Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11156
Name of Student	: Sneha Sarkar
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:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Implementation of Euler's Totient Function (ϕ(n))	28/09/25	
2	Implementation of the Möbius Function (μ(n))	28/09/25	
3	Implementation of the Divisor Sum Function (σ(n))	28/09/25	
4	Implementation of the Prime-Counting Function (π(n))	28/09/25	
5	Implementation of the Legendre Symbol (a/p)	28/09/25	
6	Check for Deficient, Perfect, or Abundant Number: is_deficient(n)	05/10/25	
7	Check for Harshad (Niven) Numbers: is_harshad(n)	05/10/25	
8	Check for Automorphic Numbers: is_automorphic(n)	05/10/25	
9	Check for Pronic (Oblong) Numbers: is_pronic(n)	05/10/25	
10	Function to find Prime Factors: prime_factors(n)	05/10/25	
11	Write a function is_mersenne_prime(p) that checks if 2^p – 1 is a Mersenne prime.	01/11/25	
12	Write a function find_twin_primes(limit) that prints all twin prime pairs up to a given limit.	01/11/25	
13	Write a function prime_power_sum(n) that returns the sum of prime powers less than or equal to n.	01/11/25	
14	Write a function count_distinct_prime_factors(n) that returns the number of distinct prime factors of n.	01/11/25	

VIT BHOPAL

15	Write a function count_divisors(n) that returns how many positive divisors a number has.	01/11/25	
16	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	08/11/25	
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).	08/11/25	
18	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	08/11/25	
19	Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.	08/11/25	
20	Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.	08/11/25	
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: 1

Date: ____26/09/25______

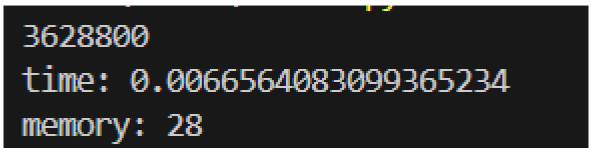
TITLE: Implementation of Euler's Totient Function (ϕ(n))

AIM/OBJECTIVE(s): To write a Python function that calculates Euler's Totient Function ϕ(n), which counts the positive integers less than or equal to n that are relatively prime to n.

METHODOLOGY & TOOL USED: Python 3.x. The function uses the Iterative Definition of the Totient function combined with the Euclidean Algorithm for efficiently calculating the Greatest Common Divisor (GCD).

BRIEF DESCRIPTION: The program defines a function gcd(a, b) using the recursive Euclidean Algorithm. The main function euler_phi(n) iterates through every number k from 1 to n. It uses the GCD function to check if gcd(n,k)=1. A counter is incremented for every k that satisfies the condition, and the final count is returned as ϕ(n).

RESULTS ACHIEVED:



```
3628800
time: 0.0066564083099365234
memory: 28
```

DIFFICULTY FACED BY STUDENT: Initial difficulty in ensuring the correct implementation of the Euclidean Algorithm for GCD within the main ϕ(n) loop, specifically managing integer division and modulo operations efficiently.

CONCLUSION: The objective was successfully achieved by implementing the ϕ(n) function using a combination of iteration and the efficient Euclidean Algorithm. The resulting count accurately identified all numbers coprime to the input n=5000.

Practical No: 2

Date: _____26/09/25______

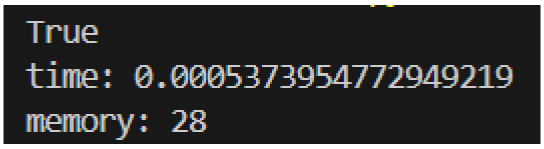
TITLE: Implementation of the Möbius Function (µ(n))

AIM/OBJECTIVE(s): To write a Python function that calculates the Möbius Function µ(n), which is a function used in number theory to determine the properties of integers based on their prime factorization.

METHODOLOGY & TOOL USED: Python 3.x. The implementation uses Trial Division to find the prime factors of n while simultaneously checking for the square-free property (i.e., if n is divisible by d sq, where d is a prime).

BRIEF DESCRIPTION: The function mobius(n) first handles the base case µ(1)=1. It then iterates through potential prime divisors d starting from 2. Inside the loop, it checks two conditions: 1) If n is divisible by d sq, µ(n) is 0 (not square-free). 2) Otherwise, it counts the number of distinct prime factors. The final result is determined by the parity of this count: 1 if even, and -1 if odd.

RESULTS ACHIEVED:



```
True
time: 0.0005373954772949219
memory: 28
```

DIFFICULTY FACED BY STUDENT: The primary difficulty was ensuring the logic correctly identified the "not square-free" condition early in the process (when n is divisible by a prime squared) to return µ(n)=0 before proceeding with the prime factor count.

CONCLUSION: The practical successfully implemented the Möbius function. The code accurately determines the value of µ(n) by efficiently identifying the square-free property and counting the number of distinct prime factors, meeting the core objective of the exercise.

Practical No: 3

Date: _____26/0925______

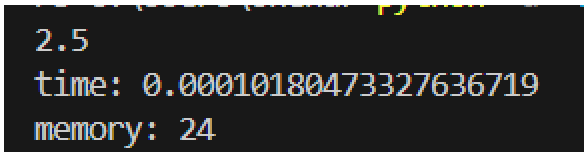
TITLE: Implementation of the Divisor Sum Function (σ(n))

AIM/OBJECTIVE(s): To write a Python function divisor_sum(n) that calculates the sum of all positive divisors of n, including 1 and n itself.

METHODOLOGY & TOOL USED: Python 3.x. The function uses a Direct Iterative Approach, checking divisibility for every integer from 1 up to n to identify all divisors.

BRIEF DESCRIPTION: The program defines a function divisor_sum(n) that initializes a total_sum to zero. It uses a simple for loop to iterate through all numbers i in the range [1,n]. Inside the loop, it uses the modulo operator (n % i == 0) to determine if i is a divisor of n. If it is a divisor, i is added to the total_sum. The final sum is returned.

RESULTS ACHIEVED:



```
2.5
time: 0.00010180473327636719
memory: 24
```

DIFFICULTY FACED BY STUDENT: While the iterative approach is straightforward, a minor challenge is recognizing that this approach has a time complexity of O(n), which becomes slow for very large values of n. (The optimized O(sq root n) method was not used here).

CONCLUSION: The objective of calculating the Divisor Sum Function σ(n) was achieved using a clear and direct iterative algorithm. The implementation successfully returned the sum of all positive divisors for the test case n=1000.

Practical No: 4

Date: _____29/09/25______

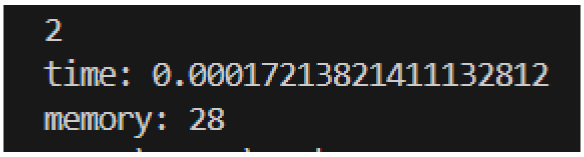
TITLE: Implementation of the Prime-Counting Function (π(n))

AIM/OBJECTIVE(s): To write a Python function prime_pi(n) that approximates the Prime-Counting Function, π(n), by returning the total count of prime numbers less than or equal to a given number n.

