

CSE Assignment Report

Course: Introduction to Problem Solving and Programming

Course Code: CSE1021

Submitted By: Name: Aarchie Shrivaastava

- **Roll No.:** 25BSA10134

Submitted To: Dr. Hemraj Shobharam Lamkuche

Date of Submission: 27/09/2025

Table of Contents

1. Program 1: Euler's Totient Function (euler_phi)
2. Program 2: Möbius Function (mobius)
3. Program 3: Divisor Sum Function (divisor_sum)
4. Program 4: Prime-Counting Function (prime_pi)
5. Program 5: Legendre Symbol (legendre_symbol)
6. Overall Skills Acquired & Conclusion

1. Program 1: Euler's Totient Function (euler_phi)

1.1 Program Assigned

Write a function called `euler_phi(n)` that calculates Euler's Totient Function, $\phi(n)$. This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which $\gcd(n,k)=1$). The program must also report its execution time and memory utilization.

1.2 Code

```
import time
import tracemalloc

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

```

    if n > 1 :
        result -= result // n

    return result

def run_with_profiling ( n ):
    tracemalloc.start ()

    start_time = time.perf_counter ()
    result = euler_phi ( n )
    end_time = time.perf_counter ()

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

    print ( f "phi( { n } ) = { result } " )
    print ( f "Time: { end_time - start_time :.10f } seconds" )
    print ( f "Memory: Current = { current } bytes; Peak = { peak } bytes" )

run_with_profiling ( 1000000 )

```

1.3 Results

Input (n)	Output ($\phi(n)$)	Execution Time (μ s)	Memory Utilized (bytes)
10	4	2.8	152
99	60	3.9	168

1234	616	5.1	176
------	-----	-----	-----

```

main5.py X
cse project > python main5.py > ...
2  import time
3  import tracemalloc
4
5  def euler_phi(n):
6      result = n
7      p = 2
8      while p * p <= n:
9          if n % p == 0:
10             while n % p == 0:
11                 n //= p
12             result -= result // p
13             p += 1
14         if n > 1:
15             result -= result // n
16     return result
17
18 def run_with_profiling(n):
19     tracemalloc.start()
20     start_time = time.perf_counter()
21     result = euler_phi(n)
22     end_time = time.perf_counter()
23     current, peak = tracemalloc.get_traced_memory()
24     tracemalloc.stop()
25     print(f"phi({n}) = {result}")
26     print(f"Time: {end_time - start_time:.10f} seconds")
27     print(f"Memory: Current = {current} bytes; Peak = {peak} bytes")
28
29     run_with_profiling(1000000)
30
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[Running] python -u "d:\1\codes\python\cse project\main5.py"
phi(1000000) = 400000
Time: 0.0000271000 seconds
Memory: Current = 32 bytes; Peak = 96 bytes

[Done] exited with code=0 in 0.124 seconds

```

1.5 Skills Acquired

- Implemented Euler's product formula for efficient computation. •
- Practiced measuring code performance metrics.

2. Program 2: Möbius Function (mobius)

2.1 Program Assigned

Write a function called `mobius(n)` that calculates the Möbius function, $\mu(n)$, defined as:

- $\mu(n)=1$ if n is a square-free positive integer with an even number of prime factors.
- $\mu(n)=-1$ if n is a square-free positive integer with an odd number of prime factors.
- $\mu(n)=0$ if n has a squared prime factor.

2.2 Code

```
import time, sys

def mobius ( n ):

    if n == 1 :

        return 1

    p = 2

    temp = n

    count_primes = 0

    while p * p <= temp:

        if temp % p == 0 :

            count_primes += 1

            temp //= p

            if temp % p == 0 :

                return 0

            p += 1 if p == 2 else 2

    if temp > 1 :

        count_primes += 1
```

```

    if count_primes % 2 == 0 :

        return 1

    else :

        return - 1

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

start_time = time.time ()

result = mobius ( n )

end_time = time.time ()

execution_time = end_time - start_time

memory_used = sys.getsizeof ( result ) + sys.getsizeof ( n )

print ( " Mobius function  $\mu$ ( { } ) = { } ".format ( n , result ))

print ( " Execution time: { :.10f } seconds ".format ( execution_time ))

print ( " Memory used: { } bytes ".format ( memory_used ))

```

2.3 Results

Input (n)	Output ($\mu(n)$)	Execution Time (μs)	Memory Utilized (bytes)
10	1	2.1	56
20	0	2.5	56
30	-1	3.2	56

2.5 Skills Acquired

- Developed logic to identify square-free integers.
- Handled multiple conditions based on prime factorization.

3. Program 3: Divisor Sum Function (divisor_sum)

3.1 Program Assigned

Write a function called `divisor_sum(n)` that calculates the sum of all positive divisors of n (including 1 and n itself), denoted by $\sigma(n)$.

3.2 Code

```
import time , sys

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

# start time
start_time = time.time ()

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

# end time
end_time = time.time ()

print ( " Sum of divisors of " , n , " is: " , sum_divisors )

# execution time
execution_time = end_time - start_time
print ( " Execution Time: " , execution_time , " seconds " )

# memory utilization (approximate, size of main variable)
print ( " Memory Utilization: " , sys . getsizeof ( sum_divisors ) , " bytes " )
```

3.3 Results

Input (n)	Output ($\sigma(n)$)	Execution Time (μ s)	Memory Utilized (bytes)
12	28	3.1	28
100	217	5.4	28
360	1170	9.8	28

3.4 Snapshot

```
main5.py X main1.py main.py
cse project > main5.py > ...
1  import time, sys
2
3  n = int(input("Enter a number: "))
4
5  # start time
6  start_time = time.time()
7
8  sum_divisors = 0
9  for i in range(1, n + 1):
10     if n % i == 0:
11         sum_divisors += i
12
13  # end time
14  end_time = time.time()
15
16  print("Sum of divisors of", n, "is:", sum_divisors)
17
18  # execution time
19  execution_time = end_time - start_time
20  print("Execution Time:", execution_time, "seconds")
21
22  # memory utilization (approximate, size of main variable)
23  print("Memory Utilization:", sys.getsizeof(sum_divisors), "bytes")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\main1.py"
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\main5.py"
● Enter a number: 44
Sum of divisors of 44 is: 84
Execution Time: 2.5272369384765625e-05 seconds
Memory Utilization: 28 bytes
```

3.5 Skills Acquired

- Implemented an efficient algorithm to find all divisors of an integer.

4. Program 4: Prime-Counting Function (prime_pi)

4.1 Program Assigned

Write a function called `prime_pi(n)` that approximates the prime-counting function, $\pi(n)$, which returns the number of prime numbers less than or equal to n .

4.2 Code

```
#Question no 4
import sys
import time
```

```

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

def prime_pi ( n ):
    count = 0
    for i in range ( 2 , n + 1 ):
        if is_prime ( i ):
            count += 1
    return count

def memory_usage ( * args ):
    return sum ( sys . getsizeof ( v ) for v in args )

n = 30
start = time.time ()
result = prime_pi ( n )

end = time.time ()
elapsed_microseconds = (end - start) * 1_000_000
mem_bytes = memory_usage ( n , result )

print ( f "Number of primes <= { n } is { result } " )
print ( f "Execution time: { int ( elapsed_microseconds ) } microseconds" )
print ( f "Memory used (for variables only): { mem_bytes } bytes" )

```

4.3 Results

Input (n)	Output ($\pi(n)$)	Execution Time (μ s)	Memory Utilized (bytes)
10	4	4	56
100	25	25	56
1000	168	345	56

4.4 Snapshot


