Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25BAI11543
Name of Student	:Manushri Gupta
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:

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Factorial of a non-negative integer n (n!)		
2	Palindrome(n) that checks if a number reads the same forwards and backwards		
3	Mean of digits(n) that returns the average of all digits in a number		
4	Digital root		
5	Sum of proper divisors of n		
6	Deficient Number Checker		
7	Harshad (Niven) Number Checker		
8	Automorphic Number Checker		
9	Pronic (Oblong) Number Checker		
10	Prime Factorization Program		
11	Count Distinct Prime Factors of a Number		
12	Check if a Number is a Prime Power		
13	Check if a Number is a Mersenne Prime		
14	Generate All Twin Prime Pairs up to a Given Limit		

15	Generate All Twin Prime Pairs up to a Given Limit		
16	Aliquot Sum Calculator		
17	Amicable Numbers Checker		
18	Multiplicative Persistence Finder		
19	Highly Composite Number Detector		
20	Modular Exponentiation (Fast Power Mod) Program		
21	Modular Multiplicative Inverse Finder		
22	Chinese Remainder Theorem Solver		
23	Quadratic Residue Checker		
24	Multiplicative Order Calculator (Order Mod n)		
25	Fibonacci Prime Checker		
26	Lucas Numbers Generator		
27	Perfect Power Checker		
28	Collatz Sequence Length Calculator		
29	Polygonal Number Calculator		
30	Carmichael Number Checker		
31	Miller–Rabin Primality Test		

32	**Pollard's Rho Algorithm**		
33	**Riemann Zeta Function Approximation**		
34	**Partition Function Calculator**		

Practical No: 1

TITLE: **Factorial of a non-negative integer n (n!)**

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

METHODOLOGY:

```
import time
import tracemalloc

def factorial(n):
    """Return the factorial of a non-negative integer n."""
    if n < 0:
        return "Factorial is not defined for negative numbers."
    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

# Example usage
n = 10  # change this to test other numbers

# Start measuring time and memory
start_time = time.perf_counter()
tracemalloc.start()

fact = factorial(n)
```

```
# Stop measuring memory and time
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.perf_counter()

print(f"{n}! = {fact}")
print(f"Execution Time: {end_time - start_time:.8f} seconds")
print(f"Memory Usage: Current = {current} bytes, Peak = {peak} bytes")
```

RESULTS ACHIEVED:

10! = 3628800
Execution Time: 0.00003548 seconds
Memory Usage: Current = 32 bytes, Peak = 104 bytes

CONCLUSION:

Skills Acquired:

- Functions & Loops
- Conditional Statements
- Error Handling
- Execution Time Measurement
- Memory Usage Tracking
- Input Validation
- Best Coding Practices
- Python Standard Libraries

Practical No: 2

TITLE: **Palindrome(n) that checks if a number reads the same forwards and backwards**

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

METHODOLOGY:

```
import time
import tracemalloc

def is_palindrome(n):
    """Check if a number reads the same forwards and backwards."""
    n_str = str(n)  # Convert number to string
    return n_str == n_str[::-1]  # Compare with its reverse

# Example usage
n = 12321  # change this number to test others

# Start measuring time and memory
start_time = time.perf_counter()
tracemalloc.start()

try:
    result = is_palindrome(n)
finally:
    # Stop measuring memory
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.perf_counter()

print(f"Is {n} a palindrome? {result}")
print(f"Execution Time: {end_time - start_time:.8f} seconds")
print(f"Memory Usage: Current = {current} bytes, Peak = {peak} bytes")
```

RESULTS ACHIEVED:

Is 12321 a palindrome? True
Execution Time: 0.00002888 seconds
Memory Usage: Current = 0 bytes, Peak = 92 bytes

CONCLUSION:

Skills Acquired:

- Functions & Return Values
- Type Conversion (int ↔ str)
- String Manipulation & Slicing
- Conditional Logic
- Execution Time Measurement
- Memory Usage Tracking
- Input Validation Awareness
- Python Standard Libraries (time, tracemalloc)

Practical No:3

TITLE: Mean of digits(n) that returns the average of all digits in a number

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

METHODOLOGY:

```
import time
import tracemalloc

def mean_of_digits(n):
    """Return the average of all digits in a number."""
    if n < 0:
        n = -n
    digits = [int(d) for d in str(n)]
    return sum(digits) / len(digits)

# Example usage
n = 12345

# Start measuring
start_time = time.perf_counter()
tracemalloc.start()

try:
    mean = mean_of_digits(n)
finally:
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.perf_counter()

print(f"Mean of digits of {n} = {mean}")
print(f"Execution Time: {end_time - start_time:.8f} seconds")
print(f"Memory Usage: Current = {current} bytes, Peak = {peak} bytes")
```

RESULTS ACHIEVED:

Mean of digits of 12345 = 3.0
Execution Time: 0.00020500 seconds
Memory Usage: Current = 0 bytes, Peak = 158 bytes

CONCLUSION:

Skills Acquired:

- Functions & Return Values
- Conditional Statements
- List Comprehension
- Type Conversion (int ↔ str)
- Arithmetic Operations (sum, division)
- Execution Time Measurement
- Memory Usage Tracking
- Code Optimization Awareness
- Python Standard Libraries (time, tracemalloc)

Practical No: 4

TITLE: Digital root

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

METHODOLOGY:

```
import time

import tracemalloc


def digital_root(n):

    """Return the digital root of a number (sum of digits until one digit remains)."""

    if n < 0:

        n = -n  # handle negative numbers


    while n >= 10:  # repeat until only one digit remains

        n = sum(int(d) for d in str(n))

    return n


# Example usage

n = 9875  # try changing this number


# Start measuring time and memory

start_time = time.perf_counter()

tracemalloc.start()
```

```
try:
    result = digital_root(n)
finally:
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.perf_counter()


print(f"Digital root of {n} = {result}")
print(f"Execution Time: {end_time - start_time:.8f} seconds")
print(f"Memory Usage: Current = {current} bytes, Peak = {peak} bytes")
```

RESULTS ACHIEVED:

Digital root of 9875 = 2
Execution Time: 0.00006928 seconds
Memory Usage: Current = 0 bytes, Peak = 489 bytes

CONCLUSION:

Skills Acquired:

- Loops & Iteration (while loop)
- List Comprehension
- Type Conversion (int ↔ str)
- Conditional Logic
- Arithmetic Operations
- Execution Time Measurement
- Memory Usage Tracking
- Problem Decomposition (breaking down multi-step logic)
- Python Standard Libraries (time, tracemalloc)