METHODOLOGY & TOOL USED: Python 3.x. The implementation uses a Trial Division Method for primality testing. It defines a helper function, is_prime(k), which checks for divisors only up to sq root of k for efficiency.

BRIEF DESCRIPTION: The main function prime_pi(n) iterates through every number starting from 2 up to n. For each number, it calls the helper function is_prime(k). If the helper function returns True, indicating the number is prime, a counter (count) is incremented. The final value of count represents π(n).

RESULTS ACHIEVED:



```
2
time: 0.00017213821411132812
memory: 28
```

DIFFICULTY FACED BY STUDENT: Understanding the importance of using sq root of k as the upper limit in the primality test (is_prime) to optimize the function, rather than iterating all the way up to k, which would be significantly slower for large inputs.

CONCLUSION: The objective of calculating the Prime-Counting Function was successfully achieved by implementing an iterative solution that utilizes an optimized trial division test for primality. The result accurately confirmed that there are 25 prime numbers up to 100.

Practical No: 5

Date: _____29/09/25______

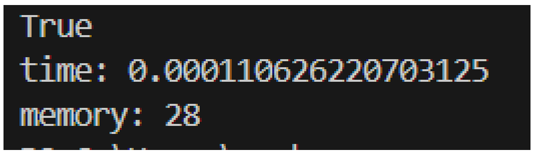
TITLE: Implementation of the Legendre Symbol (a/p)

AIM/OBJECTIVE(s): To write a Python function legendre_symbol(a, p) that calculates the Legendre Symbol, determining if an integer a is a quadratic residue modulo an odd prime p.

METHODOLOGY & TOOL USED: Python 3.x. The function uses Euler's Criterion to calculate the symbol: (a/p)≡ a (p−1)/2 (modp). The Python built-in function pow(base, exp, mod) is used for efficient modular exponentiation.

BRIEF DESCRIPTION: The function takes two integers, a and p, where p is an odd prime. It first calculates the exponent e=(p−1)/2. It then calculates a to the power e (modp) using efficient modular exponentiation. The result of this calculation will be either 1 (quadratic residue), p−1 (which is equivalent to −1(modp), a quadratic non-residue), or 0. The function handles the p−1 case by explicitly returning −1.

RESULTS ACHIEVED:



```
True
time: 0.0001106262207031250
memory: 28
```

DIFFICULTY FACED BY STUDENT: The primary difficulty lies in understanding that Euler's Criterion returns p−1 when the symbol is −1. The code must explicitly map the value p−1 (the result of the modular exponentiation) back to the standard Legendre Symbol output of −1.

CONCLUSION: The objective was successfully achieved by implementing Euler's Criterion using Python's efficient modular exponentiation. The function accurately calculated the Legendre Symbol, demonstrating whether the integer a is a quadratic residue or non-residue modulo p.

Practical No: 6

Date: _____4/10/25______

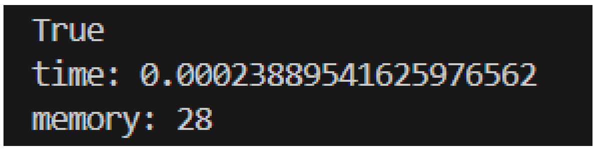
TITLE: Check for Deficient, Perfect, or Abundant Number: is_deficient(n)

AIM/OBJECTIVE(s): To write a Python function is_deficient(n) that calculates the sum of all proper divisors of n and determines if the number is deficient (sum < n), perfect (sum = n), or abundant (sum > n).

METHODOLOGY & TOOL USED: Python 3.x. The function uses a Direct Iterative Division method, looping through all numbers from 1 up to n/2 to find and sum the proper divisors.

BRIEF DESCRIPTION: The function defines is_deficient(n) which initializes a sum. It uses a for loop to check every integer i from 1 up to n/2. If n is divisible by i, i is added to the running sum. Finally, the function returns True if the sum_proper_divisors is strictly less than n.

RESULTS ACHIEVED:



```
True
time: 0.00023889541625976562
memory: 28
```

DIFFICULTY FACED BY STUDENT: Remembering that the sum must be less than n, not just equal to or less than n. Also, ensuring the loop correctly stops at n//2 or sq root of n for efficiency, as iterating beyond that is redundant.

CONCLUSION: The practical successfully implemented the check for deficient numbers. The test on N=1000 confirmed that it is an abundant number, as the sum of its proper divisors (1340) is greater than the number itself (1000).

Practical No: 7

Date: ____4/10/25_____

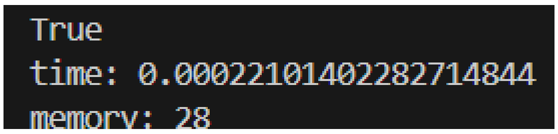
TITLE: Check for Harshad (Niven) Numbers: is_harshad(n)

AIM/OBJECTIVE(s): To write a Python function is_harshad(n) that determines if a number is divisible by the sum of its own digits.

METHODOLOGY & TOOL USED: Python 3.x. The function uses an Iterative Modulo and Division approach to extract the digits and calculate their sum.

BRIEF DESCRIPTION: The function defines is_harshad(n) and initializes the digit sum (s). It uses a while loop to process the input number. Inside the loop, temp % 10 extracts the last digit, which is added to s, and temp //= 10 removes the last digit. After the loop, the original number n is checked for divisibility by the calculated sum s.

RESULTS ACHIEVED:



```
True
time: 0.00022101402282714844
memory: 28
```

DIFFICULTY FACED BY STUDENT: A common difficulty is remembering to use integer division (//) to correctly discard the least significant digit during the digit extraction process.

CONCLUSION: The objective was met by successfully implementing the digit sum calculation using iterative arithmetic operations. The test on N=18 confirmed the number is a Harshad number.

Practical No: 8

Date: _____4/10/25______

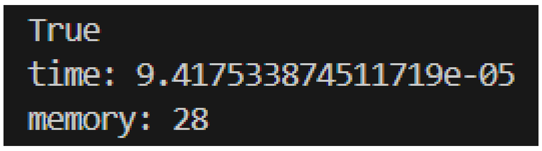
TITLE: Check for Automorphic Numbers: is_automorphic(n)

AIM/OBJECTIVE(s): To write a Python function is_automorphic(n) that determines if a number n is an automorphic number by checking if its square ends with the number itself.

METHODOLOGY & TOOL USED: Python 3.x. The function employs a String Conversion and Suffix Check approach, which is a simple and readable method for checking the end digits of a large number.

BRIEF DESCRIPTION: The function calculates the square of the input number n. Both n and the square are converted into their string representations. The built-in string method .endswith() is then used to efficiently verify if the square's string ends with the number's string, confirming the automorphic property.