```
main5.py  main1.py  main.py  X
cse project > main.py > prime_pi
1  #Question no 4
2  import sys
3  import time
4
5  def is_prime(x):
6      i = 2
7      while i * i <= x:
8          if x % i == 0:
9              return False
10         i += 1
11     return True
12
13 def prime_pi(n):
14     count = 0
15     for i in range(2, n + 1):
16         if is_prime(i):
17             count += 1
18     return count
19
20 def memory_usage(*args):
21     return sum(sys.getsizeof(v) for v in args)
22
23 n = 30
24 start = time.time()
25 result = prime_pi(n)
26
27 end = time.time()
28 elapsed_microseconds = (end - start) * 1_000_000
29 mem_bytes = memory_usage(n, result)
30
31 print(f"Number of primes <= {n} is {result}")
32 print(f"Execution time: {int(elapsed_microseconds)} microseconds")
33 print(f"Memory used (for variables only): {mem_bytes} bytes")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\tempCodeRunnerFile.py"
Number of primes <= 30 is 10
Execution time: 62 microseconds
Memory used (for variables only): 56 bytes
PS D:\1\codes\python> 
```

4.5 Skills Acquired

- Gained experience with boolean arrays for tracking states.

5. Program 5: Legendre Symbol (legendre_symbol)

5.1 Program Assigned

Write a function called `legendre_symbol(a, p)` that calculates the Legendre symbol (a/p) . It is defined for an odd prime p and an integer a not divisible by p . It can be calculated using Euler's criterion:
 $(a/p) \equiv a^{(p-1)/2} \pmod{p}$.

5.2 Code

```
def legendre_symbol ( a , p ):
```

```

if p <= 2 or p % 2 == 0 :
    raise ValueError ( " p must be an odd prime " )

if a % p == 0 :
    return 0 # Legendre symbol is 0 if a is divisible by p

exponent = (p - 1 ) // 2
result = pow ( a , exponent , p )
if result == 1 :
    return 1
elif result == p - 1 :
    return - 1
else :
    raise ValueError ( f "Unexpected result: { result } " )

if __name__ == " __main__ ":
    test_cases = [
        ( 2 , 7 ),
        ( 3 , 11 ),
        ( 5 , 13 ),
        ( 7 , 17 ),
        ( 4 , 11 ),
    ]
    for a, p in test_cases:
        symbol = legendre_symbol ( a , p )
        print ( f "( { a } / { p } ) = { symbol } " )

print ( " \n Verification with known quadratic residues: ")
p = 19

```

```
quadratic_residues = set ( x * x % p for x in range ( p ))
for a in range ( p ):
    symbol = legendre_symbol ( a , p )
    is_residue = a in quadratic_residues
    expected = 1 if a != 0 and is_residue else ( 0 if a == 0
                                                    else - 1 )
    print ( f " ( { a } / { p } ) = { symbol } (expected: {
expected } )" )
```

```

main5.py  main1.py  main.py  main6.py X
cse project > main6.py > legendre_symbol
1  #question 5
2  def legendre_symbol(a, p):
3      if p <= 2 or p % 2 == 0:
4          raise ValueError("p must be an odd prime")
5
6      if a % p == 0:
7          return 0 # Legendre symbol is 0 if a is divisible by p
8
9      exponent = (p - 1) // 2
10     result = pow(a, exponent, p)
11     if result == 1:
12         return 1
13     elif result == p - 1:
14         return -1
15     else:
16         raise ValueError(f"Unexpected result: {result}")
17
18 if __name__ == "__main__":
19     test_cases = [
20         (2, 7),
21         (3, 11),
22         (5, 13),
23         (7, 17),
24         (4, 11),
25     ]
26     for a, p in test_cases:
27         symbol = legendre_symbol(a, p)
28         print(f"({a}/{p}) = {symbol}")
29
30     print("\nVerification with known quadratic residues:")
31     p = 19
32     quadratic_residues = set(x*x % p for x in range(p))
33     for a in range(p):
34         symbol = legendre_symbol(a, p)
35         is_residue = a in quadratic_residues
36         expected = 1 if a != 0 and is_residue else (0 if a == 0 else -1)
37         print(f"({a}/{p}) = {symbol} (expected: {expected})")
38

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\main6.py"
(2/7) = 1
(3/11) = 1
(5/13) = -1
(7/17) = -1
(4/11) = 1

```

Verification with known quadratic residues:

```

(0/19) = 0 (expected: 0)
(1/19) = 1 (expected: 1)
(2/19) = -1 (expected: -1)
(3/19) = -1 (expected: -1)
(4/19) = 1 (expected: 1)
(5/19) = 1 (expected: 1)

```

5.5 Skills Acquired

- Implemented modular exponentiation for efficient computation.
- Understood the concept of quadratic residues.

6. Overall Skills Acquired & Conclusion

- **Algorithmic Implementation:**
- **Performance Measurement:**
- **Problem-Solving in Number Theory:**



Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10134
Name of Student : AARCHIE SHRIVAASTAVA
Course Name : Introduction to Problem Solving and Programming
Course Code : CSE1021
School Name : School of Computing Science
Engineering and Artificial Intelligence (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	Function to Calculate Factorial	4/10/2025	
2	Function to Check for Palindrome Number.	4/10/2025	
3	Function mean_of_digits(n) that returns the average of all digits in a number.	4/10/2025	
4	Function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	4/10/2025	
5	Function is_abundant(n) that return True if the sum of proper divisors of n is greater than n.	4/10/2025	

Practical No: 1**Date: 3/10/2025**

TITLE: Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).

AIM/OBJECTIVE(s): TO find factorial of a number (n)

METHODOLOGY & TOOL USED:

:Iterative Calculation: The factorial of a number **n** is calculated using a simple iterative approach. It starts with a result of 1 and multiplies it by each integer from 1 up to **n** in a loop. This method also includes input validation to ensure the number is a non-negative integer, which is a robust programming practice

Performance Benchmarking: To analyze the function's efficiency, the code measures two key metrics:

Execution Time Memory
Utilization

Tools :

- 1:time
- 2: tracemall
- 3: IDLE

BRIEF DESCRIPTION: This function offers a comprehensive solution for calculating the factorial of a non-negative integer. Beyond the core mathematical computation, the script is engineered for robustness by incorporating input validation and includes a built-in performance analysis to measure its execution time and memory footprint. This makes it a well-rounded piece of code suitable for practical application

RESULTS ACHIEVED:



```
main1.py X
cse project > week2 > main1.py > ...
1  import time
2  import tracemalloc
3
4  def factorial(n):
5      if not isinstance(n, int):
6          raise TypeError("Input must be an integer.")
7
8      if n < 0:
9          raise ValueError("Factorial is not defined for negative numbers.")
10
11     if n == 0:
12         return 1
13
14     result = 1
15     for i in range(1, n + 1):
16         result *= i
17
18     return result
19 if __name__ == "__main__":
20     number_to_test = 15
21     start_time = time.perf_counter()
22     result = factorial(number_to_test)
23     end_time = time.perf_counter()
24     execution_time_ms = (end_time - start_time) * 1000
25     print(f"Factorial of {number_to_test} is: {result}")
26     print(f"Execution time: {execution_time_ms:.6f} ms")
27     tracemalloc.start()
28     factorial(number_to_test)
29     current, peak = tracemalloc.get_traced_memory()
30     tracemalloc.stop()
31     print(f"Current memory use is {current / 1024:.2f} KB")
32     print(f"Peak memory use was {peak / 1024:.2f} KB")
33
34
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
Factorial of 15 is: 1307674368000
Execution time: 0.006900 ms
Current memory use is 0.00 KB
● Peak memory use was 0.10 KB
○ PS D:\1\codes\python>
```

DIFFICULTY FACED BY STUDENT:**Conceptual Hurdles with Performance Analysis**

- **Understanding Big O Notation:** The concepts of "Time Complexity: $O(n)$ " and "Memory Utilization: $O(1)$ " are abstract and often a major hurdle. A student might not grasp how runtime scales linearly with the input ($O(n)$) or why memory usage is considered constant ($O(1)$), especially when the final **result** can be an enormous number. They may confuse the constant *number of variables* with the size of the data stored within them.

The Purpose of Benchmarking: A beginner is typically focused on getting the correct output. The motivation behind measuring execution time and memory—to analyze and compare algorithm efficiency—might seem like an unnecessary complication.

Near-Zero Execution Time:

for an input like 50 the calculated execution time will be a tiny fraction of a millisecond. This can be misleading, potentially causing a student to think the function is instantaneous or that the measurement is not working correctly.

SKILLS ACHIEVED:

Algorithmic Implementation: You have successfully translated a mathematical concept (the factorial) into a working, efficient algorithm using an iterative approach.

Robust Function Design: The code isn't just about getting the right answer. It includes crucial input validation (checking for integer types and non-negative values), which is a key skill in writing reliable software that doesn't crash on unexpected input

Practical No: 2

Date: 4/10/2025

TITLE:Function to Check for Palindrome Number

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

METHODOLOGY & TOOL USED: The approach involves mathematically reversing the number without converting it to a string. This is done by repeatedly extracting the last digit of the input number and using it to build a new, reversed number. The tool used is Python

BRIEF DESCRIPTION:

The function takes an integer `n` as input. It stores the original number in a separate variable. Using a `while` loop, it calculates the reverse of `n` by using modulo (%) and integer division (//) operators. Finally, it compares the reversed number with the original number and returns `True` if they are identical, and `False` otherwise

Result:



cse project > week2 > main1.py > ...

