x
    return result
print(euler_phi(10))
print(euler_phi(100))
print(euler_phi(112330))
```

RESULTS ACHIEVED:

Correct computation of φ(n) for composite and prime numbers.

```
Elapsed time:0.100920 seconds
4
40
4488
```

DIFFICULTY FACED BY STUDENT:

Handling factorization and avoiding repeated subtraction for repeated prime factors.

SKILLS ACHIEVED:

Prime factorization, multiplicative functions, efficient integer computation.

Practical No: 2

Date: 16/11/2025

TITLE : Möbius Function μ(n)

AIM/OBJECTIVE(s): To implement **mobius(n)**, which computes the Möbius function μ(n).

METHODOLOGY & TOOL USED: Prime factorization and square-free checking; Python.

BRIEF DESCRIPTION: The Möbius function μ(n) is defined as:

- μ(n) = **1** if n is square-free and has **even** number of prime factors
- μ(n) = **-1** if n is square-free and has **odd** number of prime factors
- μ(n) = **0** if n contains any repeated (squared) prime factor

The function factorizes n and counts prime factors while checking for squares.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def mobius(n):
    if n == 0:
        return 0
    n = abs(n)
    cnt = 0
    x = n
    p = 2
    while p * p <= x:
        if x % p == 0:
            cnt += 1
            x //= p
            if x % p == 0:
                return 0
        p += 1
    if x > 1:
        cnt += 1
    return 1 if cnt % 2 == 0 else -1
print(mobius(30))
print(mobius(67))
print(mobius(4))
```

RESULTS ACHIEVED: Correct calculation of μ(n) across all cases.

```
Elapsed time:0.094902 seconds
-1
-1
0
```

DIFFICULTY FACED BY STUDENT: Detecting square prime factors accurately.

SKILLS ACHIEVED: Square-free checking, prime factor counting, number theory function evaluation.

Practical No: 3

Date: 16/11/2025

TITLE : Divisor Sum Function σ(n)

AIM/OBJECTIVE(s): To implement **divisor_sum(n)** to compute the sum of all positive divisors of n.

METHODOLOGY & TOOL USED: Divisor enumeration up to √n; Python.

BRIEF DESCRIPTION: The function σ(n) sums all positive divisors of n (including 1 and n).
We iterate from 1 to √n, collecting divisor pairs to avoid redundant checks.
Time complexity: O(√n).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def divisor_sum(n):
    if n <= 0:
        return 0
    total = 0
    limit = int(math.isqrt(n))
    for i in range(1, limit+1):
        if n % i == 0:
            total += i
            j = n // i
            if j != i:
                total += j
    return total
print(divisor_sum(12))
print(divisor_sum(90))
print(divisor_sum(1442))
```

RESULTS ACHIEVED: Correct divisor sum for prime, composite, and perfect numbers.

```
Elapsed time:0.093898 seconds
28
234
2496
```

DIFFICULTY FACED BY STUDENT: Avoiding double-counting divisors for perfect squares.

SKILLS ACHIEVED: Efficient divisor enumeration, integer arithmetic.

Practical No: 4

Date: 16/11/2025

TITLE : Prime Counting Function Approximation π(n)

AIM/OBJECTIVE(s): To implement **prime_pi(n)** that returns the count of primes ≤ n.

METHODOLOGY & TOOL USED: Simple sieve-based prime counter; Python.

BRIEF DESCRIPTION:π(n) counts primes ≤ n.
The Sieve of Eratosthenes provides an efficient way to generate all primes ≤ n in O(n log log n).
The function returns the number of True values in a boolean sieve.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def prime_pi(n):
    if n < 2:
        return 0
    sieve = [True] * (n + 1)
    sieve[0] = sieve[1] = False
    p = 2
    while p * p <= n:
        if sieve[p]:
            for i in range(p*p, n+1, p):
                sieve[i] = False
        p += 1
    return sum(sieve)
print(prime_pi(10))
print(prime_pi(100))
print(prime_pi(374))
```

RESULTS ACHIEVED: Correct prime counts for tested ranges.

```
Elapsed time:0.104332 seconds
4
25
74
```

DIFFICULTY FACED BY STUDENT: Understanding sieve optimizations & marking multiples.

SKILLS ACHIEVED: Prime generation, sieve implementation, efficient Boolean array usage.

Practical No: 5

Date: 16/11/2025

TITLE : Legendre Symbol (a/p)

AIM/OBJECTIVE(s): To implement **legendre_symbol(a, p)** using Euler's Criterion.

METHODOLOGY & TOOL USED: Modular exponentiation; Python's pow function with modulus.

BRIEF DESCRIPTION: Legendre symbol (a/p) is defined for odd prime p:

- (a/p) = 1 → a is a quadratic residue mod p
- (a/p) = -1 → a is quadratic non-residue mod p
- (a/p) = 0 → if p divides a
- If result is 1 → 1
- If result is p−1 → -1
- If result is 0 → 0

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def legendre_symbol(a, p):
    if p <= 2 or p % 2 == 0:
        raise ValueError("p must be an odd prime")
    a = a % p
    if a == 0:
        return 0
    res = pow(a, (p - 1) // 2, p)
    if res == 1:
        return 1
    elif res == p - 1:
        return -1
    else:
        return 0
print(legendre_symbol(5, 11))
print(legendre_symbol(8, 73))
print(legendre_symbol(4, 29))
```

RESULTS ACHIEVED: Accurate evaluation of quadratic residue nature of integers mod p.

```
Elapsed time:0.096946 seconds
1
1
1
```

DIFFICULTY FACED BY STUDENT: Interpreting Euler's criterion output (1, p−1, 0).

SKILLS ACHIEVED: Modular arithmetic, quadratic residues, efficient exponentiation.

Practical No: 6

Date: 16/11/2025

TITLE: Factorial Function (factorial)

AIM/OBJECTIVE(s): Implement factorial(n) to compute n! for non-negative integer n.

METHODOLOGY & TOOL USED: Iterative multiplication using Python. Handles n = 0 via definition 0! = 1.

BRIEF DESCRIPTION: The function checks if n is a non-negative integer, returns None (or raises) for negative inputs. For n >= 0 it iteratively multiplies integers from 1 to n accumulating the product. Time complexity O(n).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time: [{elapsed_time:.6f}seconds")
def factorial(n):
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")
    result = 1
    for i in range(2, n+1):
        result *= i
    return result
print(factorial(5))
print(factorial(7))
```

RESULTS ACHIEVED: Correct factorials for typical inputs; handles edge n=0 and rejects negatives.

```
Elapsed time: [0.178817seconds
120
5040
```

DIFFICULTY FACED BY STUDENT: Dealing with invalid input types (floats, negatives).

SKILLS ACHIEVED: Looping, input validation, big-integer handling in Python.

Practical No: 7

Date: 16/11/2025

TITLE: Palindrome Check (is_palindrome)

AIM/OBJECTIVE(s): Implement is_palindrome(n) to verify if integer n reads the same forwards and backwards.

METHODOLOGY & TOOL USED: Convert to string and compare to reversed string (Python). Handles negative numbers as non-palindromic by convention (unless specified otherwise).

BRIEF DESCRIPTION: Convert n to string (skip leading minus sign). Compare s == s[::-1]. Time O(d) where d = digits.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time: [{elapsed_time:.6f}seconds")
def is_palindrome(n):
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    s = str(n)
    if s.startswith('-'):
        return False
    return s == s[::-1]
print(is_palindrome(121))
print(is_palindrome(333))
print(is_palindrome(78567))
```

RESULTS ACHIEVED: Correct for positive integers; negative treated as not palindrome.

```
Elapsed time: [0.199611seconds
True
True
False
```

DIFFICULTY FACED BY STUDENT: Decide policy on negative numbers and leading zeros.

SKILLS ACHIEVED: String manipulation, edge-case handling.

Practical No: 8

Date: 16/11/2025

TITLE: Mean of Digits (mean_of_digits)

AIM/OBJECTIVE(s): Implement mean_of_digits(n) returning average of all digits in n.

METHODOLOGY & TOOL USED: Convert to absolute value string, extract digits, compute sum/length. Python.

BRIEF DESCRIPTION: For integer n, take absolute value, get digits, compute sum(digits)/len(digits). Returns a float. Handles n=0 correctly (mean 0).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time: [{elapsed_time:.6f}seconds")
def mean_of_digits(n):
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    s = str(abs(n))
    digits = [int(ch) for ch in s]
    return sum(digits) / len(digits)
print(mean_of_digits(1234))
print(mean_of_digits(1235564))
```

RESULTS ACHIEVED: Correct averages, robust to negative input.

```
Elapsed time: [0.198314seconds
2.5
3.7142857142857144
```

DIFFICULTY FACED BY STUDENT: Ensure integer inputs and avoid division by zero (not needed for integers >=0).

SKILLS ACHIEVED: List comprehensions, numeric aggregation.

Practical No: 9

Date: 16/11/2025

TITLE: Digital Root (digital_root)

AIM/OBJECTIVE(s): Implement digital_root(n) that repeatedly sums digits until a single digit remains.

METHODOLOGY & TOOL USED: Iterative digit-sum loop or use mod 9 trick. Python.