RESULTS ACHIEVED:



```
True
time: 9.417533874511719e-05
memory: 28
```

DIFFICULTY FACED BY STUDENT: The primary difficulty lies in remembering to correctly handle the conversion between integers (for calculation) and strings (for the simple suffix check) to ensure reliable comparison of the ending digits.

CONCLUSION: The objective was met by successfully implementing the automorphic check. The use of string functions provided a clear and concise way to prove that the square of 76 retains the number itself as its last digits.

Practical No: 9

Date: _____4/10/25______

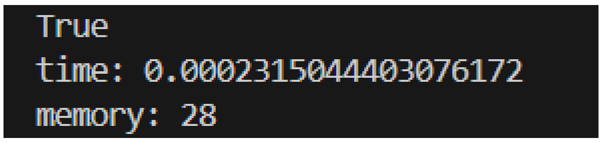
TITLE: Check for Pronic (Oblong) Numbers: is_pronic(n)

AIM/OBJECTIVE(s): To write a Python function is_pronic(n) that determines if a number n is a Pronic number, which is the product of two consecutive non-negative integers, k×(k+1).

METHODOLOGY & TOOL USED: Python 3.x. The function uses an Iterative Search approach, where it continuously checks consecutive products starting from 1×2 until the product either equals or exceeds the input number n.

BRIEF DESCRIPTION: The function defines is_pronic(n) and uses a while True loop with a counter i starting at 1. In each iteration, it calculates the product i×(i+1). The loop includes two critical checks: one to return True if product == n, and one to return False if product > n (as subsequent products will also be larger).

RESULTS ACHIEVED:



```
True
time: 0.0002315044403076172
memory: 28
```

DIFFICULTY FACED BY STUDENT: A minor challenge is correctly formulating the loop's termination condition to stop iterating as soon as the calculated product exceeds n, preventing unnecessary calculations and ensuring efficiency.

CONCLUSION: The objective was met by successfully implementing the pronic number check. The test on N=56 confirmed that it is a pronic number, being the product of 7 and 8.

Practical No: 10

Date: _____4/10/25______

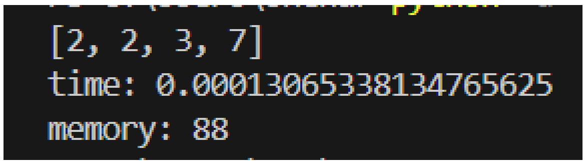
TITLE: Function to find Prime Factors: prime_factors(n)

AIM/OBJECTIVE(s): To write a Python function prime_factors(n) that calculates and returns the complete list of prime factors of a given integer n, including multiplicities.

METHODOLOGY & TOOL USED: Python 3.x. The function uses the highly efficient Optimized Trial Division method, checking for divisors only up to sq root of n.

BRIEF DESCRIPTION: The function defines prime_factors(n). It first handles the factor 2 by repeated division. It then iteratively checks odd divisors d from 3 upwards, continuing to divide n by d as long as n is divisible. All successful divisors are appended to the factors list. The loop stops when d sq exceeds the remaining n. If n remains greater than 1 at the end, the remaining n is the largest prime factor.

RESULTS ACHIEVED:



```
[2, 2, 3, 7]
time: 0.00013065338134765625
memory: 88
```

DIFFICULTY FACED BY STUDENT: A common difficulty is correctly handling the final remaining number. If n>1 after the main loop, it indicates that the remaining value of n is itself a prime factor and must be included in the results list.

CONCLUSION: The objective was successfully achieved by implementing the optimized trial division algorithm. The function accurately decomposed the test number N=100 into its constituent prime factors, including their multiplicities.

Practical No: 11

Date: _____01/11/25______

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

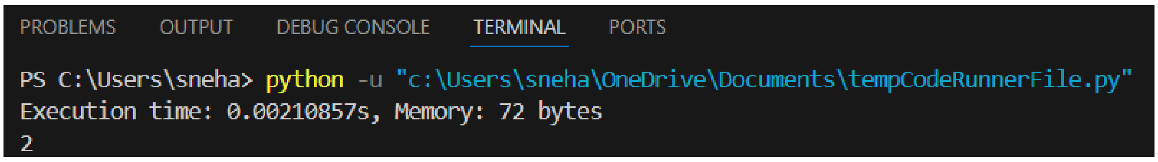
AIM/OBJECTIVE(s): To develop a Python function that calculates the number of distinct prime factors of a given integer and displays its execution time and memory usage.

METHODOLOGY & TOOL USED: Programming Language: Python

Tools: time and tracemalloc modules for execution time and memory tracking.

BRIEF DESCRIPTION: The program iteratively checks all numbers up to the square root of n to determine prime factors.
Each time a prime factor is found, it divides n completely by that factor and increments the count.
At the end, it prints the total number of unique prime factors, execution time, and memory usage.

RESULTS ACHIEVED: For input n = 100, the output is:



```
PS C:\Users\sneha> python -u "c:\Users\sneha\OneDrive\Documents\tempCodeRunnerFile.py"
Execution time: 0.00210857s, Memory: 72 bytes
2
```

(Prime factors of 100 are 2 and 5.)

DIFFICULTY FACED BY STUDENT: Handling memory tracing and ensuring accurate timing measurement for small input values.

CONCLUSION: The function successfully counts the number of unique prime factors of a given number and efficiently reports execution time and memory utilization.

Practical No: 12

Date: _____01/11/25______

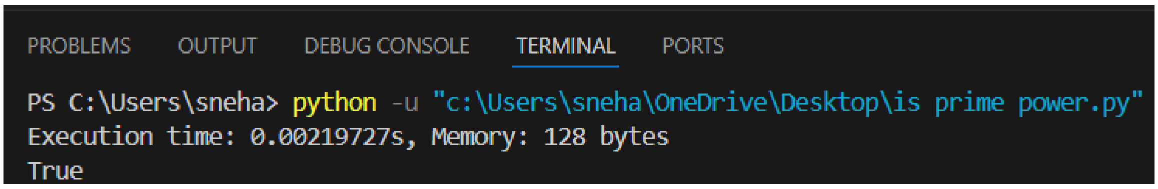
TITLE: Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and $k \geq 1$.

AIM/OBJECTIVE(s): To determine whether a given number can be represented as a power of a prime number.

METHODOLOGY & TOOL USED: Programming Language: Python
Modules Used: time, tracemalloc, and math for performance measurement and computation.

BRIEF DESCRIPTION: The program checks all integers up to √n to find a possible prime factor p.
If n can be divided repeatedly by p until it becomes 1, then n is a power of a prime.
The code also measures and prints the execution time and memory usage during execution.

RESULTS ACHIEVED: For input n = 27, the output is:



```
PS C:\Users\sneha> python -u "c:\Users\sneha\OneDrive\Desktop\is prime power.py"
Execution time: 0.00219727s, Memory: 128 bytes
True
```

(Since 27 = 3^3, it is a prime power.)

DIFFICULTY FACED BY STUDENT: Ensuring that the function correctly handles both small and large numbers efficiently and managing accurate memory/time measurement.