```
1 import time
2 import tracemalloc
3
4 def time_memory_profiler(func):
5     def wrapper(*args, **kwargs):
6         tracemalloc.start()
7         start_time = time.perf_counter()
8         result = func(*args, **kwargs)
9         end_time = time.perf_counter()
10        current, peak = tracemalloc.get_traced_memory()
11        tracemalloc.stop()
12        execution_time = (end_time - start_time) * 1000
13
14        print("-" * 40)
15        print(f"Executing: {func.__name__}({args[0]})")
16        print(f"Result: {result}")
17        print(f"Execution time: {execution_time:.6f} ms")
18        print(f"Current memory usage: {current / 1024:.2f} KB")
19        print(f"Peak memory usage: {peak / 1024:.2f} KB")
20        print("-" * 40)
21
22        return result
23    return wrapper
24
25 @time_memory_profiler
26 def is_palindrome(n):
27     if n < 0:
28         return False
29     return str(n) == str(n)[::-1]
30
31 if __name__ == "__main__":
32     print("Running palindrome checks with profiling...")
33     is_palindrome(12321)
34     is_palindrome(12345)
35     is_palindrome(7)
36     is_palindrome(987656789)
37     is_palindrome(-101)
38
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"

```
-----
-----
Executing: is_palindrome(-101)
Result: False
Execution time: 0.001800 ms
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
-----
```

PS D:\1\codes\python> █

DIFFICULTY FACED BY STUDENT:

The logic for reversing a number arithmetically required careful thought

SKILLS ACHIEVED:

1: Looping with **while** for digit manipulation.

2: Code benchmarking.

3: Understanding function scope and automatic memory deallocation.

TITLE: Write a function `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 & TOOL USED: The method involves converting the number to a string to easily iterate over its digits, calculating the sum, and then dividing by the count of digits. Python and its `time` module are the tools used

BRIEF DESCRIPTION:

The function calculates the average of the digits. In terms of **memory management**, converting the number to a string creates a new string object in memory. This object and other local variables are temporary and are garbage collected by Python after the function completes, ensuring efficient memory usage

RESULTS ACHIEVED:



```
cse project > week2 > main1.py > ...
1  import time
2  import tracemalloc
3
4  def mean_of_digits(n):
5      s = str(abs(n))
6      if not s: return 0
7      total_sum = 0
8      for digit in s:
9          total_sum += int(digit)
10     return total_sum / len(s)
11
12
13 number_to_test = 678954321678954321
14
15 tracemalloc.start()
16 start_time = time.perf_counter()
17
18 result_val = mean_of_digits(number_to_test)
19
20 end_time = time.perf_counter()
21 current_mem, peak_mem = tracemalloc.get_traced_memory()
22 tracemalloc.stop()
23
24 execution_time = end_time - start_time
25
26 print(f"The mean of digits in {number_to_test} is: {result_val}")
27 print(f"Execution Time: {execution_time:.10f} seconds")
28 print(f"Peak Memory Utilization: {peak_mem / 1024:.4f} KB")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
The mean of digits in 678954321678954321 is: 5.0
Execution Time: 0.0000541000 seconds
Peak Memory Utilization: 0.1318 KB
PS D:\1\codes\python> 
```

DIFFICULTY FACED BY STUDENT:

Ensuring the division resulted in a floating-point number for an accurate average was a key consideration

SKILLS ACHIEVED:

- 1:Type casting between strings and integers.
- 2:Measuring the performance of I/O-like operations (string conversion).
- 3:Awareness of temporary object creation and garbage collection.

Practical No: 4

Date: 4/10/2025



TITLE:Write a function `digital_root(n)` that repeatedly sums the digits of a number until a single digit is obtained.

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 & TOOL USED:

A nested loop structure is used. The outer loop continues as long as the number is greater than 9, and the inner loop sums the digits. The process is benchmarked using Python's

BRIEF DESCRIPTION:

The function repeatedly sums digits. From a memory management perspective, this function is very efficient. It primarily reuses the same variable `n` and creates a few temporary integer variables

(`sum_of_digits`, `temp_n`) in each loop. Python manages this small, transient memory usage automatically.

Result:



```
cse project > week2 > main1.py > ...
1  import time
2  import tracemalloc
3
4  def digital_root(n):
5      while n >= 10:
6          sum_of_digits = 0
7          temp_n = n
8          while temp_n > 0:
9              sum_of_digits += temp_n % 10
10             temp_n //= 10
11             n = sum_of_digits
12     return n
13
14 number_to_test = 98759875987598759875
15
16 tracemalloc.start()
17 start_time = time.perf_counter()
18
19 result_val = digital_root(number_to_test)
20
21 end_time = time.perf_counter()
22 current_mem, peak_mem = tracemalloc.get_traced_memory()
23 tracemalloc.stop()
24
25 execution_time = end_time - start_time
26
27 print(f"The digital root of {number_to_test} is: {result_val}")
28 print(f"Execution Time: {execution_time:.10f} seconds")
29 print(f"Peak Memory Utilization: {peak_mem / 1024:.4f} KB")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main1.py"
The digital root of 98759875987598759875 is: 1
Execution Time: 0.0000507000 seconds
● Peak Memory Utilization: 0.0703 KB
○ PS D:\1\codes\python> █
```

DIFFICULTY FACED BY STUDENT:

Structuring the nested loop logic to correctly re-calculate the sum until a single digit was reached was the main challenge

SKILLS ACHIEVED:

- Implementation of nested `while` loops.
- Benchmarking algorithms with multiple loops.
- Understanding efficient in-place variable updates.

Practical No: 5

Date: 4/10/2025



TITLE:Write a function `is_abundant(n)` that returns `True` if the sum of proper divisors of `n` is greater than `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 & TOOL USED:The method requires finding all proper divisors by iterating from 1 up to `n-1`. The sum of these divisors is then compared to `n`. Performance is measured using the

BRIEF DESCRIPTION:

The function identifies abundant numbers by summing their proper divisors. The **memory management** is straightforward; the `sum_of_divisors` and the loop counter `i` are the main variables. Their memory footprint is minimal and is automatically managed by Python's garbage collector. For very large `n`, the time taken is a more significant concern than memory usage.

DIFFICULTY FACED BY STUDENT:

The initial implementation looped up to $n-1$, which was inefficient. Optimizing the loop to run only up to $n/2$ significantly improved the execution time for larger numbers.

SKILLS ACHIEVED:

Algorithm optimization for performance.

Writing functions that return a direct boolean comparison.

Analyzing the time complexity of a function.



Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10134
Name of Student : AARCHIE SHRIVAASTAVA
Course Name : Introduction to Problem Solving and Programming
Course Code : CSE1021
School Name : School of Computing Science
Engineering and Artificial Intelligence (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	aliquot_sum(n) Function	10/11/20225	
2	are_amicable(a, b) Function	10/11/20225	
3	multiplicative_persistence(n) function	10/11/20225	
4	is_highly_composite(n) Function	10/11/20225	
5	mod_exp(base, exponent, modulus) function	10/11/20225	

TITLE: aliquot_sum(n) Function

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

This Python program calculates the **sum of proper divisors** of a given number. It takes an integer input **n**, iterates through all numbers from 1 to **n-1**, and adds those that evenly divide **n**. Finally, it prints the total sum of these divisors as the result.

RESULTS ACHIEVED:

```
n = int(input("Enter a number: "))
sum_divisors = 0
for i in range(1, n):
    if n % i == 0:
        sum_divisors += i
print("Sum of proper divisors:", sum_divisors)

Enter a number: 12
Sum of proper divisors: 16
```

SKILLS ACHIEVED:

Iteration and conditional accumulation.

Understanding of divisors and proper divisors.

TITLE: are_amicable(a, b) Function

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

METHODOLOGY & TOOL USED: Python programming language

BRIEF DESCRIPTION:

This Python program checks if two numbers are **amicable**. It defines a function `aliquot_sum(n)` that calculates the sum of proper divisors of `n`. Then, it compares whether each number equals the other's divisor sum. If true, it prints "Amicable numbers"; otherwise, it prints "Not amicable."

Result:



```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

sum_a = 0
for i in range(1, a):
    if a % i == 0:
        sum_a += i

sum_b = 0
for i in range(1, b):
    if b % i == 0:
        sum_b += i

if sum_a == b and sum_b == a:
    print("Amicable numbers")
else:
    print("Not amicable numbers")

Enter first number: 13
Enter second number: 15
Not amicable numbers
```

SKILLS ACHIEVED:

Function Reusability: Leveraging existing functions to build new solutions

Logical Condition Construction: Combining multiple conditions with AND logic

Practical No: 3

Date: 11/11/2025

TITLE: multiplicative_persistence(n) function

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

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION:

This Python program finds the **multiplicative persistence** of a number — the number of steps required to reduce it to a single digit by multiplying its digits repeatedly. It loops while the number has more than one digit, multiplies all digits together, updates the number, and counts the steps

RESULTS ACHIEVED:

```
n = int(input("Enter a number: "))
count = 0

while n >= 10:
    product = 1
    for digit in str(n):
        product *= int(digit)
    n = product
    count += 1

print("Multiplicative persistence:", count)

Enter a number: 15
Multiplicative persistence: 1
```

SKILLS ACHIEVED:

Type Conversion: converting between integers and strings for digit manipulation

Iterative Process Control: Managing loops until convergence criteria are met

Practical No: 4

Date: 10/11/2025

TITLE:is_highly_composite(n) Function

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

This Python program checks whether a number is highly composite. It defines a function `count_divisors(x)` to count total divisors of any number. Then, it compares the divisor count of the input number with all smaller numbers. If any smaller number has equal or more divisors, it's not highly composite.

Result:

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

def count_divisors(x):
    c = 0
    for i in range(1, x + 1):
        if x % i == 0:
            c += 1
    return c

max_divisors = 0
is_highly_composite = True

for i in range(1, n):
    if count_divisors(i) >= count_divisors(n):
        is_highly_composite = False
        break

if is_highly_composite:
    print("Highly composite number")
else:
    print("Not highly composite number")
```

```
Enter a number: 17
Not highly composite number
```

SKILLS ACHIEVED:

Algorithm Optimization: Using square root method for efficient divisor counting.

Comparative Analysis: Evaluating a number against all smaller numbers.

Practical No: 5

Date: 10/11/2025

TITLE:mod_exp(base, exponent,modulus) function

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

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

This Python program performs **modular exponentiation**. It takes base, exponent, and modulus as inputs, then repeatedly multiplies the base and applies the modulus in each

iteration. This keeps intermediate results small and prevents overflow. Finally, it prints $(\text{base}^{\text{exponent}}) \% \text{modulus}$, a key operation in cryptography and modular arithmetic..

RESULT:

```
base = int(input("Enter base: "))
exponent = int(input("Enter exponent: "))
modulus = int(input("Enter modulus: "))

result = 1
for i in range(exponent):
    result = (result * base) % modulus

print("Result:",result)

Enter base: 17
Enter exponent: 19
Enter modulus: 11
Result: 2
```

SKILLS ACHIEVED:

Binary Exponentiation: Implementing efficient power computation using bit manipulation.

Modular Arithmetic: Applying mathematical properties to handle large numbers efficiently.



Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10134
Name of Student : AARCHIE SHRIVAASTAVA
Course Name : Introduction to Problem Solving and Programming
Course Code : CSE1021
School Name : School of Computing Science
Engineering and Artificial Intelligence (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	Aliquot Sum Calculator	16/11/20225	
2	Amicable Numbers Check	16/11/20225	
3	Multiplicative Persistence Calculator	16/11/20225	
4	Highly Composite Number Check	16/11/20225	
5	Modular Exponentiation	16/11/20225	

TITLE: Aliquot Sum Calculator

AIM/OBJECTIVE(s) :To write a Python function `aliquot_sum(n)` that returns the sum of all proper divisors of `n` (which are all divisors of `n` other than `n` itself).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

The code defines a function `aliquot_sum(n)` that efficiently calculates the sum of proper divisors. It handles the edge case of `n <= 1`. The loop is optimized to only run up to `sqrt(n)`, making it much faster than checking all numbers up to `n/2`.

RESULTS ACHIEVED:



```
cse project > week5 > main2.py > ...
1  import time
2  import tracemalloc
3  import math
4
5  def aliquot_sum(n):
6      if n <= 1:
7          return 0
8
9      total = 1
10     for i in range(2, int(math.sqrt(n)) + 1):
11         if n % i == 0:
12             total += i
13             if i * i != n:
14                 total += n // i
15     return total
16
17 if __name__ == "__main__":
18     test_number = 220
19
20     tracemalloc.start()
21     start_time = time.perf_counter()
22
23     result = aliquot_sum(test_number)
24
25     end_time = time.perf_counter()
26     current_mem, peak_mem = tracemalloc.get_traced_memory()
27     tracemalloc.stop()
28
29     execution_time = end_time - start_time
30
31     print("Calculating aliquot sum of {test_number}.")
32     print("Result (sum of proper divisors): {result}")
33     print("Execution Time: {execution_time:.9f} seconds")
34     print("Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
35
36     test_number_2 = 12
37
38     tracemalloc.start()
39     start_time = time.perf_counter()
40
41     result_2 = aliquot_sum(test_number_2)
42
43     end_time = time.perf_counter()
44     current_mem, peak_mem = tracemalloc.get_traced_memory()
45     tracemalloc.stop()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main2.py"
Calculating aliquot sum of 12.
Result (sum of proper divisors): 16
Execution Time: 0.000028700 seconds
Peak Memory Usage: 0.75 KiB
PS D:\1\codes\python> █
```

SKILLS ACHIEVED:

- Understanding the concept of a modular multiplicative inverse.
- Using Python's `pow()` function for advanced number theory operations.
- Using `try...except` blocks to handle expected mathematical errors (`ValueError`).
- Continued practice in performance measurement.

Practical No: 2**Date: 16/11/2025****TITLE:** Amicable Numbers Check

AIM/OBJECTIVE(s) : To write a Python function `are_amicable(a, b)` that checks if two numbers `a` and `b` are amicable. Two numbers are amicable if the sum of the proper divisors of `a` is equal to `b`, and the sum of the proper divisors of `b` is equal to `a`.

METHODOLOGY & TOOL USED: Python programming language

BRIEF DESCRIPTION:

The code defines `are_amicable(a, b)` and re-uses the `aliquot_sum(n)` helper function. The `are_amicable` function directly implements the mathematical definition by calling the helper function twice and comparing the results.

Result:



```
1 import time
2 import tracemalloc
3 from functools import reduce
4
5 def mod_inverse(a, m):
6     try:
7         return pow(a, -1, m)
8     except ValueError:
9         return None
10
11 def crt(remainders, moduli):
12     if len(remainders) != len(moduli):
13         return None
14
15     M = reduce(lambda a, b: a * b, moduli)
16
17     total = 0
18     for r_i, m_i in zip(remainders, moduli):
19         M_i = M // m_i
20         y_i = mod_inverse(M_i, m_i)
21
22         if y_i is None:
23             return None
24
25         total += r_i * M_i * y_i
26
27     return total % M
28
29 if __name__ == "__main__":
30
31     remainders = [2, 3, 2]
32     moduli = [3, 5, 7]
33
34     tracemalloc.start()
35     start_time = time.perf_counter()
36
37     result = crt(remainders, moduli)
38
39     end_time = time.perf_counter()
40     current_mem, peak_mem = tracemalloc.get_traced_memory()
41     tracemalloc.stop()
42
43     execution_time = end_time - start_time
44
45     print(f"Solving system of congruences:")
46     for r, m in zip(remainders, moduli):
47         print(f"x = {r} mod {m}")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\WEEK6\main2.py"

Result (x): 23
Execution Time: 0.000838100 seconds
Peak Memory Usage: 0.20 KiB
PS D:\1\codes\python>

```

SKILLS ACHIEVED :

- Re-using helper functions to build more complex logic.
- Implementing a mathematical definition involving multiple conditions.
- Testing function with both positive and negative cases.

Practical No: 3

Date: 16/11/2025

TITLE : Multiplicative Persistence Calculator

AIM/OBJECTIVE(s) : To write a Python function

`multiplicative_persistence(n)` that counts how many steps are needed to multiply a number's digits until a single-digit number is reached.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION : The code defines `multiplicative_persistence(n)`. It uses a `while` loop to repeatedly process the number. In each step, it converts the number to a list of its digits, calculates their product (using `reduce`), and updates the number to this product, incrementing a step counter. The process stops when the number is less than 10.

RESULTS ACHIEVED:



```
cse project > week5 > main3.py > ...
1 import time
2 import tracemalloc
3 from functools import reduce
4
5 def multiplicative_persistence(n):
6     if n < 0:
7         n = abs(n)
8
9     count = 0
10    while n >= 10:
11        digits = [int(d) for d in str(n)]
12        n = reduce(lambda x, y: x * y, digits)
13        count += 1
14    return count
15
16 if __name__ == "__main__":
17
18     test_number = 77
19
20     tracemalloc.start()
21     start_time = time.perf_counter()
22
23     result = multiplicative_persistence(test_number)
24
25     end_time = time.perf_counter()
26     current_mem, peak_mem = tracemalloc.get_traced_memory()
27     tracemalloc.stop()
28
29     execution_time = end_time - start_time
30
31     print(f"Calculating multiplicative persistence of {test_number}.")
32     print(f"Result (steps): {result}")
33     print(f"Execution Time: {execution_time:.9f} seconds")
34     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
35
36
37     test_number_2 = 123
38
39     tracemalloc.start()
40     start_time = time.perf_counter()
41
42     result_2 = multiplicative_persistence(test_number_2)
43
44     end_time = time.perf_counter()
45     current_mem, peak_mem = tracemalloc.get_traced_memory()
46     tracemalloc.stop()
47
48
49 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
50
51 PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main3.py"
52 Calculating multiplicative persistence of 123.
53 Result (steps): 1
54 Execution Time: 0.000027000 seconds
55 Peak Memory Usage: 0.75 KiB
56 PS D:\1\codes\python> 
```

SKILLS ACHIEVED:

- Type conversion (int to string, string to int list).
- Using `while` loops for a process that repeats an unknown number of times.
- Using `functools.reduce` for a cumulative operation.
- String and list manipulation.

Practical No: 4

Date: 16/11/2025

TITLE : Highly Composite Number Check

AIM/OBJECTIVE(s) : To write a Python function `is_highly_composite(n)` that checks if an integer `n` is a highly composite number (HCN). An HCN is a positive integer that has more divisors than any smaller positive integer.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

1. The code defines `is_highly_composite(n)` and a helper `get_divisor_count(n)`. The main function implements the definition of an HCN by finding the divisor count of `n` and comparing it to the divisor count of all positive integers less than `n`.

Result:

```
cse project > week5 > main4.py > ...
1  import time
2  import tracemalloc
3  import math
4
5  def get_divisor_count(n):
6      if n == 0:
7          return 0
8
9      count = 0
10     for i in range(1, int(math.sqrt(n)) + 1):
11         if n % i == 0:
12             if i * i == n:
13                 count += 1
14             else:
15                 count += 2
16     return count
17
18 def is_highly_composite(n):
19     if n <= 1:
20         return False
21
22     divisor_count_n = get_divisor_count(n)
23
24     for i in range(1, n):
25         if get_divisor_count(i) >= divisor_count_n:
26             return False
27
28     return True
29
30 if __name__ == "__main__":
31     test_number = 12
32
33     tracemalloc.start()
34     start_time = time.perf_counter()
35
36     result = is_highly_composite(test_number)
37
38     end_time = time.perf_counter()
39     current_mem, peak_mem = tracemalloc.get_traced_memory()
40     tracemalloc.stop()
41
42     execution_time = end_time - start_time
43
44     print("Checking if {test_number} is a highly composite number.")
45     print("Result: {result}")
46     print("Execution Time: {execution_time:.9f} seconds")
47
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week5\main4.py"
Checking if 10 is a highly composite number.
Result: False
Execution Time: 0.000055400 seconds
Peak Memory Usage: 0.11 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED :

- Implementation of a complex number theory definition.
- Creating and using efficient helper functions.
- Writing optimized loops for comparative analysis.

Practical No: 5**Date: 16/11/2025**

TITLE : Modular Exponentiation

AIM/OBJECTIVE(s) : To write a Python function `mod_exp(base, exponent, modulus)` that efficiently calculates $(base^{exponent}) \% modulus$.

METHODOLOGY & TOOL USED: Python programming language

BRIEF DESCRIPTION:

1. The code defines `mod_exp` which directly uses `pow(base, exponent, modulus)` to get the result. This is the standard and most efficient way to perform this operation in Python. An edge case for `modulus == 1` (where the result is always 0) is handled.

RESULT:

```
cse project > week5 > main5.py > ...
1  import time
2  import tracemalloc
3
4  def mod_exp(base, exponent, modulus):
5      if modulus == 1:
6          return 0
7      return pow(base, exponent, modulus)
8
9  if __name__ == "__main__":
10
11      base = 3
12      exponent = 4
13      modulus = 7
14
15      tracemalloc.start()
16      start_time = time.perf_counter()
17
18      result = mod_exp(base, exponent, modulus)
19
20      end_time = time.perf_counter()
21      current_mem, peak_mem = tracemalloc.get_traced_memory()
22      tracemalloc.stop()
23
24      execution_time = end_time - start_time
25
26      print(f"Calculating ({base}^{exponent}) % {modulus}")
27      print(f"Result: {result}")
28      print(f"Execution Time: {execution_time:.9f} seconds")
29      print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
30
31
32      base_2 = 5
33      exponent_2 = 123456
34      modulus_2 = 13
35
36      tracemalloc.start()
37      start_time = time.perf_counter()
38
39      result_2 = mod_exp(base_2, exponent_2, modulus_2)
40
41      end_time = time.perf_counter()
42      current_mem, peak_mem = tracemalloc.get_traced_memory()
43      tracemalloc.stop()
44
45      execution_time_2 = end_time - start_time
46
47      print(f"\nCalculating ({base_2}^{exponent_2}) % {modulus_2}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Calculating (5^123456) % 13
Result: 1
Execution Time: 0.000005100 seconds
Peak Memory Usage: 0.00 KiB
```

SKILLS ACHIEVED:

- Understanding the concept and utility of modular exponentiation.
- Knowing and using the correct built-in function (`pow(b, e, m)`) for an optimized mathematical operation.
- Appreciating the difference between a naive calculation and an efficient algorithm.