BRIEF DESCRIPTION: Repeatedly replace n by sum of its decimal digits until n < 10. For n == 0 returns 0. Alternatively, for n>0 can use 1 + (n-1) % 9. Here implementing iterative method for clarity.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time: [{elapsed_time:.6f}seconds")
def digital_root(n):
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    n = abs(n)
    while n >= 10:
        n = sum(int(ch) for ch in str(n))
    return n
print(digital_root(9875))
print(digital_root(345))
print(digital_root(9))
```

RESULTS ACHIEVED: Correct single-digit root for inputs.

```
Elapsed time: [0.170658seconds
2
3
9
```

DIFFICULTY FACED BY STUDENT: Recognizing faster modular solution vs iterative clarity.

SKILLS ACHIEVED: Loops, digit manipulation, algorithmic optimization awareness.

Practical No: 10

Date: 16/11/2025

TITLE: Abundant Number Check (is_abundant)

AIM/OBJECTIVE(s): Implement is_abundant(n) returning True if sum of proper divisors of n > n.

METHODOLOGY & TOOL USED: Efficient divisor enumeration up to sqrt(n) to sum proper divisors (including 1, excluding n). Python.

BRIEF DESCRIPTION: For n > 1, iterate i from 1..sqrt(n), add divisor pairs to sum (ensure not to include n itself). Compare sum to n. For n <= 1, it's not abundant. Complexity roughly O(sqrt(n)).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time: [{elapsed_time:.6f}seconds")
import math
def is_abundant(n):
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    if n <= 1:
        return False
    total = 1
    limit = int(math.isqrt(n))
    for i in range(2, limit + 1):
        if n % i == 0:
            total += i
            j = n // i
            if j != i and j != n:
                total += j
    return total > n
print(is_abundant(12))
print(is_abundant(30))
print(is_abundant(667))
```

RESULTS ACHIEVED: Correct detection of abundant numbers using O(sqrt(n)).

```
Elapsed time: [0.113271seconds
True
True
False
```

DIFFICULTY FACED BY STUDENT: Avoid double-counting divisors and excluding n.

SKILLS ACHIEVED: Divisor enumeration, algorithmic complexity reduction.

Practical No: 11

Date: 16/11/2025

TITLE: Deficient Number Check (is_deficient)

AIM/OBJECTIVE(s): Implement is_deficient(n) returning True if the sum of proper divisors of n < n.

METHODOLOGY & TOOL USED: Same efficient divisor enumeration as abundant check; compare sum < n. Python.

BRIEF DESCRIPTION: For n > 1, compute sum of proper divisors and compare with n. For n = 1, proper divisors sum = 0 < 1 so True.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _= i*2
end_time=time.perf_counter()
elapsed_time= end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f}seconds")
import math
def is_deficient(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    sum_proper_divisors = 1
    limit = int(math.sqrt(n))
    for i in range(2, limit + 1):
        if n % i == 0:
            sum_proper_divisors += i
            complementary_divisor = n // i
            if i * i != n:
                sum_proper_divisors += complementary_divisor
    return sum_proper_divisors < n
print(is_deficient(65))
print(is_deficient(23))
print(is_deficient(9))
print(is_deficient(75))
```

RESULTS ACHIEVED: Correct classification for deficient numbers.

```
Elapsed time:0.090400seconds
True
True
True
True
```

DIFFICULTY FACED BY STUDENT: Edge case n=1.

SKILLS ACHIEVED: Reusing algorithms, handling special cases.

Practical No: 12

Date: 16/11/2025

TITLE: Harshad Number Check (is_harshad)

AIM/OBJECTIVE(s): Implement is_harshad(n) that checks if n is divisible by the sum of its digits.

METHODOLOGY & TOOL USED: Digit sum computation then modulus check. Python.

BRIEF DESCRIPTION: For integer n, compute s = sum(digits of abs(n)). If s == 0 (only for n==0) treat 0 as Harshad by convention (0 % 0 undefined — handle separately). Return n % s == 0.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _= i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"elapsed time:{elapsed_time:.6f}seconds")
def is_harshad(n):
    if n <= 0:
        return False
    digit_sum = sum(int(digit) for digit in str(n))
    return n % digit_sum == 0
print(is_harshad(34))
print(is_harshad(2))
print(is_harshad(9))
print(is_harshad(126374))
```

RESULTS ACHIEVED: Correct detection; handles zero specially.

```
elapsed time:0.095277seconds
False
True
True
False
```

DIFFICULTY FACED BY STUDENT: Division by zero for n=0 must be handled.

SKILLS ACHIEVED: Digit processing, guarding against zero-division.

Practical No: 13

Date: 16/11/2025

TITLE: Automorphic Number Check (is_automorphic)

AIM/OBJECTIVE(s): Implement is_automorphic(n) checking if n^2 ends with the digits of n.

METHODOLOGY & TOOL USED: Square n and compare string suffix or use modular arithmetic with modulus 10**len(str(n)). Python.

BRIEF DESCRIPTION: Convert both n and n*n to strings and test suffix, or compute n*n % 10^d == n, where d is number of digits in n. The modular method avoids string operations.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _= i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"elapsed time:{elapsed_time:.6f}seconds")
def is_automorphic(n):
    if n < 0:
        return False
    num_digits = len(str(n))
    modulus = 10 ** num_digits
    n_squared = n * n
    return n_squared % modulus == n
print(is_automorphic(12))
print(is_automorphic(3))
print(is_automorphic(56897567))
print(is_automorphic(9))
```

RESULTS ACHIEVED: Correct detection using modular arithmetic (efficient).

```
elapsed time:0.092779seconds
False
False
False
False
```

DIFFICULTY FACED BY STUDENT: Choosing whether to treat negative numbers.

SKILLS ACHIEVED: Modular arithmetic, digit-length computation.

Practical No: 14

Date: 16/11/2025

TITLE: Pronic Number Check (is_pronic)

AIM/OBJECTIVE(s): Implement is_pronic(n) to check if n = k*(k+1) for some integer k.

METHODOLOGY & TOOL USED: Check integer k = floor(sqrt(n)) and test k*(k+1) == n. Python with math.isqrt for integer sqrt.

BRIEF DESCRIPTION: Compute integer sqrt k. If k*(k+1) == n or (k-1)*k == n (for boundary), true. Time O(1).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _= i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"elapsed time:{elapsed_time:.6f}seconds")
def is_pronic(n):
    if n < 0:
        return False
    import math
    k = int(math.sqrt(n))
    return n == k * (k + 1)
print(is_pronic(67))
print(is_pronic(2))
print(is_pronic(789796878))
print(is_pronic(56))
```

RESULTS ACHIEVED: Correct pronic detection in constant time.

```
elapsed time:0.095815seconds
False
True
False
True
```

DIFFICULTY FACED BY STUDENT: Off-by-one around sqrt boundary.

SKILLS ACHIEVED: Integer sqrt usage, edge-case check

Practical No: 15

Date: 16/11/2025

TITLE: Prime Factors (prime_factors)

AIM/OBJECTIVE(s): Implement prime_factors(n) returning list of prime factors of n (with multiplicity).

METHODOLOGY & TOOL USED: Trial division up to sqrt(n) removing factors as found; append remaining prime >1. Python.

BRIEF DESCRIPTION: Repeatedly divide by 2, then iterate odd divisors from 3..sqrt(n) stepping by 2. If remainder n>1 at end, it is prime and appended. Returns sorted list in ascending order (with multiplicity). Complexity O(sqrt(n)).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _= i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"elapsed time:{elapsed_time:.6f}seconds")
def prime_factors(n):
    if n <= 1:
        return []
    factors = []
    d = 2
    while n % d == 0:
        factors.append(d)
        n //= d
    d = 3
    while d * d <= n:
        while n % d == 0:
            factors.append(d)
            n //= d
        d += 2
    if n > 1:
        factors.append(n)
    return factors
print(prime_factors(32))
print(prime_factors(3))
print(prime_factors(6783453))
print(prime_factors(88))
```

RESULTS ACHIEVED: Correct prime factorization for typical inputs; returns empty list for n<2.

```
elapsed time:0.088736seconds
[2, 2, 2, 2, 2]
[3]
[3, 3, 3, 277, 907]
[2, 2, 2, 11]
```

DIFFICULTY FACED BY STUDENT: Efficiency for large n and handling multiplicity.

SKILLS ACHIEVED: Trial division, factor extraction, integer math.

Practical No: 16

Date: 16/11/2025

TITLE : Count Distinct Prime Factors

AIM/OBJECTIVE(s): To implement **count_distinct_prime_factors(n)** which counts how many *unique* prime factors n has.

METHODOLOGY & TOOL USED: Prime checking and factorization using Python.

BRIEF DESCRIPTION: The function identifies prime factors without considering multiplicity.
For example:
60 → factors 2, 3, 5 → 3 distinct prime factors.

```
import time
start_time = time.perf_counter()
for i in range(1000000):
    _ = i * 2
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def count_distinct_prime_factors(n):
    count = 0
    factor = 2
    while factor * factor <= n:
        if n % factor == 0:
            count += 1
            while n % factor == 0:
                n //= factor
        factor += 1
    if n > 1:
        count += 1
    return count
print(count_distinct_prime_factors(6573465))
print(count_distinct_prime_factors(66678))
print(count_distinct_prime_factors(3))
print(count_distinct_prime_factors(345))
```

RESULTS ACHIEVED: Correct unique factor counting.