Practical No: 5

TITLE: Sum of proper divisors of n

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

METHODOLOGY:

```
import time

import tracemalloc


def is_abundant(n):

    """Return True if the sum of proper divisors of n is greater than n."""

    if n < 1:

        return False  # no such thing as abundant 0 or negative numbers


    divisors_sum = 1  # start with 1 since it's always a proper divisor

    for i in range(2, int(n**0.5) + 1):

        if n % i == 0:

            divisors_sum += i

            if i != n // i:  # avoid adding the square root twice

                divisors_sum += n // i


    return divisors_sum > n
```

```
# Example usage

n = 12  # 12 is abundant since 1 + 2 + 3 + 4 + 6 = 16 > 12


# Start measuring time and memory

start_time = time.perf_counter()

tracemalloc.start()


try:

    result = is_abundant(n)

finally:

    current, peak = tracemalloc.get_traced_memory()

    tracemalloc.stop()

    end_time = time.perf_counter()


print(f"Is {n} an abundant number? {result}")

print(f"Execution Time: {end_time - start_time:.8f} seconds")

print(f"Memory Usage: Current = {current} bytes, Peak = {peak} bytes")
```

RESULTS ACHIEVED:

Is 12 an abundant number? True
Execution Time: 0.00006301 seconds
Memory Usage: Current = 0 bytes, Peak = 88 bytes

CONCLUSION:

Skills Acquired:

- Functions & Return Values
- Loops & Range Iteration

- Conditional Statements (if / else)
- Divisibility & Modulo Operator (%)
- Mathematical Logic (Square Root Optimization)
- Efficient Algorithm Design
- Execution Time Measurement
- Memory Usage Tracking
- Python Standard Libraries (time, tracemalloc)

Practical No: 6

TITLE: **Deficient Number Checker**

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

METHODOLOGY:

```
import math
import time
import tracemalloc

def is_deficient(n):
    """Return True if n is a deficient number (sum of proper divisors < n)."""
    if n <= 1:
        return True  # 1 is considered deficient

    total = 1  # start with 1 (always a proper divisor)
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            total += i
            if i != n // i:  # avoid adding the same divisor twice
                total += n // i

    return total < n


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

tracemalloc.start()
start_time = time.time()

result = is_deficient(n)

end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
```

```
tracemalloc.stop()

print(f"\nIs {n} a deficient number? {result}")
print(f"Execution time: {end_time - start_time:.8f} seconds")
print(f"Current memory usage: {current} bytes")
print(f"Peak memory usage: {peak} bytes")
```

RESULTS ACHIEVED:

Enter a number: 3
Is 3 a deficient number? True
Execution time: 0.00003147 seconds
Current memory usage: 768 bytes
Peak memory usage: 768 bytes

CONCLUSION:

Skills acquired:

- Understanding of deficient numbers
- Function definition and return values in Python
- Using loops and conditional statements
- Applying math library functions (`math.sqrt`)
- Optimizing algorithms for efficiency
- Measuring execution time with `time` module
- Tracking memory usage with `tracemalloc`
- Handling user input and formatted output
- Practicing code structure and documentation

Practical No: 7

TITLE: **Harshad (Niven) Number Checker**

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

METHODOLOGY:

```
import time

import tracemalloc

def is_harshad(n):
    """Return True if n is a Harshad number (divisible by sum of its digits)."""
    if n == 0:
        return False

    digit_sum = sum(int(d) for d in str(n))
    return n % digit_sum == 0

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

tracemalloc.start()
start = time.time()

result = is_harshad(n)

end = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"\nIs {n} a Harshad number? {result}")
print(f"Execution time: {end - start:.8f} seconds")
print(f"Current memory usage: {current} bytes")
print(f"Peak memory usage: {peak} bytes")
```

RESULTS ACHIEVED:
Enter a number: 1
Is 1 a Harshad number? True
Execution time: 0.00003552 seconds
Current memory usage: 0 bytes
Peak memory usage: 486 bytes

CONCLUSION:

Skills acquired:

- Digit extraction and manipulation using strings
- Using loops and comprehensions (`sum(int(d) for d in str(n))`)
- Applying modulus operator for divisibility checks
- Implementing conditional logic
- Measuring execution time with `time` module
- Tracking memory usage with `tracemalloc`
- Handling user input/output formatting
- Writing clean, well-documented functions
- Performing basic performance analysis

Practical No:8

TITLE: Automorphic Number Checker

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

METHODOLOGY:

```
import time

import tracemalloc

def is_automorphic(n):
    """Return True if n is an automorphic number (its square ends with n)."""
    square = n ** 2
    return str(square).endswith(str(n))


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

tracemalloc.start()
start_time = time.time()

result = is_automorphic(n)

end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"\nIs {n} an Automorphic number? {result}")
print(f"Execution time: {end_time - start_time:.8f} seconds")
print(f"Current memory usage: {current} bytes")
print(f"Peak memory usage: {peak} bytes")
```

RESULTS ACHIEVED:

Enter a number: 1
Is 1 an Automorphic number? True
Execution time: 0.00003934 seconds

Current memory usage: 0 bytes
Peak memory usage: 116 bytes

CONCLUSION:
Skills acquired:

- Understanding of automorphic numbers
- Working with string methods (`endswith()`)
- Performing exponentiation and number manipulation
- Using type conversion between `int` and `str`
- Implementing logical comparison
- Measuring execution time using `time` module
- Tracking memory usage with `tracemalloc`
- Managing user input/output
- Writing efficient and readable code
- Performing performance analysis and testing

PRACTICAL NO: 9

TITLE: Pronic (Oblong) Number Checker

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

METHODOLOGY:

```
import math

import time

import tracemalloc


def is_pronic(n):

    """Return True if n is a pronic number (product of two consecutive integers)."""

    if n < 0:

        return False


    root = int(math.sqrt(n))

    return root * (root + 1) == n or (root - 1) * root == n



# --- Performance tracking ---

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


tracemalloc.start()
```

```
start = time.time()


result = is_pronic(n)


end = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()


print(f"\nIs {n} a Pronic number? {result}")
print(f"Execution time: {end - start:.8f} seconds")
print(f"Current memory usage: {current} bytes")
print(f"Peak memory usage: {peak} bytes")
```

RESULTS ACHIEVED:

Enter a number: 1
Is 1 a Pronic number? False
Execution time: 0.00001454 seconds
Current memory usage: 768 bytes
Peak memory usage: 768 bytes

CONCLUSION:

Skills acquired:

- Understanding of pronic (oblong) numbers
- Applying square roots to detect consecutive integer products
- Using math library functions (`math.sqrt`)
- Performing integer arithmetic and logical comparisons
- Implementing conditional checks
- Measuring execution time with `time` module
- Tracking memory usage with `tracemalloc`
- Managing user input and formatted output
- Writing efficient, optimized numeric algorithms

- Conducting basic performance analysis

PRACTICAL NO: 10

TITLE: Prime Factorization Program

AIM/OBJECTIVES(s): To write a function prime_factors(n) that returns the list of prime factors of a number.