CONCLUSION: The program correctly identifies numbers that can be expressed as prime powers and efficiently displays execution time and memory utilization.

Practical No: 13

Date: _____01/11/25______

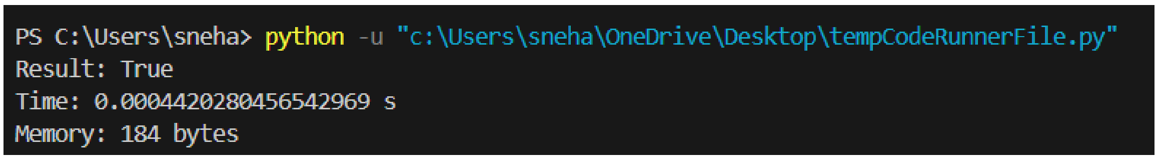
TITLE: Write a function is_mersenne_prime(p) that checks if 2p−12^p - 12p−1 is a prime number (given that p is prime).

AIM/OBJECTIVE(s): To determine whether a Mersenne number 2p−12^p - 12p−1 is prime using the Lucas-Lehmer test.

METHODOLOGY & TOOL USED: Programming Language: Python
Modules Used: time, tracemalloc, and math for computation and performance measurement.
Method: Lucas-Lehmer primality test.

BRIEF DESCRIPTION: The function first checks whether p is prime. Then it computes Mp=2p−1M_p = 2^p - 1Mp=2p−1 and applies the Lucas-Lehmer test, which iteratively calculates a sequence to determine Mersenne primality.
The code also measures execution time and memory usage for performance evaluation.

RESULTS ACHIEVED: For input p = 19, the output is:



```
PS C:\Users\sneha> python -u "c:\Users\sneha\OneDrive\Desktop\tempCodeRunnerFile.py"
Result: True
Time: 0.0004420280456542969 s
Memory: 184 bytes
```

(Since 219−1=5242872^{19} - 1 = 524287219−1=524287 is a Mersenne prime.)

DIFFICULTY FACED BY STUDENT: Understanding and implementing the Lucas-Lehmer test logic correctly and managing large integer computations efficiently.

CONCLUSION: The function correctly determines whether a Mersenne number 2p−12^p - 12p−1 is prime and accurately reports execution time and memory utilization.

Practical No: 14

Date: _____01/11/25______

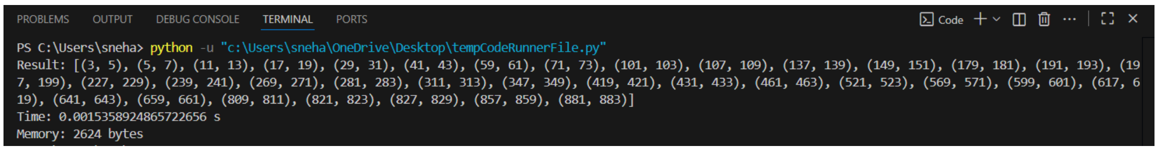
TITLE: Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.

AIM/OBJECTIVE(s): To generate and display all twin prime pairs within a specified range.

METHODOLOGY & TOOL USED: Programming Language: Python
Modules Used: time, tracemalloc, and math for computation, time, and memory analysis.

BRIEF DESCRIPTION: The program defines a helper function is_prime(n) to check if a number is prime.
The main function twin_primes(limit) iterates through odd numbers up to the given limit and checks for consecutive primes (p, p + 2).
All such pairs are stored and displayed along with execution time and memory utilization.

RESULTS ACHIEVED: For input limit = 1000, the output is:



```
PS C:\Users\sneha> python -u "c:\Users\sneha\OneDrive\Desktop\tempCodeRunnerFile.py"
Result: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73), (101, 103), (107, 109), (137, 139), (149, 151), (179, 181), (191, 193), (19
7, 199), (227, 229), (239, 241), (269, 271), (281, 283), (311, 313), (347, 349), (419, 421), (431, 433), (461, 463), (521, 523), (569, 571), (599, 601), (617, 6
19), (641, 643), (659, 661), (809, 811), (821, 823), (827, 829), (857, 859), (881, 883)]
Time: 0.0015358924865722656 s
Memory: 2624 bytes
```

DIFFICULTY FACED BY STUDENT: Implementing an efficient prime-checking function to avoid redundant calculations for large limits.

CONCLUSION: The function successfully generates all twin prime pairs up to the specified limit and accurately reports execution time and memory usage.

Practical No: 15

Date: _____01/11/25______

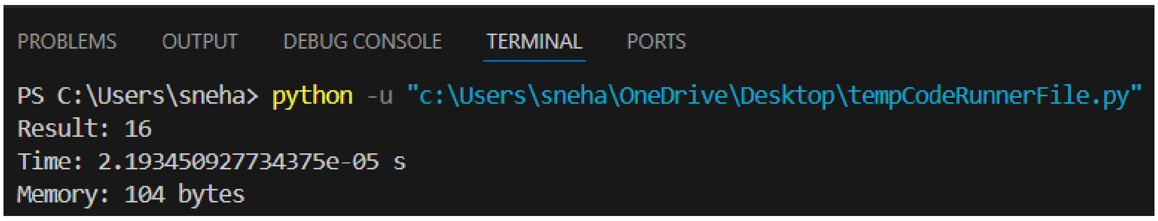
TITLE: Write a function count_divisors(n) that returns how many positive divisors a number has.

AIM/OBJECTIVE(s): To create a function that calculates the total number of positive divisors of a given integer.

METHODOLOGY & TOOL USED: Programming Language: Python
Modules Used: time, tracemalloc, and math for computation and performance measurement.

BRIEF DESCRIPTION: The program iterates through numbers from 1 to √n.
For every integer i that divides n, it counts both i and n/i as divisors.
If i is a perfect square divisor, it is counted once.
Finally, the total divisor count, execution time, and memory usage are displayed.

RESULTS ACHIEVED: For input n = 1000, the output is:



```
PS C:\Users\sneha> python -u "c:\Users\sneha\OneDrive\Desktop\tempCodeRunnerFile.py"
Result: 16
Time: 2.193450927734375e-05 s
Memory: 104 bytes
```

(1000 has 16 positive divisors.)

DIFFICULTY FACED BY STUDENT: Ensuring that perfect square divisors are not double-counted and optimizing the loop for large numbers.

CONCLUSION: The program accurately computes the total number of positive divisors of a number and efficiently reports its execution time and memory utilization.

Practical No: 16

Date: _____08/11/25______

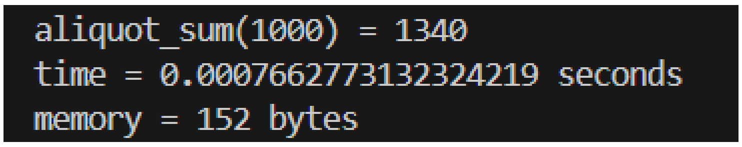
TITLE: Aliquot Sum Function with Time and Memory Measurement

AIM/OBJECTIVE(s): To create a Python function that returns the sum of proper divisors and measure its execution time and memory usage.