```
Elapsed time: 0.176361 seconds
3
3
1
3
```

DIFFICULTY FACED BY STUDENT: Avoiding counting the same prime factor multiple times.

SKILLS ACHIEVED: Set-based logic, efficient factor extraction.

Practical No: 17

Date: 16/11/2025

TITLE : Prime Power Check

AIM/OBJECTIVE(s): To implement **is_prime_power(n)** which determines whether n can be expressed as:

METHODOLOGY & TOOL USED: Exponent testing with prime checking.

BRIEF DESCRIPTION: The function tries all possible exponents and checks whether the corresponding root is prime.
Examples:
$8 = 2^3$ (prime power)
$27 = 3^3$ (prime power)
12 is not a prime power.

```
import time
start_time = time.perf_counter()
for i in range(1000000):
    _ = i * 2
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def is_prime_power(n):
    if n < 2:
        return False
    def is_prime(x):
        if x < 2:
            return False
        for i in range(2, int(x**0.5) + 1):
            if x % i == 0:
                return False
        return True
    for k in range(1, int(n.bit_length()) + 1):
        p = round(n ** (1 / k))
        if p ** k == n and is_prime(p):
            return True
    return False
print(is_prime_power(8))
print(is_prime_power(27))
print(is_prime_power(49))
print(is_prime_power(12))
print(is_prime_power(2))
```

RESULTS ACHIEVED: Accurate classification of numbers as prime powers or not.

```
Elapsed time: 0.189886 seconds
True
True
True
False
True
```

DIFFICULTY FACED BY STUDENT: Dealing with precision when testing roots.

SKILLS ACHIEVED: Prime logic, exponentiation reasoning.

Practical No: 18

Date: 16/11/2025

TITLE : Mersenne Prime Check

AIM/OBJECTIVE(s): To implement **is_mersenne_prime(p)**

is prime (given that p is prime).

METHODOLOGY & TOOL USED: Prime testing on Mersenne numbers.

BRIEF DESCRIPTION: Mersenne primes are special primes of the form $2^p - 1$.
The function first ensures p is prime, then checks whether $2^p - 1$ is also prime.

```
import time
start_time = time.perf_counter()
for i in range(1000000):
    _ = i * 2
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def is_mersenne_prime(p):
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True
    if not is_prime(p):
        return False
    mersenne = 2 ** p - 1
    return is_prime(mersenne)
print(is_mersenne_prime(2))
print(is_mersenne_prime(3))
print(is_mersenne_prime(5))
print(is_mersenne_prime(11))
print(is_mersenne_prime(4))
```

RESULTS ACHIEVED: Correct identification of small Mersenne primes like p = 2, 3, 5, 7.

```
Elapsed time: 0.092076 seconds
True
True
True
False
False
```

DIFFICULTY FACED BY STUDENT: Large Mersenne primes require heavy computation.

SKILLS ACHIEVED: Prime checking, exponent math, big integer understanding.

Practical No: 19

Date: 16/11/2025

TITLE : Twin Primes Generator

AIM/OBJECTIVE(s): To implement **twin_primes(limit)** which generates all twin prime pairs up to a given limit.

METHODOLOGY & TOOL USED: Prime sieve and pair checking.

BRIEF DESCRIPTION: Twin primes are primes of the form:

such as (3,5), (5,7), (11,13).
The function generates primes up to limit and checks for prime pairs differing by 2.

```
import time
start_time = time.perf_counter()
for i in range(1000000):
    _ = i * 2
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def twin_primes(limit):
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True
    twins = []
    prev_prime = None
    for num in range(2, limit + 1):
        if is_prime(num):
            if prev_prime is not None and num - prev_prime == 2:
                twins.append((prev_prime, num))
            prev_prime = num
    return twins
print(twin_primes(20))
```

RESULTS ACHIEVED: Correct generation of all twin primes ≤ limit.

```
Elapsed time: 0.096353 seconds
[(3, 5), (5, 7), (11, 13), (17, 19)]
```

DIFFICULTY FACED BY STUDENT: Optimizing prime checking for large ranges.

SKILLS ACHIEVED: Sieve usage, pair analysis, sequence generation.

Practical No: 20

Date: 16/11/2025

TITLE : Number of Divisors Function d(n)

AIM/OBJECTIVE(s): To implement **count_divisors(n)** which returns the total number of positive divisors of n.

METHODOLOGY & TOOL USED: Divisor counting via prime factorization.

BRIEF DESCRIPTION: The function performs prime factorization and applies this formula.

```
import time
start_time = time.perf_counter()
for i in range(1000000):
    _ = i * 2
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def count_divisors(n):
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            if i * i == n:
                count += 1
            else:
                count += 2
        i += 1
    return count
print(count_divisors(6))
print(count_divisors(10))
print(count_divisors(36))
```

RESULTS ACHIEVED: Correct divisor counts for all tested inputs.

```
Elapsed time: 0.097800 seconds
4
4
9
```

DIFFICULTY FACED BY STUDENT: Identifying exponent values for each prime factor.

SKILLS ACHIEVED: Prime factorization, application of divisor formula.

Practical No:21

Date:16/11/2025

TITLE : Calculating the Aliquot Sum of an Integer

AIM/OBJECTIVE(s) : To write an efficient function, aliquot_sum(n), that accepts a positive integer n and returns the sum of all its proper divisors (divisors less than n). This concept is fundamental to classifying numbers as deficient, perfect, or abundant.

METHODOLOGY & TOOL USED : Python 3 (Programming Language),
Iterative Approach with Square Root Optimization (Algorithm Design).

BRIEF DESCRIPTION : The function calculates the aliquot sum by efficiently finding all divisors of n.

1. It initializes the sum to 1 since 1 is a proper divisor of every integer n > 1.
2. It uses an optimization technique: instead of iterating up to n/2 or n-1, it only iterates from 2 up to the square root of n.
3. For every integer 'i' that divides n:
    - i is added to the sum.
    - The paired divisor, n/i, is also calculated and added to the sum.
4. A check ensures that if n is a perfect square ,the square root is only added once.

This methodology significantly reduces computation time compared to a linear approach.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def aliquot_sum(n):
    if n <= 1:
        return 0
    total_sum = 1
    limit = int(n**0.5)
    for i in range(2, limit + 1):
        if n % i == 0:
            total_sum += i
            paired_divisor = n // i
            if paired_divisor != i:
                total_sum += paired_divisor
    return total_sum
print(aliquot_sum(28))
print(aliquot_sum(12))
print(aliquot_sum(10))
```

RESULTS ACHIEVED : The aliquot_sum(n) function correctly returns the sum of proper divisors.

The core logic ensures that for a number like n=36, the pairs (2, 18), (3, 12), (4, 9) are found, and the square root 6 is found only once, resulting in a correct sum of 1 + 2 + 18 + 3 + 12 + 4 + 9 + 6 = 55.

```
Elapsed time:0.193617 seconds
28
16
8
```

DIFFICULTY FACED BY STUDENT : The primary challenge was ensuring the efficiency of the algorithm and handling edge cases:

1. Excluding n: Ensuring the divisor n itself is never added to the sum. (Addressed by initializing sum to 1 and iterating only up to sqrt{n}).
2. Perfect Squares: Correctly handling the case where a number is a perfect square (e.g., 36), to prevent its square root (6) from being counted twice (as both i and n/i). This was resolved with the conditional check if paired_divisor != i

SKILLS ACHIEVED :

- Algorithm Optimization (reducing complexity from O(n) to O(\sqrt{n})).
- Number Theory Concepts (proper divisors, aliquot sum).
- Handling of Edge Cases and Boundary Conditions in loops.
- Implementation of the Modulo Operator (%) for divisibility testing.

Practical No:22

Date: 16/11/2025

TITLE : Amicable Number Identification

AIM/OBJECTIVE(s) : To develop a function, are_amicable(a, b), that determines if two given positive integers a and b constitute an amicable pair.

METHODOLOGY & TOOL USED: Python 3 (Programming Language), Function Decomposition (Utilizing the existing aliquot_sum function), Boolean Logic.

BRIEF DESCRIPTION : The are_amicable(a, b) function performs two primary steps to check the amicability condition:

1. Aliquot Sum Calculation: It calls the existing $O(\sqrt{n})$ optimized aliquot_sum function twice: once for a to get the sum of its proper divisors (S_a), and once for b to get the sum of its proper divisors (S_b).
2. Amicability Check: It evaluates the critical conditions:
    - The sum of proper divisors of a must equal b ($S_a = b$). o The sum of proper divisors of b must equal a ($S_b = a$).
    - The numbers must be distinct ($a \neq b$).

The function returns a boolean True only if all three conditions are met, confirming they form an amicable pair.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def aliquot_sum(n):
    if n <= 1:
        return 0
    total_sum = 1
    limit = int(n**0.5)
    for i in range(2, limit + 1):
        if n % i == 0:
            total_sum += i
            paired_divisor = n // i
            if paired_divisor != i:
                total_sum += paired_divisor
    return total_sum
def are_amicable(a, b):
    if a <= 0 or b <= 0 or a == b:
        return False
    sum_a = aliquot_sum(a)
    is_a_amicable = (sum_a == b)
    sum_b = aliquot_sum(b)
    is_b_amicable = (sum_b == a)
    return is_a_amicable and is_b_amicable
print(f"Are 220 and 284 amicable? {are_amicable(220, 284)}")
print(f"Are 6 and 8 amicable? {are_amicable(6, 8)}")
```

RESULTS ACHIEVED: The function correctly identifies the classic amicable pair (220, 284) and distinguishes them from non-amicable pairs.