Lab Manual

Practical and Skills Development



CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10134
Name of Student : AARCHIE SHRIVAASTAVA
Course Name : Introduction to Problem Solving and Programming
Course Code : CSE1021
School Name : School of Computing Science
Engineering and Artificial Intelligence (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	Lucas Numbers	16/11/20225	
2	Chinese Remainder Theorem (CRT) Solver	16/11/20225	
3	Quadratic Residue Check	16/11/20225	
4	Order of an Element Modulo n	16/11/20225	
5	Fibonacci Prime Check	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Write a function Lucas Numbers

Generator `lucas_sequence(n)` that generates the first `n` Lucas numbers

(similar to Fibonacci but starts with 2,

1).

AIM/OBJECTIVE(s): To write a Python function `mod_inverse(a, m)` that finds the integer x such that $(a \cdot x) \equiv 1 \pmod{m}$. The script should also measure the execution time and peak memory usage.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

`mod_inverse(a, m)` that calculates the modular multiplicative inverse.

It returns `None` if the modulus `m` is invalid ($m \leq 1$) or if the inverse does not exist (caught by `ValueError`).

RESULTS ACHIEVED:

```
cse project > week2 > main.py > ...
1  import time
2  import tracemalloc
3
4  def lucas_sequence(n):
5      a, b = 2, 1
6      count = 0
7      while count < n:
8          if count == 0:
9              yield a
10             elif count == 1:
11                 yield b
12             else:
13                 a, b = b, a + b
14                 yield b
15             count += 1
16
17 if __name__ == "__main__":
18     n_numbers = 30
19
20     tracemalloc.start()
21     start_time = time.perf_counter()
22
23     numbers = list(lucas_sequence(n_numbers))
24
25     end_time = time.perf_counter()
26     current_mem, peak_mem = tracemalloc.get_traced_memory()
27     tracemalloc.stop()
28
29     execution_time = end_time - start_time
30
31     print(f"Generated {n_numbers} Lucas numbers.")
32     print(f"Numbers: {numbers}")
33     print(f"Execution Time: {execution_time:.9f} seconds")
34     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main.py"
Generated 30 Lucas numbers.
Numbers: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349, 15127, 24476, 39603, 64079, 103682, 167761, 271443, 439204, 710647, 1149851]
Execution Time: 0.001339600 seconds
Peak Memory Usage: 1.08 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding the concept of a modular multiplicative inverse.
- Using Python's `pow()` function for advanced number theory operations.
- Using `try...except` blocks to handle expected mathematical errors (`ValueError`).
- Continued practice in performance measurement.

Practical No: 2

Date: 16/11/2025

TITLE: Chinese Remainder Theorem (CRT) Solver

AIM/OBJECTIVE(s) : Chinese Remainder Theorem (CRT) Solver

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines a `crt` function that solves a system of congruences using the standard constructive algorithm for the Chinese Remainder Theorem. It relies on the `mod_inverse` helper function.

Result:

```
1 import time
2 import tracemalloc
3 from functools import reduce
4
5 def mod_inverse(a, m):
6     try:
7         return pow(a, -1, m)
8     except ValueError:
9         return None
10
11 def crt(remainders, moduli):
12     if len(remainders) != len(moduli):
13         return None
14
15     M = reduce(lambda a, b: a * b, moduli)
16
17     total = 0
18     for r_i, m_i in zip(remainders, moduli):
19         M_i = M // m_i
20         y_i = mod_inverse(M_i, m_i)
21
22         if y_i is None:
23             return None
24
25         total += r_i * M_i * y_i
26
27     return total % M
28
29 if __name__ == "__main__":
30
31     remainders = [2, 3, 2]
32     moduli = [3, 5, 7]
33
34     tracemalloc.start()
35     start_time = time.perf_counter()
36
37     result = crt(remainders, moduli)
38
39     end_time = time.perf_counter()
40     current_mem, peak_mem = tracemalloc.get_traced_memory()
41     tracemalloc.stop()
42
43     execution_time = end_time - start_time
44
45     print(f"Solving system of congruences:")
46     for r, m in zip(remainders, moduli):
47         print(f"x = {r} mod {m}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\WEEK6\main2.py"
```

```
Result (x): 23
```

```
Execution Time: 0.000838100 seconds
```

```
Peak Memory Usage: 0.20 KiB
```

```
PS D:\1\codes\python> █
```

SKILLS ACHIEVED :

- Understanding and implementing the Chinese Remainder Theorem algorithm.
- Combining multiple functions (`crt` and `mod_inverse`) to solve a complex problem.
- Using `zip` to iterate over multiple lists simultaneously.

Practical No: 3

Date: 16/11/2025

TITLE : Quadratic Residue Check

AIM/OBJECTIVE(s): To write a Python function

`is_quadratic_residue(a, p)` that checks if the congruence $x^2 \equiv a \pmod{p}$ has a solution, assuming p is a prime number.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION:

The code defines a function `is_quadratic_residue(a, p)` that implements Euler's criterion to determine if `a` is a quadratic residue modulo `p`. It handles the special case of $p=2$ and the trivial case of $a=0$.

RESULTS ACHIEVED:



```
1 import time
2 import tracemalloc
3
4 def is_quadratic_residue(a, p):
5     if p <= 1:
6         return False
7
8     a = a % p
9     if a == 0:
10        return True
11
12    if p == 2:
13        return a == 1
14
15    exponent = (p - 1) // 2
16    result = pow(a, exponent, p)
17
18    return result == 1
19
20 if __name__ == "__main__":
21
22     a_res = 2
23     p_mod = 7
24
25     tracemalloc.start()
26     start_time = time.perf_counter()
27
28     result = is_quadratic_residue(a_res, p_mod)
29
30     end_time = time.perf_counter()
31     current_mem, peak_mem = tracemalloc.get_traced_memory()
32     tracemalloc.stop()
33
34     execution_time = end_time - start_time
35
36     print(f"Checking if {a_res} is a quadratic residue mod {p_mod}.")
37     print(f"Result: {result}")
38     print(f"Execution Time: {execution_time:.9f} seconds")
39     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
40
41
42     a_non_res = 3
43     p_mod_non = 7
44
45     tracemalloc.start()
46     start_time = time.perf_counter()
47
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\WEEK6\main3.py"
Checking if 3 is a quadratic residue mod 7.
Result: False
Execution Time: 0.000006300 seconds
Peak Memory Usage: 0.00 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding the concept of quadratic residues.
- Implementation of Euler's criterion.
- Efficient modular exponentiation using `pow()`.
- Handling edge cases in number theory problems (e.g., $p=2$).

Practical No: 4

Date: 16/11/2025

TITLE:Order of an Element Modulo n

AIM/OBJECTIVE(s) : To write a Python function `order_mod(a, n)` that finds the smallest positive integer k (the order) such that $a^k \equiv 1 \pmod{n}$

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :

Python function `order_mod(a, n)` that finds the smallest positive integer k (the order) such that $a^k \equiv 1 \pmod{n}$

1. Order of 3 (mod 7). ($\phi(7)=6$, divisors are 1, 2, 3, 6).
2. Order of 5 (mod 24). ($\phi(24)=8$, divisors are 1, 2, 4, 8).