METHODOLOGY:

```
import math

import time

import tracemalloc


def prime_factors(n):

    """Return a list of prime factors of n."""

    factors = []

    if n <= 1:

        return factors


    while n % 2 == 0:

        factors.append(2)

        n //= 2


    for i in range(3, int(math.sqrt(n)) + 1, 2):

        while n % i == 0:
```

```
            factors.append(i)

            n //= i


    if n > 2:

        factors.append(n)


    return factors




# --- Performance tracking ---

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


tracemalloc.start()

start_time = time.time()


result = prime_factors(n)


end_time = time.time()

current, peak = tracemalloc.get_traced_memory()

tracemalloc.stop()


print(f"\nPrime factors of {n}: {result}")

print(f"Execution time: {end_time - start_time:.8f} seconds")

print(f"Current memory usage: {current} bytes")
```

```
print(f"Peak memory usage: {peak} bytes")
```

RESULTS ACHIEVED:

Enter a number: 1
Prime factors of 1: []
Execution time: 0.00000858 seconds
Current memory usage: 768 bytes
Peak memory usage: 768 bytes

CONCLUSION:

Skills acquired:

- Understanding of prime factorization
- Implementing loops and conditional logic
- Using math functions (`math.sqrt`)
- Practicing integer division and modulus
- Optimizing algorithms for efficiency
- Measuring execution time with `time` module
- Tracking memory usage with `tracemalloc`
- Handling user input and clean output formatting
- Writing structured, readable code
- Performing performance and memory analysis

Practical No: 11

TITLE: **Count Distinct Prime Factors of a Number**

AIM/OBJECTIVE(s): To write a function count distinct prime factors(n) that returns how many unique prime factors a number has.

METHODOLOGY:

```
import time
import tracemalloc

def count_distinct_prime_factors(n):
    count = 0
    i = 2
    while i * i <= n:
        if n % i == 0:
            count += 1
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        count += 1
    return count

num = int(input("Enter a number: "))

start_time = time.time()
tracemalloc.start()

result = count_distinct_prime_factors(num)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()

print("\nNumber of distinct prime factors:", result)
print("Time taken: {:.6f} seconds".format(end_time - start_time))
```

```
print("Current memory usage: {:.2f} KB".format(current / 1024))
print("Peak memory usage: {:.2f} KB".format(peak / 1024))
```

RESULTS ACHIEVED:

Enter a number: 24
Number of distinct prime factors: 2
Time taken: 0.000043 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB

CONCLUSION:

Skills acquired:

- Prime factorization logic
- Algorithmic thinking
- Mathematical optimization
- Function definition
- Looping and conditional statements
- Input/output handling
- Time and memory profiling
- Problem decomposition
- Efficiency awareness
- Debugging and reasoning

Practical No: 12

TITLE: **Check if a Number is a Prime Power**

AIM/OBJECTIVE(s): To write a function is prime power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.

METHODOLOGY:

```
import time
import tracemalloc

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

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

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

num = int(input("Enter a number: "))

start_time = time.time()
tracemalloc.start()
```

```
result = is_prime_power(num)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()

print("\nIs prime power:", result)
print("Time taken: {:.6f} seconds".format(end_time - start_time))
print("Current memory usage: {:.2f} KB".format(current / 1024))
print("Peak memory usage: {:.2f} KB".format(peak / 1024))
```

RESULTS ACHIEVED:

Enter a number: 2
Is prime power: False
Time taken: 0.000066 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.09 KB

CONCLUSION:

Skills acquired:

- Prime number checking logic
- Power and exponentiation concepts
- Looping and conditional statements
- Function definition and modular programming
- Nested loop handling
- Mathematical reasoning and number theory
- Input/output handling
- Time and memory profiling using time and tracemalloc
- Problem decomposition and logical thinking
- Efficiency and optimization awareness

Practical No.: 13

TITLE: Check if a Number is a Mersenne Prime

AIM/OBJECTIVE(s): To write a function is mersenne prime(p) that checks if 2p-1 is a prime number (given that p is prime).

METHODOLOGY:

```
import time

import tracemalloc


def is_prime(n):

    if n < 2:

        return False

    for i in range(2, int(n**0.5) + 1):

        if n % i == 0:

            return False

    return True


def is_mersenne_prime(p):

    if not is_prime(p):

        return False  # p must be prime

    m = 2**p - 1

    return is_prime(m)


p = int(input("Enter a value for p: "))
```

```
start_time = time.time()
tracemalloc.start()

result = is_mersenne_prime(p)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()

print(f"\nIs 2^{p} - 1 a Mersenne prime? {result}")
print("Time taken: {:.6f} seconds".format(end_time - start_time))
print("Current memory usage: {:.2f} KB".format(current / 1024))
print("Peak memory usage: {:.2f} KB".format(peak / 1024))
```

RESULTS ACHIEVED:

```
Enter a value for p: 2
Is 2^2 - 1 a Mersenne prime? True
Time taken: 0.000062 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.09 KB
```

CONCLUSION:

Skills acquired:

- Prime number checking logic
- Understanding of Mersenne primes (2^p − 1 form)
- Function definition and modular programming
- Conditional statements and return handling
- Mathematical reasoning and number theory
- Input/output handling
- Time and memory profiling using time and tracemalloc
- Efficiency and logical problem-solving

PRACTICAL NO.: 14

TITLE: Generate All Twin Prime Pairs up to a Given Limit

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

METHODOLOGY:

```
import time
import tracemalloc

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

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

limit = int(input("Enter the limit: "))
```

```
start_time = time.time()
tracemalloc.start()

result = twin_primes(limit)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()

print("\nTwin prime pairs up to", limit, "are:")
print(result)
print("Time taken: {:.6f} seconds".format(end_time - start_time))
print("Current memory usage: {:.2f} KB".format(current / 1024))
print("Peak memory usage: {:.2f} KB".format(peak / 1024))
```

RESULTS ACHIEVED:

Enter the limit: 2
Twin prime pairs up to 2 are:[]
Time taken: 0.000032 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.09 KB

CONCLUSION:

Skills acquired:

- Prime number checking logic
- Understanding of twin primes concept
- Looping and conditional statements
- Function definition and modular programming
- List creation and tuple handling
- Input/output handling

- Time and memory profiling using time and tracemalloc
- Logical thinking and pattern recognition
- Efficiency and algorithmic reasoning