```
Elapsed time:0.168615 seconds
Are 220 and 284 amicable? True
Are 6 and 8 amicable? False
```

DIFFICULTY FACED BY STUDENT : The main consideration was Function Decomposition. By properly utilizing the pre-written aliquot_sum function, the are_amicable function became cleaner and easier to read. The only difficulty was ensuring all three required conditions for amicable numbers were explicitly included in the check to prevent misidentification.

SKILLS ACHIEVED :

- Function Decomposition and Reusability (Using aliquot_sum within are_amicable).

- Boolean and Conditional Logic (Implementing the two-way check for amicable pairs).
- Code Clarity and Readability (Separating the concerns of summing divisors and checking the final condition).

Practical No:23

Date:16/11/2025

TITLE : Calculating Multiplicative Persistence

AIM/OBJECTIVE(s) : To develop a function, multiplicative_persistence(n), that counts the number of times the digit multiplication process must be repeated until the number is reduced to a single-digit integer.

METHODOLOGY & TOOL USED : Python 3 (Programming Language),
Iterative Algorithm, Function Composition (Utilizing a helper function for digit multiplication).

BRIEF DESCRIPTION : The solution uses two functions:

1. multiply_digits(n): A helper function that takes an integer, converts it to a string to access individual digits, calculates the product of these digits, and returns the result.
2. multiplicative_persistence(n): The main function. It first handles single-digit inputs (which have a persistence of 0). For numbers $n \ge 10$, it uses a while loop that continues as long as the current_number is two or more digits. In each iteration, it calls multiply_digits, updates the current_number, and increments the persistence_count.

This iterative process guarantees that the problem terminates, as the product of the digits of any number greater than 9 will always be significantly smaller than the original number, eventually leading to a single digit.

```
import time
start_time=time.perf_counter()
for i in range (1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time: {elapsed_time:.6f} seconds")
def multiply_digits(n):
    if n < 0:
                return 0
    product = 1
    for digit_char in str(n):
        product *= int(digit_char)
    return product
def multiplicative_persistence(n):
    if n < 10:
        return 0
    if n <= 0:
        return 0
    persistence_count = 0
    current_number = n
    while current_number >= 10:
        current_number = multiply_digits(current_number)
        persistence_count += 1
    return persistence_count
print(f"Persistence of 39: {multiplicative_persistence(39)}")
print(f"Persistence of 77: {multiplicative_persistence(77)}")
```

RESULTS ACHIEVED : The function correctly calculates the multiplicative persistence for various inputs

```
Elapsed time: 0.178351 seconds
Persistence of 39: 3
Persistence of 77: 4
```

DIFFICULTY FACED BY STUDENT : The main difficulty was handling the transition from number to digits and back to a number. This was solved by using Python's flexibility to convert the integer to a string (str(n)) for easy iteration over the digits, and then converting each character back to an integer (int(digit_char)) for the multiplication.

SKILLS ACHIEVED :

- Iterative Algorithm Design: Implementing a loop structure for repeated transformations.
- Data Type Conversion: Mastering the use of string and integer conversions for digit manipulation.

- Function Composition: Utilizing a clear, small helper function to maintain code clarity and separation of concerns.

Practical No:24

Date: 16/11/2025

TITLE : Highly Composite Number Identification

AIM/OBJECTIVE(s) : To construct a function, is_highly_composite(n), that determines if a positive integer n has a strictly greater number of divisors than any integer i where $i < n$.

METHODOLOGY & TOOL USED : Python 3 (Programming Language), Function Composition, Iterative Comparison Algorithm.

BRIEF DESCRIPTION : The solution requires two functions:

1. count_divisors(n): This helper function calculates the total number of divisors for n using an efficient $O(\sqrt{n})$ technique, which is critical since it is called repeatedly.
2. is_highly_composite(n): The main function.
   - It first calculates the divisor count $D(n)$ for the input number n. o It then iterates through every number i from 1 up to $n-1$. o In each iteration, it calculates $D(i)$ using the helper function.
   - The loop terminates and returns False immediately if it finds any i such that $D(i) \ge D(n)$. o If the loop completes without finding such a number, it returns True.

This direct implementation ensures strict adherence to the mathematical definition of a highly composite number.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def count_divisors(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    count = 2
    limit = int(n**0.5)
    for i in range(2, limit + 1):
        if n % i == 0:
            if i * i != n:
                count += 2
            else:
                count += 1
    return count
def is_highly_composite(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    divisors_of_n = count_divisors(n)
    for i in range(1, n):
        divisors_of_i = count_divisors(i)
        if divisors_of_i >= divisors_of_n:
            return False
    return True
print(f"Is 12 highly composite? {is_highly_composite(12)}")
print(f"Is 18 highly composite? {is_highly_composite(18)}")
print(f"Is 24 highly composite? {is_highly_composite(24)}")
```

RESULTS ACHIEVED : The function correctly identifies whether a number is highly composite.

```
Elapsed time:0.159990 seconds
Is 12 highly composite? True
Is 18 highly composite? False
Is 24 highly composite? True
```

DIFFICULTY FACED BY STUDENT : The primary difficulty is

Computational Complexity. Because the algorithm must check all $i < n$ and each check requires $O(\sqrt{i})$ time, the total complexity approaches $O(n \cdot \sqrt{n})$, which is slow for large values of n. This iterative comparison is necessary to ensure strict adherence to the definition.

SKILLS ACHIEVED :

- Algorithm Efficiency: Implementing the $O(\sqrt{n})$ divisor counting method.
- Definition Implementation: Translating a complex mathematical definition into a precise computational loop structure.
- Early Exit Optimization: Using an immediate return False upon finding a counter-example to save computation time.

Practical No:25

Date:16/11/2025

TITLE : Modular Exponentiation using Binary Exponentiation

AIM/OBJECTIVE(s) : To implement an efficient function, mod_exp(base, exponent, modulus), that calculates the value of $(\text{base}^{\text{exponent}}) \pmod{\text{modulus}}$ while preventing intermediate results from growing prohibitively large, typically achieving $O(\log(\text{exponent}))$ time complexity.

METHODOLOGY & TOOL USED : Python 3 (Programming Language), Binary Exponentiation Algorithm (Exponentiation by Squaring), Modular Arithmetic Properties.

BRIEF DESCRIPTION : The mod_exp function uses the Binary Exponentiation technique, which exploits the binary representation of the exponent. Instead of performing exponent number of multiplications, it performs only $\log_2(\text{exponent})$ squarings.

The core properties of modular arithmetic are used throughout the process:

- $(A \cdot B) \pmod{M} = ((A \pmod{M}) \cdot (B \pmod{M})) \pmod{M}$

The algorithm works as follows:

1. Initialize result = 1 and reduce base modulo modulus.
2. Iterate while the exponent is greater than 0.
3. In each step, if the exponent is odd (meaning the current power of two is present in the exponent's binary form), we incorporate that power into the result: result = (result * base) % modulus.
4. The base is squared in every step: base = (base * base) % modulus.
5. The exponent is halved (exponent //= 2), effectively checking the next bit.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def mod_exp(base, exponent, modulus):
    if modulus <= 0:
        raise ValueError("Modulus must be a positive integer.")
    if exponent < 0:
        raise ValueError("Exponent must be a non-negative integer for this operation.")
    if exponent == 0:
        return 1 % modulus
    base %= modulus
    result = 1
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exponent //= 2
    return result
print(f"(5^3) % 7 = {mod_exp(5, 3, 7)}")
print(f"(3^10) % 11 = {mod_exp(3, 10, 11)}")
print(f"(2^100) % 13 = {mod_exp(2, 100, 13)}")
```

RESULTS ACHIEVED : The function correctly and efficiently handles large exponents by keeping all intermediate products within the scale of the modulus.

```
Elapsed time:0.213199 seconds
(5^3) % 7 = 6
(3^10) % 11 = 1
(2^100) % 13 = 3
```

DIFFICULTY FACED BY STUDENT : The primary difficulty was understanding the Binary Exponentiation concept itself, which requires associating the current base with a power of 2 and correlating the odd/even check of the exponent with its binary bits. Once the iterative structure was established, maintaining the modulo operation (% modulus) on *every* multiplication was crucial to prevent overflow, which was the final critical implementation detail.

SKILLS ACHIEVED :

- Advanced Algorithm Implementation: Mastering the Binary Exponentiation method ($O(\log n)$).
- Modular Arithmetic: Correctly applying modular properties to reduce intermediate values.
- Bitwise Operations (Implicit): Understanding how integer division by 2 and the modulo 2 check relates to the binary representation of a number.

Practical No:26

Date: 16/11/2025

TITLE : Modular Multiplicative Inverse using Extended Euclidean Algorithm

AIM/OBJECTIVE(s) : To implement a function, mod_inverse(a, m), that finds the integer x such that $(a \cdot x) \equiv 1 \pmod{m}$. This inverse is critical for performing division operations within modular arithmetic systems.

METHODOLOGY & TOOL USED : Python 3 (Programming Language), Extended Euclidean Algorithm (EEA) (Primary Algorithm), Euclidean Algorithm (Helper Function).

BRIEF DESCRIPTION : The Modular Multiplicative Inverse x exists if and only if a and m are coprime (i.e., their greatest common divisor $\text{gcd}(a, m) = 1$).

The solution relies on the Extended Euclidean Algorithm (EEA), which finds integers x and y satisfying Bézout's Identity:

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

1. extended_gcd(a, b): This recursive helper function is implemented to return (gcd, x, y).
2. mod_inverse(a, m):
    - It calls extended_gcd(a, m).
    - If $\text{gcd}(a, m) \neq 1$, the inverse does not exist, and the function returns None.
    - If $\text{gcd}(a, m) = 1$, Bézout's identity becomes $a \cdot x + m \cdot y = 1$. Taking this equation modulo m gives: o Thus, the Bézout coefficient x is the modular inverse. The function then returns $x \pmod{m}$ to ensure a positive result within the range $[0, m-1]$.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    gcd_val, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd_val, x, y
def mod_inverse(a, m):
    gcd_val, x, y = extended_gcd(a, m)
    if gcd_val != 1:
        return None
    else:
        return x % m
print(f"Inverse of 3 mod 11: {mod_inverse(3, 11)}")
print(f"Inverse of 2 mod 4: {mod_inverse(2, 4)}")
```

RESULTS ACHIEVED : The function correctly finds the modular inverse when it exists, and handles cases where it does not.

```
Elapsed time:0.165218 seconds
Inverse of 3 mod 11: 4
Inverse of 2 mod 4: None
```

DIFFICULTY FACED BY STUDENT : The primary difficulty was the implementation and understanding of the recursive relationship in the Extended Euclidean Algorithm to correctly calculate the Bézout coefficients (x, y). Ensuring the final result x is normalized to be positive by calculating $x \pmod{m}$ was another key detail that required careful attention.