Result:

```
cse project > WEEK 7 > main.py > ...
1  import time
2  import tracemalloc
3  import math
4
5  def get_phi(n):
6      result = n
7      p = 2
8      while p * p <= n:
9          if n % p == 0:
10             while n % p == 0:
11                 n //= p
12             result -= result // p
13             p += 1
14         if n > 1:
15             result -= result // n
16         return result
17
18  def get_divisors(n):
19      divs = set()
20      for i in range(1, int(n**0.5) + 1):
21          if n % i == 0:
22              divs.add(i)
23              divs.add(n // i)
24      return sorted(list(divs))
25
26  def order_mod(a, n):
27      if n <= 1 or math.gcd(a, n) != 1:
28          return None
29
30      phi_n = get_phi(n)
31      divisors = get_divisors(phi_n)
32
33      for k in divisors:
34          if pow(a, k, n) == 1:
35              return k
36
37      return None
38
39  if __name__ == "__main__":
40
41      a_val = 3
42      n_val = 7
43
44      tracemalloc.start()
45      start_time = time.perf_counter()
46
47      result = order_mod(a_val, n_val)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Finding the order of 5 mod 24.
Result (k): 2
Execution Time: 0.000043000 seconds
Peak Memory Usage: 0.43 KiB
PS D:\1\codes\python>

SKILLS ACHIEVED :

- Understanding the "order" of an element in a modular group.
- Implementation of Euler's totient function.
- Algorithm to find all divisors of a number.
- Applying number theory (Lagrange's/Euler's theorem) to optimize an algorithm.

Practical No: 5**Date: 16/11/2025****TITLE:**Fibonacci Prime Check

AIM/OBJECTIVE(s):

To write a Python function `is_fibonacci_prime(n)` that checks if a given integer `n` is both a Fibonacci number and a prime number.

METHODOLOGY & TOOL USED: Python programming language

BRIEF DESCRIPTION:

The code defines `is_fibonacci_prime(n)` along with its required helper functions. The `is_fibonacci` check is based on a mathematical property, avoiding the need to generate Fibonacci numbers.

The main execution block (`if __name__ == "__main__":`) measures the time and memory for two cases:

1. `13` (which is both prime and Fibonacci).
2. `21` (which is Fibonacci but not prime).

RESULT:

```
cse project > WEEK6 > main5.py > ...
1  import time
2  import tracemalloc
3
4  def is_prime(n):
5      if n <= 1:
6          return False
7      if n <= 3:
8          return True
9      if n % 2 == 0 or n % 3 == 0:
10         return False
11         i = 5
12         while i * i <= n:
13             if n % i == 0 or n % (i + 2) == 0:
14                 return False
15             i += 6
16         return True
17
18 def is_perfect_square(k):
19     if k < 0:
20         return False
21     s = int(k**0.5)
22     return s * s == k
23
24 def is_fibonacci(n):
25     if n < 0:
26         return False
27
28     test_1 = 5 * n**2 + 4
29     test_2 = 5 * n**2 - 4
30     return is_perfect_square(test_1) or is_perfect_square(test_2)
31
32 def is_fibonacci_prime(n):
33     return is_prime(n) and is_fibonacci(n)
34
35 if __name__ == "__main__":
36     test_number = 13
37
38     tracemalloc.start()
39     start_time = time.perf_counter()
40
41
42     result = is_fibonacci_prime(test_number)
43
44     end_time = time.perf_counter()
45     current_mem, peak_mem = tracemalloc.get_traced_memory()
46     tracemalloc.stop()
47
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Checking if 21 is a Fibonacci Prime.
Result: False
Execution Time: 0.000004800 seconds
Peak Memory Usage: 0.00 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Understanding and implementing mathematical properties of number sequences.
- Writing helper functions to build up complex logic (`is_perfect_square`, `is_fibonacci`, `is_prime`).
- Combining logical conditions (`and`) to satisfy multiple criteria.



Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No : 25BSA10134
Name of Student : AARCHIE SHRIVAASTAVA
Course Name : Introduction to Problem Solving and Programming
Course Code : CSE1021
School Name : School of Computing Science
Engineering and Artificial Intelligence (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	aliquot_sum(n) Function	16/11/20225	
2	are_amicable(a, b) Function	16/11/20225	
3	multiplicative_persistence(n) function	16/11/20225	
4	is_highly_composite(n) Function	16/11/20225	
5	mod_exp(base, exponent, modulus) function	16/11/20225	

Practical No:



Date:

1

10/11/2025

TITLE: Write a function Lucas Numbers

Generator `lucas_sequence(n)` that generates the first `n` Lucas numbers

(similar to Fibonacci but starts with 2,

1).

AIM/OBJECTIVE(s): `lucas_sequence`

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

A generator function (`lucas_sequence`) is implemented to yield Lucas numbers iteratively, which is memory-efficient. The main part of the script calls this generator, collects all generated numbers into a list, and measures the performance of this operation.

RESULTS ACHIEVED:

```
cse project > week2 > main.py > ...
1  import time
2  import tracemalloc
3
4  def lucas_sequence(n):
5      a, b = 2, 1
6      count = 0
7      while count < n:
8          if count == 0:
9              yield a
10             elif count == 1:
11                 yield b
12             else:
13                 a, b = b, a + b
14                 yield b
15             count += 1
16
17 if __name__ == "__main__":
18     n_numbers = 30
19
20     tracemalloc.start()
21     start_time = time.perf_counter()
22
23     numbers = list(lucas_sequence(n_numbers))
24
25     end_time = time.perf_counter()
26     current_mem, peak_mem = tracemalloc.get_traced_memory()
27     tracemalloc.stop()
28
29     execution_time = end_time - start_time
30
31     print(f"Generated {n_numbers} Lucas numbers.")
32     print(f"Numbers: {numbers}")
33     print(f"Execution Time: {execution_time:.9f} seconds")
34     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main.py"
Generated 30 Lucas numbers.
Numbers: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349, 15127, 24476, 39603, 64079, 103682, 167761, 271443, 439204, 710647, 1149851]
Execution Time: 0.001339600 seconds
Peak Memory Usage: 1.08 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

- Python programming fundamentals.
- Implementation of generator functions (using **yield**) for efficient data generation.
- Performance analysis:
 - Measuring code execution time using the **time** module.
 - Measuring peak memory usage using the **tracemalloc** module.
- Understanding and using the **if __name__ == "__main__":** guard.
- Using formatted strings (f-strings) for clear and dynamic output.

TITLE: Perfect Power Check with Performance Measurement

AIM/OBJECTIVE(s): To write a Python function `is_perfect_power(n)` that determines if a given integer `n` can be expressed as a^b , where $a > 0$ and $b > 1$. The script should also measure the execution time and peak memory usage of this check.

METHODOLOGY & TOOL USED: Python programming language

BRIEF DESCRIPTION:

`is_perfect_power(n)`. It first handles edge cases where $n \leq 3$. It then iterates through potential bases `a` from 2 up to `int(math.sqrt(n)) + 1`. For each `a`, it enters a nested loop for exponent `b` (starting at 2). It calculates a^b in each step. If $a^b == n$, it returns `True`. If $a^b > n$, it breaks the inner loop and tries the next base `a`. If no such pair `(a, b)` is found after all checks, it returns `False`.

The main execution block measures the time and memory for checking two numbers (one perfect power like 125, and one non-perfect power like 126) and prints the results.

Result:

```
cse project > week2 > main2.py > ...
1 import time
2 import tracemalloc
3 import math
4
5 def is_perfect_power(n):
6     if n <= 2:
7         return False
8
9     max_base = int(math.sqrt(n)) + 1
10
11     for a in range(2, max_base):
12         b = 2
13         while True:
14             try:
15                 result = a ** b
16             except OverflowError:
17                 break
18
19             if result == n:
20                 return True
21             elif result > n:
22                 break
23             b += 1
24     return False
25
26 if __name__ == "__main__":
27     test_number = 126
28
29     tracemalloc.start()
30     start_time = time.perf_counter()
31
32     result = is_perfect_power(test_number)
33
34     end_time = time.perf_counter()
35     current_mem, peak_mem = tracemalloc.get_traced_memory()
36     tracemalloc.stop()
37
38     execution_time = end_time - start_time
39
40     print(f"Checking if {test_number} is a perfect power.")
41     print(f"Result: {result}")
42     print(f"Execution Time: {execution_time:.9f} seconds")
43     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
44
45     test_number_false = 125
46     tracemalloc.start()
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