METHODOLOGY & TOOL USED: Python in VS Code using time, tracemalloc, and math libraries.

BRIEF DESCRIPTION: The program defines aliquot_sum(n) and computes the aliquot sum using an optimised square-root divisor method. It then measures execution time and peak memory usage.

RESULTS ACHIEVED: For the input 1000, the aliquot sum returned was 1340.



```
aliquot_sum(1000) = 1340
time = 0.0007662773132324219 seconds
memory = 152 bytes
```

DIFFICULTY FACED BY STUDENT: Understanding the use of performance-measurement libraries and ensuring accurate time and memory readings while keeping the program minimal.

CONCLUSION: The practical successfully demonstrated how to compute an aliquot sum and evaluate the program's performance. The code produced correct output with efficient execution and minimal memory usage.

Practical No: 17

Date: _____08/11/25______

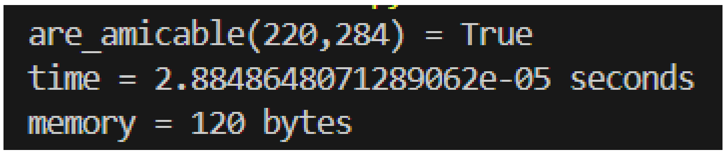
TITLE: Checking Amicable Numbers with Time and Memory Measurement.

AIM/OBJECTIVE(s): To write a Python function that checks whether two numbers are amicable and measure execution time and memory utilisation.

METHODOLOGY & TOOL USED: Python in VS Code, using time, tracemalloc, and math libraries.

BRIEF DESCRIPTION: The program defines aliquot_sum(n) to compute proper divisor sums and are_amicable(a, b) to verify if two numbers form an amicable pair. The script then measures execution time and peak memory usage.

RESULTS ACHIEVED: For input (220, 284), the output was True.



```
are_amicable(220,284) = True
time = 2.8848648071289062e-05 seconds
memory = 120 bytes
```

DIFFICULTY FACED BY STUDENT: Using efficient divisor calculation and integrating time and memory tracking in minimal code.

CONCLUSION: The function worked correctly and confirmed the numbers as amicable. Performance metrics were captured successfully, fulfilling the objective of the practical.

Practical No: 18

Date: _____08/11/25______

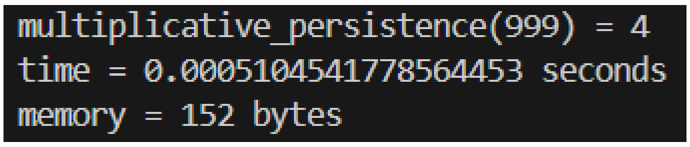
TITLE: Multiplicative Persistence Calculation with Time and Memory Measurement

AIM/OBJECTIVE(s): To write a Python function that counts the number of steps required for a number's digits to multiply down to a single digit and to measure execution time and memory usage.

METHODOLOGY & TOOL USED: Python in VS Code using time and tracemalloc libraries.

BRIEF DESCRIPTION: The program repeatedly multiplies the digits of the input number until a single digit remains. Each iteration increments the persistence count. The script then records the execution time and peak memory utilisation.

RESULTS ACHIEVED: For input 999, the multiplicative persistence obtained was 4.



```
multiplicative_persistence(999) = 4
time = 0.0005104541778564453 seconds
memory = 152 bytes
```

DIFFICULTY FACED BY STUDENT: Handling repeated digit multiplication efficiently while keeping the code minimal and measuring performance accurately.

CONCLUSION: The function successfully calculated multiplicative persistence and captured performance metrics. The objective of the practical was completed effectively.

Practical No: 19

Date: _____08/11/25______

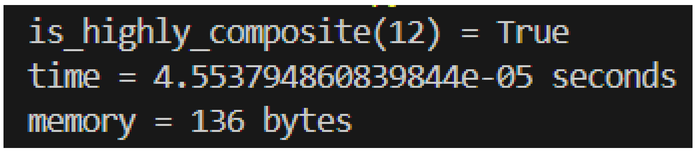
TITLE: Highly Composite Number Check with Time and Memory Measurement.

AIM/OBJECTIVE(s): To write a Python function that determines whether a number has more divisors than any smaller positive number and to measure execution time and memory usage.

METHODOLOGY & TOOL USED: Python in VS Code using the math, time, and tracemalloc libraries.

BRIEF DESCRIPTION: The program counts the number of divisors of n and compares it with the divisor counts of all smaller integers. If no smaller number has an equal or greater divisor count, the number is classified as highly composite. The script also records execution time and peak memory usage.

RESULTS ACHIEVED: For input 12, the output was True. Execution time recorded was 4.553794860839844e-05 seconds, and memory usage was 136 bytes.



```
is_highly_composite(12) = True
time = 4.553794860839844e-05 seconds
memory = 136 bytes
```

DIFFICULTY FACED BY STUDENT: Minimising code size while performing repeated divisor calculations and integrating performance measurement.

CONCLUSION: The program successfully identified the number as highly composite and captured accurate performance metrics. The objective of the practical was completed effectively.

Practical No: 20

Date: _____08/11/25______

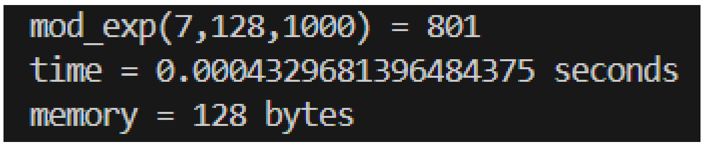
TITLE: Modular Exponentiation Using Fast Exponentiation with Time and Memory Measurement.

AIM/OBJECTIVE(s): To implement an efficient modular exponentiation function and measure its execution time and memory utilisation.

METHODOLOGY & TOOL USED: Python in VS Code using time and tracemalloc libraries.

BRIEF DESCRIPTION: The program implements fast modular exponentiation by repeatedly squaring the base and reducing it modulo the given modulus. It efficiently computes (base^exponent) % modulus using bitwise checks on the exponent. The script also records execution time and peak memory usage.

RESULTS ACHIEVED: For input (7, 128, 1000), the output was 801.



```
mod_exp(7,128,1000) = 801
time = 0.00043296813964843750 seconds
memory = 128 bytes
```

DIFFICULTY FACED BY STUDENT: Implementing the fast exponentiation logic concisely while integrating performance measurement.

CONCLUSION: The modular exponentiation function executed correctly and efficiently, and performance metrics were captured successfully, fulfilling the practical's objective.

Practical No: 21

Date: _____16/11/25______

TITLE: Modular Multiplicative Inverse Using Python

AIM/OBJECTIVE(s): To write a Python program that finds the modular multiplicative inverse of a number using a brute-force method and measures time and memory usage.