SKILLS ACHIEVED :

- Extended Euclidean Algorithm: Implementation of a recursive number theory algorithm.
- Modular Arithmetic: Understanding the core requirement for the existence of the inverse ($\text{gcd}(a, m) = 1$).
- Bézout's Identity: Relating the results of the EEA to the solution of the modular inverse problem.

Practical No:27

Date: 16/11/2025

TITLE : Chinese Remainder Theorem (CRT) Solver

AIM/OBJECTIVE(s) : To develop a function, crt(remainders, moduli), that solves a system of simultaneous linear congruences, $x \equiv r_i \pmod{m_i}$, using the constructive proof of the Chinese Remainder Theorem, assuming the moduli m_i are pairwise coprime.

METHODOLOGY & TOOL USED : Python 3 (Programming Language), Chinese Remainder Theorem Constructive Algorithm, Extended Euclidean Algorithm (EEA), Modular Multiplicative Inverse**.**

BRIEF DESCRIPTION : The function solves for the unique solution. The solution is found by constructing x according to the formula:

The implementation uses two essential helper functions:

1. extended_gcd: Calculates $\text{gcd}(a, m)$ and the Bézout coefficients.
2. mod_inverse: Uses EEA to find y_i.

The main crt function iteratively calculates M_i and its inverse y_i for each congruence and sums the product terms $r_i \cdot M_i \cdot y_i$ to find the final solution x.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    gcd_val, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd_val, x, y
def mod_inverse(a, m):
    gcd_val, x, y = extended_gcd(a, m)
    if gcd_val != 1:
        raise ValueError(f"Modular inverse does not exist: {a} and {m} are not coprime (gcd={gcd_val})")
    else:
        return x % m
def crt(remainders, moduli):
    if len(remainders) != len(moduli) or not remainders:
        raise ValueError("Input lists must have the same length and not be empty.")
    M = 1
    for m in moduli:
        M *= m
    solution_x = 0
    for r_i, m_i in zip(remainders, moduli):
        M_i = M // m_i
        try:
            y_i = mod_inverse(M_i, m_i)
        except ValueError as e:
            raise ValueError(f"CRT requirement failed: Moduli must be pairwise coprime. {e}")
        term = r_i * M_i * y_i
        solution_x += term
    return solution_x % M
print(f"Solution x: {crt([2, 3, 2], [3, 5, 7])}")
```

RESULTS ACHIEVED : The function correctly solves systems of congruences

```
Elapsed time:0.197884 seconds
Solution x: 23
```

DIFFICULTY FACED BY STUDENT : The main challenge was the heavy reliance on prerequisite functions (EEA and Modulo Inverse) and ensuring the pairwise coprime condition was handled implicitly through the inverse calculation (as mod_inverse will fail if $\text{gcd}(M_i, m_i) \neq 1$). Structuring the iterative loop to correctly calculate and sum the three components (r_i, M_i, y_i) for each congruence without error was critical.

SKILLS ACHIEVED :

- Advanced Number Theory Implementation: Solving a system of linear congruences.
- Algorithm Decomposition: Combining EEA, Modulo Inverse, and the CRT formula into a single robust solver.
- Iterative and Accumulation Logic: Correctly building the final solution through iterative summation.

- Constraint Handling: Understanding and implicitly verifying the requirement for pairwise coprime moduli.

Practical No:28

Date: 16/11/2025

TITLE : Check for Quadratic Residue

AIM/OBJECTIVE(s) : To implement a Python function is_quadratic_residue(a, p) that determines if the quadratic congruence $x^2 \equiv a \pmod{p}$ has a solution.

METHODOLOGY & TOOL USED : Brute-force search of all residues, Python.

BRIEF DESCRIPTION : A number a is defined as a quadratic residue modulo p if there exists an integer x such that $x^2 \equiv a \pmod{p}$. If no such x exists, a is a quadratic nonresidue.

The implemented function utilizes a direct, brute-force approach, which is simple and effective for small moduli p. The logic is as follows:

1. The function first normalizes a by calculating $a \pmod{p}$.
2. It then iterates through all possible integer values for x in the complete residue system, $\{0, 1, 2, \dots, p-1\}$.
3. In each iteration, it calculates $x^2 \pmod{p}$.
4. If $x^2 \pmod{p}$ is found to be equal to $a \pmod{p}$ for any x, the function immediately returns True.
5. If the loop completes without finding a match, the function returns False.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def is_quadratic_residue(a: int, p: int) -> bool:
    if p <= 1:
        print(f"Error: Modulus p must be greater than 1. Received p={p}")
        return False
    a_mod_p = a % p
    for x in range(p):
        x_squared_mod_p = (x * x) % p
        if x_squared_mod_p == a_mod_p:
            return True
    return False
a1, p1 = 10, 13
print(f"Is {a1} a quadratic residue modulo {p1}? -> {is_quadratic_residue(a1, p1)}")
a2, p2 = 5, 7
print(f"Is {a2} a quadratic residue modulo {p2}? -> {is_quadratic_residue(a2, p2)}")
```

RESULTS ACHIEVED : The implemented function successfully checks for quadratic residues for the given test cases:

```
Elapsed time:0.174189 seconds
Is 10 a quadratic residue modulo 13? -> True
Is 5 a quadratic residue modulo 7? -> False
```

DIFFICULTY FACED BY STUDENT : For large prime moduli p, the brute-force method has a time complexity of $O(p)$, which becomes computationally expensive. An optimized approach using Euler's Criterion ($a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}$) would be necessary for practical applications with large primes.

SKILLS ACHIEVED :

1. Understanding the definition of quadratic residues in Number Theory.
2. Implementing modular arithmetic operations.
3. Developing iterative algorithms for mathematical congruence problems.

Practical No:29

Date: 16/11/2025

TITLE : Multiplicative Order of an Integer Modulo n

AIM/OBJECTIVE(s) : To implement a Python function order_mod(a, n) that calculates the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$, or returns -1 if no such order exists.

METHODOLOGY & TOOL USED : Direct iterative search with modular exponentiation, Python (using the math library for the greatest common divisor function).

BRIEF DESCRIPTION : The multiplicative order of a modulo n, denoted $\text{ord}_n(a)$, is only defined if $\gcd(a, n) = 1$. If they are coprime, the set of powers $\{a^k \pmod{n}\}$ forms a cyclic group. The function order_mod(a, n) first checks for the coprimality condition using math.gcd(). If $\gcd(a, n) \neq 1$, it immediately returns -1.

If the order is defined, the function iteratively calculates $a^k \pmod{n}$ starting from $k=1$. To do this efficiently, it uses the identity $a^{k+1} \equiv (a^k \cdot a) \pmod{n}$, avoiding repeated base exponentiation.
The loop terminates and returns k as soon as the result is 1. By Euler's Totient Theorem, this process is guaranteed to terminate for $k \le \phi(n)$, where $\phi(n)$ is Euler's totient function.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def order_mod(a: int, n: int) -> int:
    if n <= 1:
        return -1
    if math.gcd(a, n) != 1:
        return -1
    a = a % n
    if a == 0:
        return -1
    k = 1
    current_power = a
    while k <= n:
        if current_power == 1:
            return k
        current_power = (current_power * a) % n
        k += 1
    return -1
a1, n1 = 3, 7
print(f"Order of {a1} mod {n1}: {order_mod(a1, n1)}")
a2, n2 = 2, 8
print(f"Order of {a2} mod {n2}: {order_mod(a2, n2)}")
a3, n3 = 10, 13
print(f"Order of {a3} mod {n3}: {order_mod(a3, n3)}")
a4, n4 = 2, 5
print(f"Order of {a4} mod {n4}: {order_mod(a4, n4)}")
```

RESULTS ACHIEVED : The implemented function successfully calculates the multiplicative order for various inputs:

```
Elapsed time:0.176114 seconds
Order of 3 mod 7: 6
Order of 2 mod 8: -1
Order of 10 mod 13: 6
Order of 2 mod 5: 4
```

DIFFICULTY FACED BY STUDENT : The primary complexity in implementing this function accurately is ensuring the check for the coprimality condition ($\gcd(a, n) = 1$) is correctly implemented, as the order is undefined otherwise. Additionally, the need for efficient

modular exponentiation was addressed by iteratively updating the power (current_power = (current_power * a) % n) instead of recalculating a^k from scratch in each loop iteration.

SKILLS ACHIEVED :

1. Modular Arithmetic and Exponentiation.
2. Understanding of the Multiplicative Order and Coprimality ($\gcd$).
3. Implementing iterative algorithms for Number Theory concepts.
4. Using the math.gcd function in Python.

Practical No:30

Date:16/11/2025

TITLE : Fibonacci Prime Check

AIM/OBJECTIVE(s) : To implement a Python function is_fibonacci_prime(n) that determines if a positive integer n is both a Fibonacci number and a prime number.

METHODOLOGY & TOOL USED : Combined check using primality test and a mathematical identity for Fibonacci numbers, Python (using the math library).

BRIEF DESCRIPTION : A number n is classified as a Fibonacci Prime if it satisfies two independent conditions: it must be a prime number, and it must belong to the Fibonacci sequence (F0=0, F1=1, F2=1, F3=2, F4=3, F5=5, ...).

The function is_fibonacci_prime(n) is built on two helper functions:

1. is_prime(n): Uses an optimized trial division algorithm, checking divisibility only up to $\sqrt{n}$ and skipping multiples of 2 and 3, to efficiently determine if n is prime.
2. is_fibonacci_number(n): Determines if n belongs to the Fibonacci sequence using the identity that n is a Fibonacci number if and only if $5n^2 + 4$ or $5n^2 - 4$ is a perfect square. This avoids the high cost of generating the sequence up to n.

The main function combines these checks: it returns True only if n passes both the primality test and the Fibonacci number test; otherwise, it returns False.

```python
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def is_prime(n: int) -> bool:
    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_fibonacci_number(n: int) -> bool:
    if n < 0:
        return False
    if n == 0 or n == 1:
        return True
    def is_perfect_square(k):
        if k < 0:
            return False
        root = int(math.isqrt(k))
        return root * root == k
    expr1 = 5 * n * n + 4
    expr2 = 5 * n * n - 4
    return is_perfect_square(expr1) or is_perfect_square(expr2)
def is_fibonacci_prime(n: int) -> bool:
    if not is_prime(n):
        return False
    if not is_fibonacci_number(n):
        return False
    return True
n1 = 13
print(f"Is {n1} a Fibonacci Prime? -> {is_fibonacci_prime(n1)}")
n2 = 21
print(f"Is {n2} a Fibonacci Prime? -> {is_fibonacci_prime(n2)}")
n3 = 11
print(f"Is {n3} a Fibonacci Prime? -> {is_fibonacci_prime(n3)}")
```

RESULTS ACHIEVED : The implemented function successfully checks the combined property for the given test cases:

```
Elapsed time:0.178353 seconds
Is 13 a Fibonacci Prime? -> True
Is 21 a Fibonacci Prime? -> False
Is 11 a Fibonacci Prime? -> False
```

DIFFICULTY FACED BY STUDENT : The main challenge was choosing an efficient method for the Fibonacci number check. Generating the sequence iteratively becomes slow for large n. The use of the mathematical identity ($5n^2 \pm 4$ is a square) provided a $O(\log n)$ or $O(1)$ solution (depending on the square root implementation), which is significantly faster than the $O(n)$ or $O(\log n)$ required by sequence generation.

SKILLS ACHIEVED :

1. Implementing efficient primality tests.
2. Applying mathematical identities (Binet's formula related property) for sequence membership checks.
3. Combining multiple specialized checks into a single classification function.
4. Using the math.isqrt and other math library functions in Python.

Practical No: 31

Date:16/11/2025

TITLE : Lucas Numbers Generator

AIM/OBJECTIVE(s) : To implement a Python function lucas_sequence(n) that generates the first n numbers in the Lucas sequence, which is defined by $L_k = L_{k-1} + L_{k-2}$ with initial values $L_0 = 2$ and $L_1 = 1$.

METHODOLOGY & TOOL USED : Iterative sequence generation using list manipulation, Python.

BRIEF DESCRIPTION : The Lucas sequence is closely related to the Fibonacci sequence but begins with different initial conditions. The sequence starts $2, 1, 3, 4, 7, 11, 18, \dots$.

The function lucas_sequence(n) handles the small cases $n=0, n=1, n=2$ directly. For $n > 2$, it initializes a list with the first two terms $[2, 1]$. It then enters an iterative loop that runs $n-2$ times. In each iteration, the next Lucas number is calculated by summing the last two elements of the list (using the standard Fibonacci recurrence relation) and appending the new term to the list. This method has a time complexity of $O(n)$, which is efficient for generating the sequence.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def lucas_sequence(n: int) -> list[int]:
    if n <= 0:
        return []
    if n == 1:
        return [2]
    lucas_list = [2, 1]
    if n == 2:
        return lucas_list
    for _ in range(2, n):
        next_lucas = lucas_list[-1] + lucas_list[-2]
        lucas_list.append(next_lucas)
    return lucas_list
n1 = 5
print(f"First {n1} Lucas numbers: {lucas_sequence(n1)}")
n2 = 10
print(f"First {n2} Lucas numbers: {lucas_sequence(n2)}")
n3 = 1
print(f"First {n3} Lucas number: {lucas_sequence(n3)}")
```

RESULTS ACHIEVED : The function correctly generates the requested number of Lucas numbers

```
Elapsed time:0.092613 seconds
First 5 Lucas numbers: [2, 1, 3, 4, 7]
First 10 Lucas numbers: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
First 1 Lucas number: [2]
```

DIFFICULTY FACED BY STUDENT : Ensuring the correct handling of the edge cases, particularly $n=1$ and $n=2$, was crucial due to the sequence starting with two distinct initial values (2 and 1). Standard sequence generators often start with $F_0=0, F_1=1$, requiring careful modification to use $L_0=2, L_1=1$.

SKILLS ACHIEVED :

1. Understanding and implementing linear recurrence relations.
2. Handling edge cases in sequence generation.
3. Efficient list manipulation for iterative calculations in Python.

Practical No:32

Date:16/11/2025

TITLE : Perfect Powers Check

AIM/OBJECTIVE(s) : To implement a Python function is_perfect_power(n) that checks if a number n can be expressed as a^b, where the base $a > 0$ and the exponent $b > 1$.

METHODOLOGY & TOOL USED : Exhaustive search for the exponent b combined with root calculation, Python (using the math library).

BRIEF DESCRIPTION : A number n is a perfect power if it is the product of equal factors, i.e., $n = a^b$.

The implemented function iterates over all possible integer exponents b, starting from $b=2$. Since $a \ge 2$, the maximum possible value for the exponent b is $\lfloor \log_2(n) \rfloor$.

For each exponent b:

1. The function estimates the base a by calculating the b-th root of n, $a \approx n^{1/b}$.
2. The estimated base is rounded to the nearest integer.
3. The function verifies if the rounded integer base a, when raised to the power b, exactly equals n. If this holds, n is a perfect power, and the function returns True.

If no such base a is found for any exponent b within the maximum range, the function returns False. The complexity is roughly $O(\log^2 n)$ or better, making it efficient for typical inputs.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def is_perfect_power(n: int) -> bool:
    if n <= 3:
        return n == 1
    max_b = math.floor(math.log2(n))
    for b in range(2, max_b + 1):
        try:
            a = round(n ** (1 / b))
        except OverflowError:
            continue
        if a > 1 and a**b == n:
            return True
    return False
n1 = 8
print(f"Is {n1} a perfect power? -> {is_perfect_power(n1)}")
n2 = 125
print(f"Is {n2} a perfect power? -> {is_perfect_power(n2)}")
n3 = 10
print(f"Is {n3} a perfect power? -> {is_perfect_power(n3)}")
n4 = 16
print(f"Is {n4} a perfect power? -> {is_perfect_power(n4)}")
n5 = 1
print(f"Is {n5} a perfect power? -> {is_perfect_power(n5)}")
```

RESULTS ACHIEVED : The function correctly identifies perfect powers

```
Elapsed time:0.090231 seconds
Is 8 a perfect power? -> True
Is 125 a perfect power? -> True
Is 10 a perfect power? -> False
Is 16 a perfect power? -> True
Is 1 a perfect power? -> True
```

DIFFICULTY FACED BY STUDENT : Handling floating-point precision when calculating $n^{1/b}$ was a subtle point. Direct comparison of $n^{1/b}$ to an integer can fail due to precision errors. This was resolved by rounding the calculated root and checking if round(n**(1/b))**b == n.

SKILLS ACHIEVED :

1. Understanding logarithmic bounds in search algorithms.

2. Working with floating-point calculations and handling precision for integer checks.
3. Implementing a number theoretic property involving exponents and bases.

Practical No: 33

Date:16/11/2025

TITLE : Collatz Sequence Length

AIM/OBJECTIVE(s) : To implement a Python function `collatz_length(n)` that returns the number of steps required for a positive integer n to reach 1 under the rules of the Collatz conjecture.

METHODOLOGY & TOOL USED : Iterative simulation of the Collatz process, Python.

BRIEF DESCRIPTION : The **Collatz conjecture** is a famous unsolved problem in mathematics. It involves a sequence defined by two rules for a positive integer n:

The conjecture posits that this sequence always eventually reaches 1, regardless of the starting number n.

The function collatz_length(n) simulates this process. It uses a while loop that continues as long as the current number is not 1. Inside the loop, it checks the parity of the current number to apply the correct rule (division by 2 or $3n+1$) and increments a step counter. The process stops when the terminal value of 1 is reached, and the total step count is returned.