PRACTICAL NO.: 15

TITLE: Generate All Twin Prime Pairs up to a Given Limit

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

METHODOLOGY:

```
import time

import tracemalloc


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


num = int(input("Enter a number: "))
```

```
start_time = time.time()

tracemalloc.start()


result = count_divisors(num)


current, peak = tracemalloc.get_traced_memory()

tracemalloc.stop()

end_time = time.time()


print("\nNumber of divisors:", result)

print("Time taken: {:.6f} seconds".format(end_time - start_time))

print("Current memory usage: {:.2f} KB".format(current / 1024))

print("Peak memory usage: {:.2f} KB".format(peak / 1024))
```

RESULTS ACHIEVED:

Enter a number: 2
Number of divisors: 2
Time taken: 0.000251 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB

CONCLUSION:

Skills acquired:

- Divisor counting logic
- Mathematical reasoning and factorization
- Looping and conditional statements

- Function definition and modular programming
- Handling perfect square cases
- Input/output handling
- Time and memory profiling using `time` and `tracemalloc`
- Efficiency and optimization awareness
- Logical thinking and problem decomposition

Practical No: 16

TITLE: **Aliquot Sum Calculator**

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

METHODOLOGY:

```
import time
import tracemalloc  # built-in tool to track memory usage


def aliquot_sum(n):
    """Return sum of all proper divisors of n (less than n)."""
    total = 0
    # only loop till sqrt(n) for efficiency
    i = 1
    while i * i <= n:
        if n % i == 0:
            if i != n:
                total += i
            other = n // i
            if other != i and other != n:
                total += other
        i += 1
    return total


# --- Performance Test ---
```

```
n = int(input("Enter a number: "))


tracemalloc.start()        # start memory tracking
start_time = time.time()    # start timer


result = aliquot_sum(n)


end_time = time.time()      # end timer
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()


print(f"\nAliquot sum of {n} = {result}")
print(f"Time taken: {end_time - start_time:.6f} seconds")
print(f"Memory used: {current / 1024:.3f} KB (current), {peak / 1024:.3f} KB (peak)")
```

RESULTS ACHIEVED:

Enter a number: 1
Aliquot sum of 1 = 0
Time taken: 0.000009 seconds
Memory used: 0.000 KB (current), 0.000 KB (peak)

CONCLUSION:

Skills acquired:

- Function Writing
- Mathematical Logic
- Time Measurement
- Memory Profiling
- Algorithmic Thinking
- Input/Output Handling
- Code Efficiency Awareness

Practical No: 17

TITLE: **Amicable Numbers Checker**

AIM/OBJECTIVE(s): To write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).

METHODOLOGY:

```
import time
import tracemalloc

def aliquot_sum(n):
    """Return the sum of all proper divisors of n."""
    total = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            if i != n:
                total += i
            other = n // i
            if other != i and other != n:
                total += other
        i += 1
    return total

def are_amicable(a, b):
```

```
    """Check if two numbers are amicable."""
    return aliquot_sum(a) == b and aliquot_sum(b) == a


# ---- Performance Test ----
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

tracemalloc.start()          # start tracking memory
start_time = time.time()     # start timer

result = are_amicable(a, b)

end_time = time.time()       # end timer
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

# ---- Output ----
if result:
    print(f"\n{a} and {b} are amicable numbers!")
else:
    print(f"\n{a} and {b} are NOT amicable numbers.")

print(f"Time taken: {end_time - start_time:.6f} seconds")
print(f"Memory used: {current / 1024:.3f} KB (current), {peak / 1024:.3f} KB (peak)")
```

RESULTS ACHIEVED:

Enter first number: 220
Enter second number: 284
220 and 284 are amicable numbers!
Time taken: 0.000044 seconds
Memory used: 0.750 KB (current), 0.750 KB (peak)

CONCLUSION:

Skills acquired:

- Function Writing
- Conditional Logic
- Mathematical Reasoning
- Time Measurement
- Memory Profiling
- Input/Output Handling
- Code Optimization Awareness

Practical No: 18

TITLE: Multiplicative Persistence Finder

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

METHODOLOGY:

```
import time
import tracemalloc


def multiplicative_persistence(n):
    """Return the number of steps needed to reach a single digit
    by repeatedly multiplying the digits of n."""
    count = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        count += 1
    return count


# ---- Performance Test ----
num = int(input("Enter a number: "))


tracemalloc.start()
```

```
start_time = time.time()


result = multiplicative_persistence(num)


end_time = time.time()

current, peak = tracemalloc.get_traced_memory()

tracemalloc.stop()


print(f"\nMultiplicative Persistence of {num}: {result}")

print(f"Time taken: {end_time - start_time:.6f} seconds")

print(f"Memory used: {current / 1024:.3f} KB (current), {peak / 1024:.3f} KB (peak)")
```

RESULTS ACHIEVED:

Enter a number: 50
Multiplicative Persistence of 50: 1
Time taken: 0.000028 seconds
Memory used: 0.000 KB (current), 0.116 KB (peak)

CONCLUSION:

Skills acquired:

- Function Writing
- Looping & Iteration
- String-to-Digit Manipulation
- Mathematical Reasoning
- Time Measurement
- Memory Profiling
- Input/Output Handling

Practical No: 19

TITLE: Highly Composite Number Detector

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

METHODOLOGY:

```
import time
import tracemalloc

def count_divisors(n):
    """Return the number of positive divisors of n."""
    count = 0
    i = 1
    while i * i <= n:
        if n % i == 0:
            count += 1
            if i != n // i:
                count += 1
        i += 1
    return count

def is_highly_composite(n):
    """Return True if n has more divisors than any smaller number."""
    div_n = count_divisors(n)
    for i in range(1, n):
        if count_divisors(i) >= div_n:
```

```
        return False
    return True

# ---- Performance Test ----
num = int(input("Enter a number: "))

tracemalloc.start()
start_time = time.time()

result = is_highly_composite(num)

end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"\nIs {num} highly composite? {result}")
print(f"Time taken: {end_time - start_time:.6f} seconds")
print(f"Memory used: {current / 1024:.3f} KB (current), {peak / 1024:.3f} KB (peak)")
```

RESULTS ACHIEVED:

Enter a number: 2
Is 2 highly composite? True
Time taken: 0.000026 seconds
Memory used: 0.750 KB (current), 0.750 KB (peak)

CONCLUSION:

Skills acquired:

- Function Writing
- Divisor Counting Logic

- Comparative Reasoning
- Looping & Iteration
- Time Measurement
- Memory Profiling
- Input/Output Handling