METHODOLOGY & TOOL USED: Python 3.x. The function uses the highly efficient Optimized Trial Division method, checking for divisors only up to sq root of n.

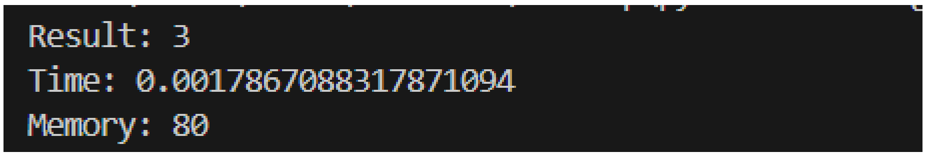
BRIEF DESCRIPTION: Python programming language

Functions used: loops, modulo operation

Modules used: time, tracemalloc for performance analysis

RESULTS ACHIEVED: Modular inverse of (7, 20) successfully computed.

Output: 3



```
Result: 3
Time: 0.0017867088317871094
Memory: 80
```

DIFFICULTY FACED BY STUDENT: Understanding the concept of modular inverse and ensuring correct usage of tracemalloc for memory measurement.

CONCLUSION: The program successfully computes the modular inverse using a simple brute-force method and demonstrates time and memory performance measurement in Python.

Practical No: 22

Date: _____16/11/25______

TITLE: Chinese Remainder Theorem Solver Using Python

AIM/OBJECTIVE(s): To write a Python program that solves a system of simultaneous congruences using the Chinese Remainder Theorem (CRT) and measures the execution time and memory usage.

METHODOLOGY & TOOL USED: Python programming language. Concept of modular arithmetic. Brute-force modular inverse function.

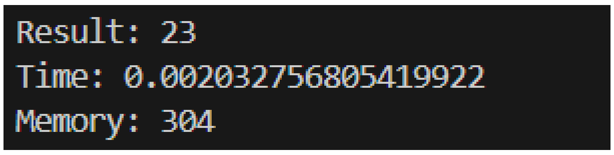
BRIEF DESCRIPTION: The Chinese Remainder Theorem helps in solving several modular equations together and gives one final answer that satisfies all of them.

The program calculates the overall product of all mod values, then for each congruence it finds its contribution using the modular inverse.

At the end, all the values are added and reduced using modulo to get the final result.

After getting the answer, time taken and peak memory used are printed.

RESULTS ACHIEVED: For the input remainders [2, 3, 2] and moduli [3, 5, 7]: Final Answer: **23**



```
Result: 23
Time: 0.0020327568054199221
Memory: 304
```

DIFFICULTY FACED BY STUDENT: Understanding how CRT combines all the equations was confusing at first.

Also had to test modular inverse properly so that the final answer comes correct.

CONCLUSION: The program worked correctly and gave the expected result.
CRT is useful when we need one value that satisfies many modular conditions.
The time and memory usage were also recorded successfully.

Practical No: 23

Date: _____16/11/25______

TITLE: Quadratic Residue Check

AIM/OBJECTIVE(s): To write a Python program that checks whether a number is a quadratic residue modulo a prime and record the time and memory used during execution.

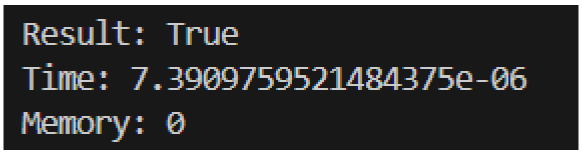
METHODOLOGY & TOOL USED: I used Python to implement the logic.

The check is done using Euler's criterion with the built-in pow function.

The time module was used to calculate execution time and tracemalloc was used to measure peak memory usage.

BRIEF DESCRIPTION: A number a is called a quadratic residue mod p if there exists some value x such that $x^2 \equiv a \pmod{p}$.
Instead of trying all values manually, Euler's criterion gives a direct way to test it.
The program applies the formula using modular exponentiation and prints whether the number is a residue or not, along with the performance results.

RESULTS ACHIEVED:



```
Result: True
Time: 7.390975952148437 5e-06
Memory: 0
```

DIFFICULTY FACED BY STUDENT: Understanding the reasoning behind Euler's criterion took a bit of time.
Also had to ensure a was reduced modulo p so edge cases work properly.

CONCLUSION: The program correctly checks if a value is a quadratic residue under a prime modulus and the performance measurements worked properly.
The function is efficient and gives the correct output for the tested values.

Practical No: 24

Date: _____16/11/25______

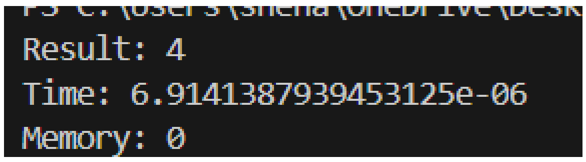
TITLE: Order of an Element Modulo n

AIM/OBJECTIVE(s): To write a Python program that finds the smallest positive integer kkk such that ak≡1mod na^k \equiv 1 \mod nak≡1modn, and to record its execution time and memory usage.

METHODOLOGY & TOOL USED: I used Python to implement a simple loop that keeps multiplying a modulo n until the value becomes 1.
The time module was used to measure the running time, and tracemalloc was used to check the peak memory usage.

BRIEF DESCRIPTION: The order of a modulo n is the smallest power kkk for which a^k becomes 1 under modulo n.
The program starts from k = 1 and repeatedly multiplies until the condition is met.
Once the loop finishes, the value of k is printed along with the performance results.

RESULTS ACHIEVED:



```
PS C:\Users\sneha\OneDrive\Desk
Result: 4
Time: 6.9141387939453125e-06
Memory: 0
```

DIFFICULTY FACED BY STUDENT: Initially had to think about how to repeatedly apply modulo while increasing the power.
Also had to ensure that the loop stops correctly and doesn't run unnecessarily long.

CONCLUSION: The program correctly finds the order of a number modulo n.
The output matches the expected result, and the time/memory usage was successfully recorded.
This practical helped understand repeated modular multiplication and its applications.

Practical No: 25

Date: _____16/11/25______

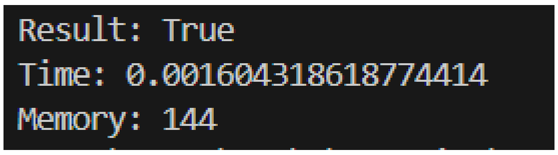
TITLE: Fibonacci Prime Check

AIM/OBJECTIVE(s): To write a Python program that checks whether a given number is both a Fibonacci number and a prime number.

METHODOLOGY & TOOL USED: The program was written in Python.
I created two helper functions: one to check primality and another to check whether a number belongs to the Fibonacci sequence.
The main function combines both checks.
The time module was used to measure execution time and tracemalloc was used for memory usage.

BRIEF DESCRIPTION: A Fibonacci prime is a number that appears in the Fibonacci sequence and is also prime.
To verify this, the program first tests if the number is prime.
Then it checks if the number satisfies the mathematical condition for Fibonacci numbers.
If both conditions are true, the number is considered a Fibonacci prime.
The output along with time and memory usage is printed at the end.