```python
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def collatz_length(n: int) -> int:
    if n <= 0:
        return 0
    if n == 1:
        return 0
    steps = 0
    current_n = n
    while current_n != 1:
        if current_n % 2 == 0:
            current_n = current_n // 2
        else:
            current_n = 3 * current_n + 1
        steps += 1
    return steps
n1 = 6
print(f"Collatz length for {n1}: {collatz_length(n1)}")
n2 = 1
print(f"Collatz length for {n2}: {collatz_length(n2)}")
n3 = 27
print(f"Collatz length for {n3}: {collatz_length(n3)}")
n4 = 15
print(f"Collatz length for {n4}: {collatz_length(n4)}")
```

RESULTS ACHIEVED : The function correctly calculates the path length for various inputs

```
Elapsed time:0.092614 seconds
Collatz length for 6: 8
Collatz length for 1: 0
Collatz length for 27: 111
Collatz length for 15: 17
```

DIFFICULTY FACED BY STUDENT : While the implementation is straightforward, a critical consideration for large inputs is the potential for the intermediate value n to become very large (known as the 'peak' value) before falling back to 1. This could lead to integer overflow in languages with fixed-size integers, but standard Python integers handle arbitrary size, mitigating this issue.

SKILLS ACHIEVED :

1. Implementing mathematical recursive/iterative processes with conditional logic.
2. Understanding and simulating number theoretic conjectures.
3. Using efficient integer division (//) to ensure integer results.

Practical No: 34

Date: 16/11/2025

TITLE : Polygonal Numbers Generator

AIM/OBJECTIVE(s) : To implement a Python function `polygonal_number(s, n)` that returns the n-th number in the sequence of s-gonal numbers.

METHODOLOGY & TOOL USED : Direct mathematical formula implementation, Python.

BRIEF DESCRIPTION : Polygonal numbers are figurative numbers represented by points arranged in the shape of a regular polygon. Examples include triangular numbers ($s=3$), square numbers ($s=4$), and pentagonal numbers ($s=5$).

The function `polygonal_number(s, n)` implements this formula directly. It first performs validation to ensure $s \ge 3$ and $n \ge 1$, which are the conditions for a valid polygonal number sequence. The calculation is broken down into simple steps to ensure accurate integer arithmetic, using integer division (`//`) since the final result is always an integer.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def polygonal_number(s: int, n: int) -> int:
    if s < 3 or n < 1:
        return -1
    term1 = (s - 2) * n
    term2 = (s - 4)
    inner_term = term1 - term2
    result = (n * inner_term) // 2
    return result
s1, n1 = 3, 4
print(f"The {n1}-th {s1}-gonal (Triangular) number: {polygonal_number(s1, n1)}")
s2, n2 = 4, 5
print(f"The {n2}-th {s2}-gonal (Square) number: {polygonal_number(s2, n2)}")
s3, n3 = 5, 3
print(f"The {n3}-th {s3}-gonal (Pentagonal) number: {polygonal_number(s3, n3)}")
s4, n4 = 6, 4
print(f"The {n4}-th {s4}-gonal (Hexagonal) number: {polygonal_number(s4, n4)}")
```

RESULTS ACHIEVED : The function accurately calculates the required polygonal numbers

```
Elapsed time:0.088222 seconds
The 4-th 3-gonal (Triangular) number: 10
The 5-th 4-gonal (Square) number: 25
The 3-th 5-gonal (Pentagonal) number: 12
The 4-th 6-gonal (Hexagonal) number: 28
```

DIFFICULTY FACED BY STUDENT : The main difficulty lay in translating the fractional formula $\frac{n}{2} \cdot (\dots)$ into accurate integer arithmetic in code, ensuring that floating-point intermediate values are avoided or handled correctly. Using the equivalent integer form $\frac{n \cdot [(s-2)n - (s-4)]}{2}$ with integer division (//) solved this.

SKILLS ACHIEVED :

1. Translating complex mathematical formulas into robust code.
2. Understanding and calculating figurative numbers sequences.
3. Implementing accurate integer arithmetic for closed-form expressions.

Practical No:35

Date:16/11/2025

TITLE : Carmichael Number Check

AIM/OBJECTIVE(s) : To implement a Python function is_carmichael(n) that checks if a composite number n satisfies the Carmichael property: $a^{n-1} \equiv 1 \pmod{n}$ for all integers a coprime to n.

METHODOLOGY & TOOL USED : Combined primality/compositeness check and iterative modular exponentiation for a set of bases, Python (using the math library and built-in pow).

BRIEF DESCRIPTION : A Carmichael number is a composite number n that acts like a prime number with respect to Fermat's Little Theorem. It satisfies $a^{n-1} \equiv 1 \pmod{n}$ for all integers a such that $\gcd(a, n) = 1$. The smallest Carmichael number is 561.

The function is_carmichael(n) first ensures that n is composite (not prime and $n > 1$) and greater than the smallest known Carmichael number (561, for a quick filter).

It then performs a probabilistic check by testing the Carmichael property for a small, robust set of bases a (e.g., $a \in \{2, 3, 5, 7, \dots\}$). For each base a that is coprime to n:

1. It calculates $a^{n-1} \pmod{n}$ efficiently using Python's built-in pow(a, n - 1, n).
2. If the result is not 1, the number n is immediately disqualified and the function returns False.

If n passes the test for all small, coprime bases, it is highly likely to be a Carmichael number, and the function returns True. (A fully deterministic check requires factorization, which is computationally infeasible for large numbers).

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def is_prime(n: int) -> bool:
    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_carmichael(n: int) -> bool:
    if n <= 560 or is_prime(n):
        return False
    bases_to_check = [2, 3, 5, 7, 11, 13]
    for a in bases_to_check:
        if a >= n:
            break
        if math.gcd(a, n) == 1:
            if pow(a, n - 1, n) != 1:
                return False
    return True


n1 = 561
print(f"Is {n1} a Carmichael number? -> {is_carmichael(n1)}")
n2 = 1105
print(f"Is {n2} a Carmichael number? -> {is_carmichael(n2)}")
n3 = 91
print(f"Is {n3} a Carmichael number? -> {is_carmichael(n3)}")
n4 = 17
print(f"Is {n4} a Carmichael number? -> {is_carmichael(n4)}")
```

RESULTS ACHIEVED : The function correctly identifies the two smallest Carmichael numbers:

```
Elapsed time:0.094066 seconds
Is 561 a Carmichael number? -> True
Is 1105 a Carmichael number? -> True
Is 91 a Carmichael number? -> False
Is 17 a Carmichael number? -> False
```

DIFFICULTY FACED BY STUDENT : Implementing the core check requires efficient modular exponentiation for large n. This was solved by leveraging the optimized three-argument version of Python's pow(base, exponent, modulus). A theoretical difficulty is the need to prove the property for all coprime a, which was addressed by using a strong, probabilistic test common in computational number theory.

SKILLS ACHIEVED :

1. Understanding and implementing the definition of Carmichael numbers.
2. Utilizing advanced built-in functions for optimized modular exponentiation.
3. Implementing composite number tests derived from number theory (Fermat pseudoprimes).

VIT BHOPAL

Practical No: 36

Date: 16/11/2025

TITLE : Probabilistic Primality Test (Miller-Rabin)

AIM/OBJECTIVE(s) : To implement the **Miller-Rabin primality test** function, is_prime_miller_rabin(n, k), which uses k random bases to probabilistically determine if a number n is prime.

METHODOLOGY & TOOL USED : Probabilistic testing based on modular exponentiation and the properties of quadratic residues, Python (using the random module and built-in pow).

BRIEF DESCRIPTION : The **Miller-Rabin test** is one of the most widely used probabilistic primality tests, significantly more reliable than the simpler Fermat primality test, as it can successfully identify Carmichael numbers as composite.

The test relies on factoring $n-1$ as $2^s \cdot d$, where d is odd. For a given base a, n is considered likely prime if either:

1. $a^d \equiv 1 \pmod{n}$, OR
2. $a^{2^r d} \equiv -1 \pmod{n}$ for some $0 \le r < s$.

The function is_prime_miller_rabin(n, k) runs the core test k times, selecting a new random base a in the range $[2, n-2]$ for each round.
If the test fails for any base a (meaning a is a **strong witness** to n's compositeness), the function immediately returns False. If it passes all k rounds, the probability of n being composite is less than 4^{-k}, making it "likely prime."

```python
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import random
def miller_rabin_test(a: int, n: int) -> bool:
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1
    x = pow(a, d, n)
    if x == 1 or x == n - 1:
        return True
    for _ in range(s - 1):
        x = pow(x, 2, n)
        if x == n - 1:
            return True
    return False
def is_prime_miller_rabin(n: int, k: int = 10) -> bool:
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    for _ in range(k):
        a = random.randint(2, n - 2)
        if not miller_rabin_test(a, n):
            return False
    return True
n1 = 17
print(f"Is {n1} likely prime (k=5)? -> {is_prime_miller_rabin(n1, 5)}")
n2 = 91
print(f"Is {n2} likely prime (k=5)? -> {is_prime_miller_rabin(n2, 5)}")
n3 = 2147483647
print(f"Is {n3} likely prime (k=10)? -> {is_prime_miller_rabin(n3, 10)}")
n4 = 561
print(f"Is {n4} likely prime (k=3)? -> {is_prime_miller_rabin(n4, 3)}")
```

RESULTS ACHIEVED : The function demonstrates the probabilistic power of the test.