Practical No: 20

TITLE: Modular Exponentiation (Fast Power Mod) Program

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

METHODOLOGY:

```
import time
import tracemalloc


def mod_exp(base, exponent, modulus):
    """Efficiently compute (base^exponent) % modulus."""
    result = 1
    base = base % modulus

    while exponent > 0:
        # If exponent is odd, multiply result with base
        if exponent % 2 == 1:
            result = (result * base) % modulus

        # Square the base
        base = (base * base) % modulus
        exponent //= 2

    return result
```

```
# ---- Performance Test ----
b = int(input("Enter base: "))
e = int(input("Enter exponent: "))
m = int(input("Enter modulus: "))

tracemalloc.start()
start_time = time.time()

answer = mod_exp(b, e, m)

end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"\nResult: {answer}")
print(f"Time taken: {end_time - start_time:.6f} seconds")
print(f"Memory used: {current / 1024:.3f} KB (current), {peak / 1024:.3f} KB (peak)")
```

RESULT ACHIEVED:

Enter base: 2
Enter exponent: 10
Enter modulus: 1000
Result: 24
Time taken: 0.000033 seconds
Memory used: 0.750 KB (current), 0.750 KB (peak)

CONCLUSION:

Skills acquired:

- Function Writing

- Fast Exponentiation Logic
- Mathematical Optimization
- Looping & Bitwise Reasoning
- Time Measurement
- Memory Profiling
- Input/Output Handling

Practical No: 21

TITLE: **Modular Multiplicative Inverse Finder**

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

METHODOLOGY:

```
import time
import tracemalloc


def mod_inverse(a, m):
    start_time = time.time()
    tracemalloc.start()

    # Extended Euclidean Algorithm
    def egcd(a, b):
        if b == 0:
            return a, 1, 0
        gcd, x1, y1 = egcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y

    gcd, x, _ = egcd(a, m)

    if gcd != 1:
```

```
        inverse = None
    else:
        inverse = x % m

    # Tracking usage
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    print(f"Time taken: {end_time - start_time} seconds")
    print(f"Memory used: {current} bytes; Peak: {peak} bytes")

    return inverse

# Example call
print("Modular inverse of 3 mod 11 is:")
output_value = mod_inverse(3, 11)
print("Output:", output_value)
```

RESULTS ACHIEVED:

Modular inverse of 3 mod 11 is:
Time taken: 2.5272369384765625e-05 seconds
Memory used: 160 bytes; Peak: 208 bytes
Output: 4

CONCLUSION:

Skills acquired:

- Understanding Modular Arithmetic
- Extended Euclidean Algorithm
- Computing Modular Inverses

- Recursion Technique
- Checking Coprimality
- Performance Tracking
- Cleaner Function Design

Practical No: 22

TITLE: **Chinese Remainder Theorem Solver**

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

METHODOLOGY:

```
import time
import tracemalloc

def crt(remainders, moduli):
    start_time = time.time()
    tracemalloc.start()

    # Step 1: Compute the product of all moduli
    M = 1
    for m in moduli:
        M *= m

    # Helper: Extended Euclidean Algorithm
    def egcd(a, b):
        if b == 0:
            return a, 1, 0
        gcd, x1, y1 = egcd(b, a % b)
        return gcd, y1, x1 - (a // b) * y1
```

```
# Step 2: Apply CRT formula
x = 0
for r, m in zip(remainders, moduli):
    Mi = M // m
    gcd, inv, _ = egcd(Mi, m)

    if gcd != 1:
        print("Error: Moduli are not pairwise coprime. No unique solution.")
        tracemalloc.stop()
        return None

    inv %= m
    x += r * Mi * inv

# Final result modulo M
x %= M

# Performance tracking
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()

print(f"Time taken: {end_time - start_time} seconds")
print(f"Memory used: {current} bytes; Peak: {peak} bytes")

return x
```

Example usage

remainders = [2, 3, 2]

moduli = [3, 5, 7]

print("CRT Solution:", crt(remainders, moduli))

RESULTS ACHIEVED:

Time taken: 0.0009455680847167969 seconds
Memory used: 160 bytes; Peak: 368 bytes
CRT Solution: 23

CONCLUSION:

Skills acquired:

- Understanding the Chinese Remainder Theorem
- Working with Pairwise Coprime Moduli
- Using the Extended Euclidean Algorithm for Inverses
- Implementing Multi-Equation Modular Systems
- Handling Large Integer Arithmetic
- Using Performance Tracking (time + memory)
- Structuring Algorithms with Helper Functions

Practical No: 23

TITLE: Quadratic Residue Checker

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

METHODOLOGY:

```
import time
import tracemalloc

def is_quadratic_residue(a, p):
    start_time = time.time()
    tracemalloc.start()

    # Euler's Criterion: a^((p-1)/2) mod p
    result = pow(a, (p - 1) // 2, p)

    # 1 → quadratic residue
    # p-1 (i.e., -1 mod p) → non-residue
    if result == 1:
        answer = True
    else:
        answer = False

    # Performance tracking
    current, peak = tracemalloc.get_traced_memory()
```

```
    tracemalloc.stop()
    end_time = time.time()

    print(f"Time taken: {end_time - start_time} seconds")
    print(f"Memory used: {current} bytes; Peak: {peak} bytes")

    return answer

# Example:
print("Is 5 a quadratic residue mod 11?")
print(is_quadratic_residue(5, 11))
```

RESULTS ACHIEVED:

Is 5 a quadratic residue mod 11?
Time taken: 2.288818359375e-05 seconds
Memory used: 0 bytes; Peak: 0 bytes
True

CONCLUSION:

Skills acquired:

- Understanding Quadratic Residues
- Applying Euler’s Criterion
- Using Fast Modular Exponentiation
- Working with Prime Moduli
- Logical Interpretation of Modular Results
- Incorporating Time & Memory Profiling
- Writing Clean Boolean-Returning Functions

Practical No: 24

TITLE: Multiplicative Order Calculator (Order Mod n)

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

METHODOLOGY:

```
import time

import tracemalloc


def order_mod(a, n):

    start_time = time.time()

    tracemalloc.start()


    # order only exists if gcd(a, n) = 1

    def gcd(x, y):

        while y != 0:

            x, y = y, x % y

        return x


    if gcd(a, n) != 1:

        print("Order does not exist because a and n are not coprime.")

        tracemalloc.stop()
```

```
        return None

    k = 1
    value = pow(a, k, n)

    # keep increasing k until a^k ≡ 1 mod n
    while value != 1:
        k += 1
        value = pow(a, k, n)

    # performance tracking
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    end_time = time.time()

    print(f"Time taken: {end_time - start_time} seconds")
    print(f"Memory used: {current} bytes; Peak: {peak} bytes")

    return k

# Example
print("Order of 2 mod 7:")
print(order_mod(2, 7))
```