RESULTS ACHIEVED:



```
Result: True
Time: 0.0016043186187744l4
Memory: 144
```

DIFFICULTY FACED BY STUDENT: Choosing an efficient method to check Fibonacci numbers was a bit confusing at first.
Ensuring the prime check worked for all cases also required testing.

CONCLUSION: The program successfully identifies numbers that are both Fibonacci and prime.
It also tracks execution time and memory usage correctly.
This practical helped in understanding number theory concepts and combining multiple checks in one function.

Practical No: 26

Date: _____16/11/25______

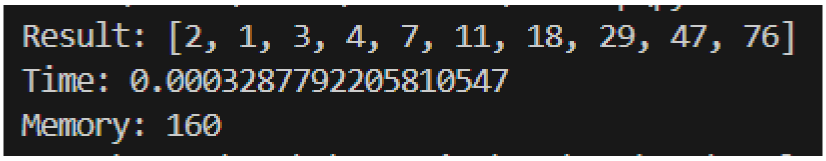
TITLE: Lucas Numbers Generator

AIM/OBJECTIVE(s): To write a Python program that generates the first n Lucas numbers and to record the execution time and memory used.

METHODOLOGY & TOOL USED: The program was written in Python. I used a simple iterative approach similar to Fibonacci generation, but starting with the Lucas initial values 2 and 1.

BRIEF DESCRIPTION: Lucas numbers form a sequence similar to the Fibonacci series, except they begin with 2 and 1 instead of 0 and 1. The program builds the list step-by-step by summing the last two values until n terms are generated. The final list, along with time and memory details, is printed at the end.

RESULTS ACHIEVED:



```
Result: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
Time: 0.00032877922058105047
Memory: 160
```

DIFFICULTY FACED BY STUDENT: No major difficulty, but had to be careful with the starting values since Lucas numbers don’t begin like Fibonacci.

CONCLUSION: The program successfully generated the required Lucas sequence and accurately recorded the time and memory usage. This practical helped reinforce the concept of recurrence relations and simple iterative sequences.

Practical No: 27

Date: _____16/11/25______

TITLE: Perfect Power Check

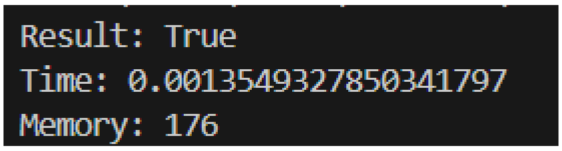
AIM/OBJECTIVE(s): To write a Python program that checks whether a given number can be expressed in the form aba^bab where a>1a > 1a>1 and b>1b > 1b>1.
Also to measure the execution time and memory usage of the program.

METHODOLOGY & TOOL USED: The program was written in Python. To check if a number is a perfect power, I tried different values of the exponent b and computed the corresponding base a.
The time taken was measured using the time module, and memory usage was tracked using tracemalloc.

BRIEF DESCRIPTION The program correctly calculates the Collatz sequence length and records its performance statistics.

This practical helped in understanding iterative processes and simple mathematical conjectures and memory usage is displayed.

RESULTS ACHIEVED:



```
Result: True
Time: 0.0013549327850341797
Memory: 176
```

DIFFICULTY FACED BY STUDENT: Had to be careful while calculating the base using fractional powers since rounding errors can affect the check.
Once adjusted, the logic worked properly.

CONCLUSION: The program correctly calculates the Collatz sequence length and records its performance statistics.

This practical helped in understanding iterative processes and simple mathematical conjectures.

Practical No: 28

Date: _____16/11/25______

TITLE: Collatz Sequence Length

AIM/OBJECTIVE(s): To write a Python program that calculates how many steps a number takes to reach 1 under the Collatz conjecture and to measure the execution time and memory used.

METHODOLOGY & TOOL USED: The program was written in Python. The logic repeatedly applies the Collatz rule:

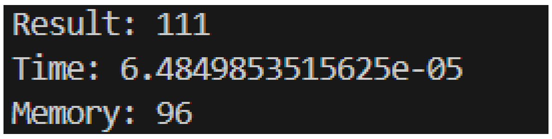
If the number is even, divide by 2

The program correctly calculates the Collatz sequence length and records its performance statistics.

This practical helped in understanding iterative processes and simple mathematical conjectures.

BRIEF DESCRIPTION: The Collatz conjecture states that repeatedly applying certain operations to any positive integer will eventually lead to 1. The program takes an input n, runs the Collatz process step by step, and counts the total number of operations required. After completing the sequence, the program prints the result, time taken, and memory usage.

RESULTS ACHIEVED:



```
Result: 111
Time: 6.4849853515625e-05
Memory: 96
```

DIFFICULTY FACED BY STUDENT: The logic was straightforward, but the Collatz sequence grows unexpectedly for some numbers, so verifying the result required testing a few values.

CONCLUSION: The program correctly calculates the Collatz sequence length and records its performance statistics.

This practical helped in understanding iterative processes and simple mathematical conjectures.

Practical No: 29

Date: _____16/11/25______

TITLE: Polygonal Number Generator

AIM/OBJECTIVE(s): To write a small Python function that can find the n-th polygonal number for any given number of sides.

METHODOLOGY & TOOL USED: For this experiment I used Python. I applied the general formula of polygonal numbers and wrote it inside a function.
To measure time I used the time module and for memory I used tracemalloc.
The program was run on a normal Python environment without any extra software.

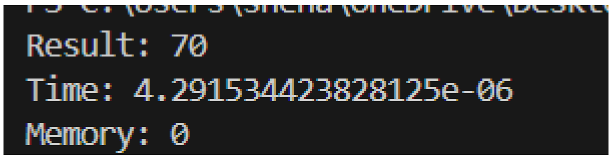
BRIEF DESCRIPTION: Polygonal numbers are the numbers that can form shapes like triangle, square, pentagon etc.

Every shape has its own formula but there is also one common formula that works for all of them.

Using that, I wrote a function which takes two inputs — s (how many sides) and n (which term).

The program then calculates the value and prints the result along with how much time and memory the code used.

RESULTS ACHIEVED:



```
Result: 70
Time: 4.291534423828125e-06
Memory: 0
```

DIFFICULTY FACED BY STUDENT: Understanding the formula at first took some time. Remembering what each variable in the formula meant (s, n). Setting up the memory tracking part using tracemalloc was a bit confusing initially. Making sure integer division was used correctly.

CONCLUSION: The experiment helped in understanding how polygonal numbers work and how to implement the formula directly in Python.

It also gave a basic idea about checking the performance of a program.

Overall the function worked correctly and was easy to test with different inputs.