```
Elapsed time:0.093379 seconds
Is 17 likely prime (k=5)? -> True
Is 91 likely prime (k=5)? -> False
Is 2147483647 likely prime (k=10)? -> True
Is 561 likely prime (k=3)? -> False
```

DIFFICULTY FACED BY STUDENT : The main conceptual difficulty is the decomposition of $n-1$ into $2^s \cdot d$ and correctly implementing the iterative squaring step within the modular exponentiation. The use of Python's built-in

pow(base, exp, mod) was essential for performing the large modular exponentiation efficiently.

SKILLS ACHIEVED :

1. Understanding and implementing probabilistic cryptographic algorithms.
2. Advanced modular arithmetic and exponentiation.
3. Decomposition of integers and iterative processes (Floyd's cyclefinding logic adapted for the test).

Practical No: 37

Date: 16/11/2025

TITLE : Integer Factorization (Pollard's Rho Algorithm)

AIM/OBJECTIVE(s) : To implement the **Pollard's Rho algorithm**, pollard_rho(n), which efficiently finds a non-trivial factor of a composite number n.

METHODOLOGY & TOOL USED : Floyd's cycle-finding algorithm (tortoise and hare) applied to a pseudo-random sequence defined by $f(x) = (x^2 + c) \pmod{n}$, combined with the Greatest Common Divisor (GCD) calculation, Python.

BRIEF DESCRIPTION : Pollard's Rho algorithm is a specialized factorization method effective for numbers n that have at least one relatively small prime factor p. Its efficiency is proportional to $O(\sqrt{p})$, significantly faster than trial division if p is small.

The algorithm relies on detecting a cycle in the sequence $x_k = f(x_{k-1}) \pmod{n}$. When considered modulo a prime factor p of n, this sequence eventually repeats, forming a shape like the Greek letter ρ. The core steps are:

1. **Sequence Generation:** Define the sequence x_k using the function $f(x) = (x^2 + c) \pmod{n}$, where c is a random constant.
2. **Cycle Detection:** Use **Floyd's cycle-finding algorithm** (tortoise x moves one step, hare y moves two steps).
3. **Factor Discovery:** Calculate the Greatest Common Divisor $g = \gcd(|x-y|, n)$. When the hare catches the tortoise (i.e., $x \equiv y \pmod{p}$), the GCD calculation will likely find p, since $|xy|$ is a multiple of p.

The loop continues until a non-trivial factor $g > 1$ is found.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
import random
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a
def pollard_rho(n: int) -> int:
    if n <= 3:
        return n
    if n % 2 == 0:
        return 2
    x = random.randint(2, n - 1)
    y = x
    c = random.randint(1, n - 1)
    g = 1
    def f(val, n_mod, c_offset):
        return (val * val + c_offset) % n_mod
    while g == 1:
        x = f(x, n, c)
        y = f(y, n, c)
        y = f(y, n, c)
        g = gcd(abs(x - y), n)
        if g == n:
            return pollard_rho(n)
    return g
n1 = 91
print(f"A factor of {n1} is: {pollard_rho(n1)}")
n2 = 8051
print(f"A factor of {n2} is: {pollard_rho(n2)}")
n3 = 24991
print(f"A factor of {n3} is: {pollard_rho(n3)}")
```

RESULTS ACHIEVED : The function successfully returns a factor for the test cases

```
Elapsed time:0.099293 seconds
A factor of 91 is: 13
A factor of 8051 is: 83
A factor of 24991 is: 67
```

DIFFICULTY FACED BY STUDENT : The main challenge is the correct implementation of Floyd's cycle-finding algorithm within the factorization context, ensuring that the modular arithmetic is consistent and that the GCD is calculated on the absolute difference $|x-y|$. Handling the case where the algorithm fails (returns $g=n$) by restarting with new parameters is also critical for robustness.

SKILLS ACHIEVED :

1. Implementing advanced factorization algorithms.
2. Applying iterative sequence generation and modular arithmetic.
3. Utilizing the Greatest Common Divisor (gcd) in a number theoretic context.
4. Understanding and implementing Floyd's cycle-finding technique.

Practical No: 38

Date:16/11/2025

TITLE: Riemann Zeta Function Approximation

AIM/OBJECTIVE(s): To implement a Python function `zeta_approx(s, terms)` that approximates the value of the **Riemann zeta function**, $\zeta(s)$, by summing the first `terms` of its defining infinite series.

METHODOLOGY & TOOL USED: Direct calculation of the infinite series partial sum, Python (using the `math` library for power and constants).

BRIEF DESCRIPTION: The Riemann zeta function, $\zeta(s)$, is a fundamental function in analytic number theory, defined by the infinite series for complex numbers s with $\text {Re}(s) > 1$:

The function zeta_approx (s, terms) calculates a partial sum of this series.

1. **Convergence Check:** It first verifies the convergence condition. If $s \le 1$ (the harmonic series for $s=1$), the function returns ∞ as the series does not converge in this region.
2. **Summation:** It iterates from $k=1$ up to terms, calculating the value of $\frac{1}{k^s}$ for each term and accumulating the result.

This method provides an increasingly accurate approximation as the number of terms increases.

```python
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
import math
def zeta_approx(s: float, terms: int) -> float:
    if s <= 1.0:
        return math.inf
    if terms <= 0:
        return 0.0
    zeta_sum = 0.0
    for k in range(1, terms + 1):
        term = 1.0 / (k ** s)
        zeta_sum += term
    return zeta_sum
s1, terms1 = 2.0, 100
approx1 = zeta_approx(s1, terms1)
exact1 = (math.pi**2) / 6
print(f"ζ({s1}) approx with {terms1} terms: {approx1:.6f} (Exact: {exact1:.6f})")
s2, terms2 = 3.0, 50
approx2 = zeta_approx(s2, terms2)
print(f"ζ({s2}) approx with {terms2} terms: {approx2:.6f}")
s3, terms3 = 1.0, 100
print(f"ζ({s3}) (Harmonic series): {zeta_approx(s3, terms3)}")
```

RESULTS ACHIEVED: The function provides a good approximation of $\zeta(s)$ for values where $s > 1$

```
Elapsed time:0.097342 seconds
ζ(2.0) approx with 100 terms: 1.634984 (Exact: 1.644934)
ζ(3.0) approx with 50 terms: 1.201861
ζ(1.0) (Harmonic series): inf
```

DIFFICULTY FACED BY STUDENT: The primary consideration was handling the **convergence condition** $s > 1$. Failure to check this would lead to incorrect (diverging) results or potential runtime errors if s were complex. Additionally, ensuring that floating-point arithmetic is used for the sum and power calculation is necessary for accurate approximation.

SKILLS ACHIEVED:

1. Implementing approximations of infinite mathematical series.
2. Handling convergence requirements for functions.
3. Advanced mathematical functions and power calculation in Python.

Practical No: 39

Date: 16/11/2025

TITLE: Partition Function $p(n)$ Calculation

AIM/OBJECTIVE(s): To implement a Python function `partition_function(n)` that calculates the number of distinct ways to write a positive integer n as a sum of positive integers, denoted $p(n)$.

METHODOLOGY & TOOL USED: Dynamic Programming (DP) based on the recursive structure of partitions, Python.

BRIEF DESCRIPTION: The **partition function** $p(n)$ counts the number of ways to express n as a sum of positive integers, where the order of the summands does not matter. For example, $p(4) = 5$ because 4 can be partitioned as 4, $3+1$, $2+2$, $2+1+1$, and $1+1+1+1$.

```
import time
start_time=time.perf_counter()
for i in range(1000000):
    _=i*2
end_time=time.perf_counter()
elapsed_time=end_time-start_time
print(f"Elapsed time:{elapsed_time:.6f} seconds")
def partition_function(n: int) -> int:
    if n < 0:
        return 0
    if n == 0:
        return 1
    dp = [0] * (n + 1)
    dp[0] = 1
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            dp[i] += dp[i - j]
    dp = [0] * (n + 1)
    dp[0] = 1
    for j in range(1, n + 1):
        for i in range(j, n + 1):
            dp[i] += dp[i - j]
    return dp[n]
n1 = 4
print(f"Number of partitions of {n1}, p({n1}): {partition_function(n1)}")
n2 = 5
print(f"Number of partitions of {n2}, p({n2}): {partition_function(n2)}")
n3 = 10
print(f"Number of partitions of {n3}, p({n3}): {partition_function(n3)}")
```

RESULTS ACHIEVED: The function correctly calculates the partition numbers for the test cases

```
Elapsed time:0.087080 seconds
Number of partitions of 4, p(4): 5
Number of partitions of 5, p(5): 7
Number of partitions of 10, p(10): 42
```

DIFFICULTY FACED BY STUDENT: The primary difficulty is selecting and correctly implementing the most efficient recurrence relation. While simpler recursive definitions exist, the generating function identity provides a more performant solution suitable for calculating $p(n)$ for many values. Understanding how the DP loop structure corresponds to the inclusion of new part sizes was key.

SKILLS ACHIEVED:

1. Understanding the mathematical concept of integer partitions.
2. Implementing efficient Dynamic Programming algorithms.