RESULTS ACHIEVED:

Order of 2 mod 7:
Time taken: 4.410743713378906e-05 seconds
Memory used: 160 bytes; Peak: 160 bytes
3

CONCLUSION:

Skills acquired:

- Understanding Multiplicative Order
- Checking Coprimality with GCD
- Using Modular Exponentiation Efficiently
- Implementing Iterative Search for Order
- Applying Number Theory Concepts in Code
- Integrating Time & Memory Tracking
- Structuring Clear, Logical Functions

Practical No: 25

TITLE: Fibonacci Prime Checker

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

METHODOLOGY:

```
import time

import tracemalloc

import math


def is_fibonacci_prime(n):

    start_time = time.time()

    tracemalloc.start()


    # --- Helper: check prime ---

    def is_prime(x):

        if x < 2:

            return False

        if x == 2:

            return True

        if x % 2 == 0:

            return False
```

```
    for i in range(3, int(x**0.5) + 1, 2):
        if x % i == 0:
            return False
    return True


# --- Helper: check Fibonacci via perfect square test ---
# A number n is Fibonacci iff (5n² + 4) or (5n² - 4) is a perfect square
def is_perfect_square(x):
    r = int(math.isqrt(x))
    return r * r == x


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


# Main logic
if is_fibonacci(n) and is_prime(n):
    result = True
else:
    result = False


# --- Performance tracking ---
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
end_time = time.time()
```

```
    print(f"Time taken: {end_time - start_time} seconds")

    print(f"Memory used: {current} bytes; Peak: {peak} bytes")


    return result


# Example

print("Is 13 a Fibonacci prime?")

print(is_fibonacci_prime(13))
```

RESULTS ACHIEVED:

Is 13 a Fibonacci prime?
Time taken: 9.322166442871094e-05 seconds
Memory used: 504 bytes; Peak: 592 bytes
True

CONCLUSION:

- Identifying Fibonacci Numbers Using the Perfect Square Test
- Implementing Prime Checking
- Combining Multiple Mathematical Conditions
- Using Helper Functions for Cleaner Code
- Applying Number Theory Logic in Programs
- Integrating Time & Memory Profiling
- Writing Efficient Mathematical Predicates

Practical No: 26

TITLE: Lucas Numbers Generator

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

METHODOLOGY:

```
import time

import tracemalloc


def lucas_sequence(n):

    # Start tracking

    start_time = time.time()

    tracemalloc.start()


    # Lucas numbers start with 2, 1

    if n <= 0:

        result = []

    elif n == 1:

        result = [2]

    else:

        result = [2, 1]
```

```
        for i in range(2, n):
            result.append(result[-1] + result[-2])

    # Stop tracking
    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Time + memory info
    print(f"Time taken: {end_time - start_time:.6f} seconds")
    print(f"Memory used: {current / 1024:.2f} KB (current), {peak / 1024:.2f} KB (peak)")

    return result

# Example output
print("Example (first 10 Lucas numbers):")
print(lucas_sequence(10))
```

RESULTS ACHIEVED:

Example (first 10 Lucas numbers):
Time taken: 0.000030 seconds
Memory used: 0.12 KB (current), 0.16 KB (peak)
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]

CONCLUSION:

Skills acquired:

- Understanding of recurrence relations
- Ability to generate number sequences programmatically
- Working with list operations in Python
- Implementing iterative algorithms
- Using time and memory profiling tools (time, tracemalloc)
- Writing clean and efficient Python functions

Practical No: 27

TITLE: Perfect Power Checker

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

METHODOLOGY:

```
import time

import tracemalloc

import math


def is_perfect_power(n):

    # Start tracking

    start_time = time.time()

    tracemalloc.start()


    if n <= 1:

        result = False

    else:

        result = False

        # Try all possible exponents b

        for b in range(2, int(math.log2(n)) + 2):
```

```
        # Compute possible base a
        a = round(n ** (1 / b))
        if a > 1 and a ** b == n:
            result = True
            break

    # Stop tracking
    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Time + memory info
    print(f"Time taken: {end_time - start_time:.6f} seconds")
    print(f"Memory used: {current / 1024:.2f} KB (current), {peak / 1024:.2f} KB
(peak)")

    return result

# Example usage
print("Example checks:")
print("8  →", is_perfect_power(8))     # 2^3
print("12 →", is_perfect_power(12))    # Not a perfect power
print("81 →", is_perfect_power(81))    # 3^4 or 9^2
```

RESULTS ACHIEVED:

Example checks:

Time taken: 0.000047 seconds

Memory used: 0.05 KB (current), 0.16 KB (peak)

8 → True

CONCLUSION:

Skills acquired:

- Understanding of exponential number properties
- Implementing mathematical checks using logarithms
- Using iterative search to test multiple exponent–base combinations
- Applying numerical methods (root extraction + rounding)
- Writing efficient condition-based logic in Python
- Profiling program performance using `time` and `tracemalloc`
- Strengthening algorithmic thinking for number theory problems

Practical No: 28

TITLE: Collatz Sequence Length Calculator

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

METHODOLOGY:

```
import time

import tracemalloc


def collatz_length(n):

    # Start tracking

    start_time = time.time()

    tracemalloc.start()


    steps = 0

    x = n


    while x != 1:

        if x % 2 == 0:

            x //= 2

        else:
```

```python
        x = 3 * x + 1

    steps += 1


  # Stop tracking

  end_time = time.time()

  current, peak = tracemalloc.get_traced_memory()

  tracemalloc.stop()


  # Time + memory info

  print(f"Time taken: {end_time - start_time:.6f} seconds")

  print(f"Memory used: {current / 1024:.2f} KB (current), {peak / 1024:.2f} KB (peak)")


  return steps


# Example usage

print("Example:")

print("Collatz length of 12 →", collatz_length(12))
```

RESULTS ACHIEVED:

Example:
Time taken: 0.000016 seconds
Memory used: 0.00 KB (current), 0.00 KB (peak)
Collatz length of 12 → 9

CONCLUSION:

Skills acquired:

- Implementing iterative algorithms with conditional logic
- Working with mathematical sequences and conjectures
- Using loops efficiently to simulate step-based processes
- Handling integer operations and transformations
- Applying performance measurement tools (`time`, `tracemalloc`)
- Strengthening logical reasoning through problem-solving
- Understanding behavior of recursive-style sequences without recursion