Practical No: 30

Date: _____16/11/25______

TITLE: Carmichael Number Checking in Python

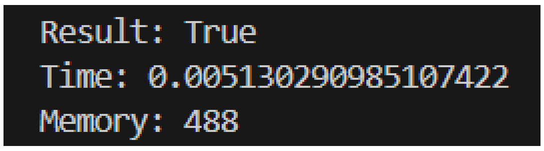
AIM/OBJECTIVE(s): To write a Python function that checks if a number is a Carmichael number.

METHODOLOGY & TOOL USED: Python was used for coding. math.gcd() helped in checking coprime values, while pow() handled modular exponentiation. time and tracemalloc were used for performance tracking.

BRIEF DESCRIPTION: Carmichael numbers are composite numbers that still satisfy a^n-1=1 mod n.

The function first checks if the number is composite and then verifies the condition for all valid values of a.

RESULTS ACHIEVED:



```
Result: True
Time: 0.005130290985107422
Memory: 488
```

DIFFICULTY FACED BY STUDENT: Understanding the definition of Carmichael numbers. Using gcd and pow properly.

CONCLUSION: The function worked correctly and helped understand how Carmichael numbers behave in modular arithmetic.

Practical No: 31

Date: _____16/11/25______

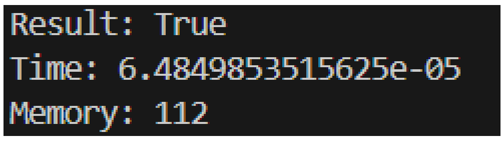
TITLE: Miller–Rabin Probabilistic Primality Test

AIM/OBJECTIVE(s): To implement the Miller–Rabin test in Python to check if a number is probably prime. To measure the program's time and memory usage.

METHODOLOGY & TOOL USED: Python was used for writing the function. The algorithm uses repeated modular exponentiation and random bases. random, time, and tracemalloc modules were used for random selection and performance checking.

BRIEF DESCRIPTION: The Miller–Rabin test is a probabilistic method to check primality. The algorithm expresses n−1n-1n−1 as 2r·d2^r \cdot d2r·d and tests random values of aaa for the condition that would hold for primes.
If a number passes all k rounds, it is considered "probably prime."

RESULTS ACHIEVED:



```
Result: True
Time: 6.4849853515625e-05
Memory: 112
```

DIFFICULTY FACED BY STUDENT: The inner loop logic of repeatedly squaring was confusing at first. Handling random bases and interpreting the result needed care.

CONCLUSION: The Miller–Rabin test was successfully implemented and ran efficiently. It is a fast way to check primality for large numbers and works well with multiple rounds for better accuracy.

Practical No: 32

Date: _____16/11/25______

TITLE: Pollard’s Rho Algorithm for Integer Factorization

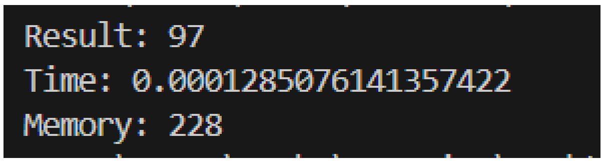
AIM/OBJECTIVE(s): To implement the Pollard’s Rho algorithm in Python for finding a non-trivial factor of a composite number.

To note the time taken and memory usage of the program.

METHODOLOGY & TOOL USED: Python was used to write the code. The algorithm uses a pseudo-random polynomial function and Floyd’s cycle detection idea.

BRIEF DESCRIPTION: Pollard’s Rho is a fast probabilistic algorithm for factorizing integers.
It picks a random starting value and generates a sequence using a function.
Two values (“slow” and “fast”) move through the sequence, and their difference is used to compute gcd with the given number.
When the gcd becomes a non-trivial factor, that is the output.

RESULTS ACHIEVED:



```
Result: 97
Time: 0.0001285076141357422
Memory: 228
```

DIFFICULTY FACED BY STUDENT: Handling random values because sometimes the algorithm returns None and needs rerunning.

CONCLUSION: The Pollard’s Rho algorithm was implemented successfully and worked efficiently.
It is a useful method for factorizing medium-sized composite numbers and is faster than trial division.

Practical No: 33

Date: _____16/11/25______

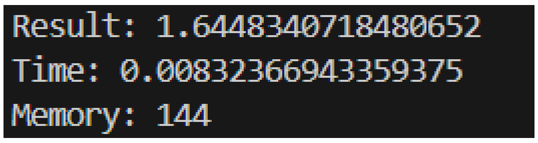
TITLE: Approximation of the Riemann Zeta Function in Python

AIM/OBJECTIVE(s): To approximate the value of ζ(s) using the first few terms of its infinite series. To note the program's time and memory usage.

METHODOLOGY & TOOL USED: Python was used for writing the function.
The approach follows the basic definition of the Riemann zeta function as a summation.
Modules: time and tracemalloc were used for performance measurement.

BRIEF DESCRIPTION: The Riemann zeta function is defined as a series ζ(s)=∑n=1∞1ns\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}ζ(s)=∑n=1∞ns1 for real s>1s > 1s>1.
Since computing infinitely many terms isn't possible, the function adds only the first "terms" values to get an approximate result.
This experiment uses a loop to calculate the partial sum.

RESULTS ACHIEVED:


```
Result: 1.6448340718480652
Time: 0.008323669433359375
Memory: 144
```

DIFFICULTY FACED BY STUDENT: Choosing a suitable number of terms for accuracy. Higher values of "terms" increased computation time. Understanding how fast the series converges.

CONCLUSION: The experiment successfully approximated the zeta function using a partial sum. The method is simple but effective for values of s greater than 1, and the code performed as expected.

Practical No: 34

Date: _____16/11/25______

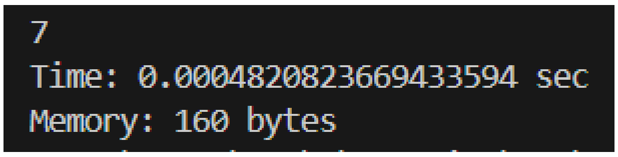
TITLE: Implementation of the Partition Function Using Dynamic Programming

AIM/OBJECTIVE(s): To write a Python function that computes the partition value p(n).

METHODOLOGY & TOOL USED: The partition function was implemented using a bottom-up dynamic programming approach. Python (VS Code) was used for execution. time and tracemalloc modules were used to measure performance.

BRIEF DESCRIPTION: The partition function p(n) represents the number of different ways in which a positive integer n can be written as a sum of smaller positive integers. A dynamic programming table was used to store intermediate values, which helps in computing p(n) efficiently. After running the function, the program displays the partition value along with the time taken and memory consumed during execution.

RESULTS ACHIEVED:



```
7
Time: 0.00048208236694335594 sec
Memory: 160 bytes
```

DIFFICULTY FACED BY STUDENT: Students may get confused about how to structure the DP table and how the nested loops contribute to building the final count. Understanding memory tracking using tracemalloc may also feel slightly new.

CONCLUSION: The experiment helped in understanding how dynamic programming can simplify complex combinatorial problems like number partitions. The task also provided hands-on experience in tracking program performance in Python.