Practical No: 29

TITLE: Polygonal Number Calculator

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

METHODOLOGY:

```
import time

import tracemalloc


def polygonal_number(s, n):

    """

    Return the n-th s-gonal number.

    s : int, number of sides (must be >= 3)

    n : int, index in sequence (must be >= 1)

    """

    # Start tracking

    start_time = time.time()

    tracemalloc.start()


    # Input validation

    if not isinstance(s, int) or not isinstance(n, int):

        raise TypeError("s and n must be integers.")
```

```
    if s < 3:
        raise ValueError("s must be >= 3 (3 = triangular).")
    if n < 1:
        raise ValueError("n must be >= 1.")

    # Compute using integer arithmetic
    result = ((s - 2) * n * n - (s - 4) * n) // 2

    # Stop tracking
    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Time + memory info
    print(f"Time taken: {end_time - start_time:.6f} seconds")
    print(f"Memory used: {current / 1024:.2f} KB (current), {peak / 1024:.2f} KB (peak)")

    return result

# Example outputs
print("Examples:")
print("Triangular (s=3)  n=5  →", polygonal_number(3, 5))   # 15
print("Square     (s=4)  n=5  →", polygonal_number(4, 5))   # 25
print("Pentagonal (s=5)  n=4  →", polygonal_number(5, 4))   # 22
```

RESULTS ACHIEVED:

Examples:
Time taken: 0.000011 seconds
Memory used: 0.00 KB (current), 0.00 KB (peak)
Triangular (s=3) n=5 → 15

CONCLUSION:

Skills acquired:

- Applying mathematical formulas to compute number sequences
- Understanding and using the polygonal number formula
- Implementing input validation for robust functions
- Strengthening arithmetic and algebraic computation in Python
- Using performance profiling tools (`time`, `tracemalloc`)
- Writing reusable, parameterized functions
- Improving accuracy using integer arithmetic instead of floats

Practical No: 30

TITLE: Carmichael Number Checker

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

METHODOLOGY:

```
import time

import tracemalloc

import math


# Helper to compute gcd

def gcd(a, b):

    while b:

        a, b = b, a % b

        return a


def is_carmichael(n):

    # Start tracking

    start_time = time.time()

    tracemalloc.start()
```

```
    # Must be composite
    if n < 3 or is_prime(n := n):  # using walrus just for neatness
        tracemalloc.stop()
        return False

    # Carmichael test:
    # For all a coprime to n: a^(n-1) ≡ 1 (mod n)
    result = True
    for a in range(2, n):
        if math.gcd(a, n) == 1:
            if pow(a, n - 1, n) != 1:
                result = False
                break

    # Stop tracking
    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Time + memory info
    print(f"Time taken: {end_time - start_time:.6f} seconds")
    print(f"Memory used: {current / 1024:.2f} KB (current), {peak / 1024:.2f} KB (peak)")

    return result
```

```
# Quick prime check for composite detection
def is_prime(x):
    if x <= 1:
        return False
    if x <= 3:
        return True
    if x % 2 == 0 or x % 3 == 0:
        return False
    i = 5
    while i * i <= x:
        if x % i == 0 or x % (i + 2) == 0:
            return False
        i += 6
    return True


# Example usage
print("Examples:")
print("561  →", is_carmichael(561))   # True, classic Carmichael
print("1105 →", is_carmichael(1105))  # True
print("15   →", is_carmichael(15))    # False
```

RESULTS ACHIEVED:

Examples:
Time taken: 0.003422 seconds
Memory used: 0.05 KB (current), 0.16 KB (peak)
561 → True

CONCLUSION:

Skills acquired:

- Understanding Fermat's Little Theorem and its exceptions
- Identifying and validating composite numbers vs primes
- Implementing modular exponentiation using Python's built-in `pow()`
- Working with gcd to test coprimality
- Writing efficient loops for mathematical verification
- Profiling runtime and memory with `time` and `tracemalloc`
- Strengthening number-theory reasoning and algorithm design

Practical No: 31

TITLE: Miller–Rabin Primality Test

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

METHODOLOGY:

```
import time
import tracemalloc
import secrets


def is_prime_miller_rabin(n: int, k: int = 5) -> bool:
    """
    Probabilistic Miller-Rabin primality test.
    n : number to test
    k : number of rounds
    """
    if n < 2:
        return False

    # Quick small primes
    small_primes = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
    if n in small_primes:
```

```
    return True

if n % 2 == 0:
    return False

# Write n - 1 = 2^s * d (d odd)
d = n - 1
s = 0
while d % 2 == 0:
    d //= 2
    s += 1

# k rounds
for _ in range(k):
    a = 2 + secrets.randbelow(max(1, n - 3))   # random base
    x = pow(a, d, n)

    if x == 1 or x == n - 1:
        continue

    composite = True
    for _ in range(s - 1):
        x = (x * x) % n
        if x == n - 1:
```

```
            composite = False
            break

        if composite:
            return False

    return True


def measure_time_memory(fn, *args):
    """Measure execution time + peak memory for a function call."""
    tracemalloc.start()
    t0 = time.perf_counter()

    result = fn(*args)

    t1 = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    return result, t1 - t0, peak / 1024  # KiB


# Example usage:
```

```
n = 2147483647   # change number here

k = 10           # number of rounds


res, elapsed, memory = measure_time_memory(is_prime_miller_rabin, n, k)


print(f"Number: {n}")

print(f"Rounds: {k}")

print(f"Probable Prime? {res}")

print(f"Time: {elapsed:.6f} seconds")

print(f"Peak Memory: {memory:.2f} KiB")
```

RESULTS ACHIEVED:

Number: 2147483647
Rounds: 10
Probable Prime? True
Time: 0.002280 seconds
Peak Memory: 0.37 KB

CONCLUSION:

Skills acquired:

- Modular Exponentiation
- Number Theory Concepts
- Probabilistic Algorithm Design
- Cryptographic Applications
- Efficient Factor Decomposition
- Time and Memory Profiling
- Secure Random Number Usage
- Algorithm Optimization

Practical No: 32

TITLE: Pollard's Rho Algorithm

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

METHODOLOGY:

```
import time

import tracemalloc

import secrets

import math

from typing import List, Optional


def is_probable_prime(n: int, k: int = 8) -> bool:

    """Miller-Rabin probabilistic primality test."""

    if n < 2:

        return False

    small_primes = (2,3,5,7,11,13,17,19,23,29)

    for p in small_primes:

        if n == p:

            return True

        if n % p == 0:

            return False
```

```
    # write n-1 = 2^s * d with d odd
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1
    for _ in range(k):
        # choose a in [2, n-2]
        a = secrets.randbelow(n - 3) + 2 if n > 4 else 2
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        composite = True
        for _ in range(s - 1):
            x = (x * x) % n
            if x == n - 1:
                composite = False
                break
        if composite:
            return False
    return True


def pollard_rho(n: int, max_iter: int = 10000, max_restarts: int = 10) -> Optional[int]:
    """
```

```
Pollard's Rho: return a non-trivial factor of n, or None on failure.

Not guaranteed to return prime factor; may return composite factor.

"""

if n % 2 == 0:

    return 2

if n % 3 == 0:

    return 3

if n <= 3:

    return None

if is_probable_prime(n):

    return n


# small trial division to quickly strip tiny factors

for p in (5,7,11,13,17,19,23,29,31,37,41,43,47,53,59):

    if n % p == 0:

        return p


for restart in range(max_restarts):

    # choose random polynomial x^2 + c (mod n), with c in [1, n-1]

    c = secrets.randbelow(n - 1) + 1  # 1..n-1

    # choose random start x in [2, n-2]

    x = secrets.randbelow(n - 4) + 2 if n > 5 else 2

    y = x

    d = 1
```

```
    for iteration in range(max_iter):
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        d = math.gcd(abs(x - y), n)
        if d == 1:
            continue
        if d == n:
            # failure for this polynomial — break to restart
            break
        # found non-trivial factor
        return d
    # failed to find factor
    return None


def factorize(n: int) -> List[int]:
    """Fully factor n into prime factors (not necessarily sorted)."""
    if n == 1:
        return []
    if is_probable_prime(n):
        return [n]
    # strip small primes
    for p in (2,3,5,7,11,13,17,19,23,29):
```

```
        while n % p == 0:
            n //= p
            # accumulate p and continue factoring the reduced n
            return [p] + factorize(n)
    f = pollard_rho(n)
    if f is None or f == n:
        # fallback to trial division (rare)
        limit = int(math.isqrt(n)) + 1
        for i in range(2, limit):
            if n % i == 0:
                return factorize(i) + factorize(n // i)
        return [n]
    # recursively factor the found factor and the cofactor
    return factorize(f) + factorize(n // f)


def measure_time_memory(fn, *args, **kwargs):
    """Return (result, elapsed_seconds, peak_kib)."""
    tracemalloc.start()
    t0 = time.perf_counter()
    result = fn(*args, **kwargs)
    t1 = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return result, (t1 - t0), peak / 1024.0
```

```
# ------------------------
# Example usage
# ------------------------
if __name__ == "__main__":
    examples = [
        8051,          # 83 * 97
        10403,         # 101 * 103
        600851475143,  # big composite (example from Project Euler)
        2_147_483_647, # prime (Mersenne 31)
    ]

    for n in examples:
        factors, elapsed, peak_kib = measure_time_memory(factorize, n)
        print(f"n = {n}")
        print(f"  factors (unsorted) : {factors}")
        print(f"  factors (sorted)   : {sorted(factors)}")
        print(f"  time               : {elapsed:.6f} s")
        print(f"  peak memory        : {peak_kib:.2f} KiB")
        print("-" * 50)
```

RESULT ACHIEVED:

```
n = 8051
  factors (unsorted) : [83, 97]
  factors (sorted)   : [83, 97]
```

time : 0.000494 s
peak memory : 0.30 KiB

CONCLUSION:

Skills acquired:

- Randomized Factorization Techniques
- Cycle Detection Methods
- GCD-Based Factor Extraction
- Probabilistic Primality Testing
- Recursive Decomposition Strategies
- Optimization with Trial Division
- Time and Memory Performance Analysis
- Robust Error and Edge-Case Handling

Practical No: 33

TITLE: Riemann Zeta Function Approximation

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

METHODOLOGY:

```
import time

import tracemalloc


def zeta_approx(s, terms):

    """

    Approximate the Riemann zeta function ζ(s) using the

    first 'terms' of the series sum(1 / n^s).

    """

    total = 0.0

    for n in range(1, terms + 1):

        total += 1 / (n ** s)

    return total


def measure_time_memory(fn, *args, **kwargs):

    """Measure execution time + peak memory usage."""

    tracemalloc.start()
```

```
    start = time.perf_counter()

    result = fn(*args, **kwargs)

    end = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    return result, end - start, peak / 1024  # KiB

# Example usage
if __name__ == "__main__":
    s = 2
    terms = 100000

    value, elapsed, mem = measure_time_memory(zeta_approx, s, terms)

    print(f"ζ({s}) approximation = {value}")
    print(f"Time elapsed        = {elapsed:.6f} seconds")
    print(f"Peak memory         = {mem:.2f} KiB")
```

RESULT ACHIEVED:

```
ζ(2) approximation = 1.6449240668982423
Time elapsed        = 0.166138 seconds
Peak memory         = 0.15 KiB
```

CONCLUSION:

Skills acquired:

- Infinite Series Approximation
- Exponentiation and Floating-Point Computation
- Numerical Analysis Basics
- Performance Measurement Techniques
- Efficient Loop-Based Summation
- Handling Mathematical Functions in Code

Practical No: 34

TITLE: Partition Function Calculator

AIM/OBJECTIVE(s): To write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers.

METHODOLOGY:

```
import time
import tracemalloc


def partition_function(n):
    """
    Compute the partition number p(n) using Euler's pentagonal
    number theorem recurrence.
    """
    if n < 0:
        return 0
    p = [0] * (n + 1)
    p[0] = 1  # base case

    for i in range(1, n + 1):
        total = 0
```

```
        k = 1
        while True:
            # generalized pentagonal numbers
            gp1 = k * (3 * k - 1) // 2
            gp2 = k * (3 * k + 1) // 2

            if gp1 > i:
                break

            sign = -1 if (k % 2 == 0) else 1  # + + − − + + ...

            total += sign * p[i - gp1]

            if gp2 <= i:
                total += sign * p[i - gp2]

            k += 1

        p[i] = total

    return p[n]

def measure_time_memory(fn, *args):
    """Measure execution time + peak memory usage."""
```

```
    tracemalloc.start()
    start = time.perf_counter()

    result = fn(*args)

    end = time.perf_counter()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    return result, end - start, peak / 1024  # KiB

# Example usage
if __name__ == "__main__":
    n = 50  # change n as needed
    value, elapsed, mem = measure_time_memory(partition_function, n)

    print(f"p({n}) = {value}")
    print(f"Time elapsed = {elapsed:.6f} seconds")
    print(f"Peak memory  = {mem:.2f} KiB")
```

RESULT ACHIEVED:

p(50) = 204226
Time elapsed = 0.000505 seconds
Peak memory = 1.56 KiB

CONCLUSION:

Skills acquired:

- Understanding of integer partitions
- Recursion and memoization
- Breaking down a problem into subproblems
- Efficient caching to avoid recalculating states
- Dynamic programming thinking