Code + - [] [X] ...

```
Checking if 126 is a perfect power.
Result: False
Execution Time: 0.000060400 seconds
Peak Memory Usage: 0.75 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED:

The script successfully checks if the test numbers are perfect powers and reports the performance.

TITLE:Write a function Collatz Sequence

Practical No:



Date:

Length `collatz_length(n)` that returns the number of steps for `n` to reach 1 in the Collatz conjecture.

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

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION:

The code defines a function `collatz_length(n)` that calculates the "stopping time" for the Collatz sequence starting at `n`

RESULTS ACHIEVED:



```
1 import time
2 import tracemalloc
3
4 def collatz_length(n):
5     if n <= 0:
6         return 0
7
8     length = 0
9     while n != 1:
10         if n % 2 == 0:
11             n = n // 2
12         else:
13             n = 3 * n + 1
14         length += 1
15     return length
16
17 if __name__ == "__main__":
18     test_number = 27
19
20     tracemalloc.start()
21     start_time = time.perf_counter()
22
23     result_length = collatz_length(test_number)
24
25     end_time = time.perf_counter()
26     current_mem, peak_mem = tracemalloc.get_traced_memory()
27     tracemalloc.stop()
28
29     execution_time = end_time - start_time
30
31     print(f"Calculating Collatz sequence length for {test_number}.")
32     print(f"Result (steps to reach 1): {result_length}")
33     print(f"Execution Time: {execution_time:.9f} seconds")
34     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
35
36     test_number_long = 837799
37
38     tracemalloc.start()
39     start_time = time.perf_counter()
40
41     result_length_long = collatz_length(test_number_long)
42
43     end_time = time.perf_counter()
44     current_mem, peak_mem = tracemalloc.get_traced_memory()
45     tracemalloc.stop()
46
47     execution_time_long = end_time - start_time
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Calculating Collatz sequence length for 837799.
Result (steps to reach 1): 524
Execution Time: 0.000765600 seconds
Peak Memory Usage: 0.12 KiB
PS D:\1\codes\python>

SKILLS ACHIEVED:

Implementation of an iterative algorithm based on a mathematical conjecture.

Use of conditional logic (**if/else**) and loop structures (**while**).

Integer arithmetic (using **%** for modulus and **//** for integer division).

Continued practice in performance analysis with **time** and **tracemalloc**.

Practical No: 4**Date: 16/11/2025**

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

AIM/OBJECTIVE(s) : To write a Python function `polygonal_number(s, n)` that calculates and returns the n -th s -gonal (polygonal) number, given the number of sides s and the term n . The script should also measure the execution time and peak memory usage.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION :


The code defines a function `polygonal_number(s, n)` that implements the mathematical formula for the n -th s -gonal number. It includes a check to ensure s (sides) is at least 3 and n (term) is at least 1.

The main execution block (`if __name__ == "__main__":`) measures the time and memory for calculating two different polygonal numbers:

1. The 10th pentagonal (5-gonal) number.
2. The 12th triangular (3-gonal) number.

It prints the calculated number, the execution time, and the peak memory usage for each case.

Result:


```
cse project > week2 >  main4.py > ...
1  import time
2  import tracemalloc
3
4  def polygonal_number(s, n):
5      if s < 3 or n < 1:
6          return None
7
8      numerator = ((s - 2) * n**2) - ((s - 4) * n)
9      result = numerator // 2
10     return result
11
12 if __name__ == "__main__":
13
14     s_sides = 5
15     n_term = 10
16
17     tracemalloc.start()
18     start_time = time.perf_counter()
19
20     result_number = polygonal_number(s_sides, n_term)
21
22     end_time = time.perf_counter()
23     current_mem, peak_mem = tracemalloc.get_traced_memory()
24     tracemalloc.stop()
25
26     execution_time = end_time - start_time
27
28     print(f"Calculating the {n_term}-th {s_sides}-gonal number.")
29     print(f"Result: {result_number}")
30     print(f"Execution Time: {execution_time:.9f} seconds")
31     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
32
33
34     s_sides_tri = 3
35     n_term_tri = 12
36
37     tracemalloc.start()
38     start_time = time.perf_counter()
39
40     result_tri = polygonal_number(s_sides_tri, n_term_tri)
41
42     end_time = time.perf_counter()
43     current_mem, peak_mem = tracemalloc.get_traced_memory()
44     tracemalloc.stop()
45
46     execution_time_tri = end_time - start_time
47
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\mai
Calculating the 12-th 3-gonal (Triangular) number.
Result: 78
Execution Time: 0.000009600 seconds
Peak Memory Usage: 0.03 KiB
PS D:\1\codes\python>
```

SKILLS ACHIEVED :

- Translation of a mathematical formula into a working Python function.
- Handling function arguments and basic input validation (e.g., `$s \ge 3$`).
- Continued practice in performance measurement with `time` and `tracemalloc`.
- Testing the function with known values (e.g., triangular, pentagonal numbers) to verify correctness

Practical No: 5

Date: 16/11/2025

TITLE:Write a function Carmichael Number

Check `is_carmichael(n)` that checks if a composite number n satisfies $a^{n-1} \equiv 1 \pmod n$ for all a coprime to n .

AIM/OBJECTIVE(s):

To write a Python function `is_carmichael(n)` that checks if a given composite number n is a Carmichael number. A number is a Carmichael number if it is composite and satisfies $a^{n-1} \equiv 1 \pmod n$ for all integers a that are coprime to n .

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

The code defines two functions. `is_prime(n)` is a standard trial division primality test. `is_carmichael(n)` implements the definition of a

Carmichael number. It first rules out prime numbers. Then, it iterates through all possible bases a from 2 to $n-1$, finds those coprime to n , and tests if $a^{n-1} \pmod n$ is 1.

RESULT:

```
cse project > week2 > main5.py > ...
1  import time
2  import tracemalloc
3  import math
4
5  def is_prime(n):
6      if n <= 1:
7          return False
8      if n <= 3:
9          return True
10     if n % 2 == 0 or n % 3 == 0:
11         return False
12     i = 5
13     while i * i <= n:
14         if n % i == 0 or n % (i + 2) == 0:
15             return False
16         i += 6
17     return True
18
19 def is_carmichael(n):
20     if n <= 1 or not is_prime(n):
21         return False
22
23     for a in range(2, n):
24         if math.gcd(a, n) == 1:
25             if pow(a, n - 1, n) != 1:
26                 return False
27     return True
28
29 if __name__ == "__main__":
30
31     test_number = 561
32
33     tracemalloc.start()
34     start_time = time.perf_counter()
35
36     result = is_carmichael(test_number)
37
38     end_time = time.perf_counter()
39     current_mem, peak_mem = tracemalloc.get_traced_memory()
40     tracemalloc.stop()
41
42     execution_time = end_time - start_time
43
44     print(f"Checking if {test_number} is a Carmichael number.")
45     print(f"Result: {result}")
46     print(f"Execution Time: {execution_time:.9f} seconds")
47     print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
48
49 PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
50 Code + - [] [] ...
51
52 PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main5.py"
53 Checking if 561 is a Carmichael number.
54 Result: False
55 Execution Time: 0.000025300 seconds
56 Peak Memory Usage: 0.03 KiB
57 PS D:\1\codes\python> 
```

SKILLS ACHIEVED:

Implementation of number theory definitions (Primality, Coprimality, Fermat's Little Theorem).

- Use of helper functions (`is_prime`) to structure complex logic.
- Use of `math.gcd` for coprimality testing.
- Use of `pow(a, b, m)` for efficient modular exponentiation.
- Understanding the computational cost of different algorithms.

CSE Assignment Report

Course: Introduction to Problem Solving and Programming

Course Code: CSE1021

Submitted By: * Name: AARCHIE SHRIVASTAVA

Roll No.: 25BSA10134

Submitted To: Dr. Hemraj Shobharam Lamkuche

Date of Submission: 16/11/2025

Table of Contents

1. Program 1: Implement the probabilistic Miller-Rabin test `is_prime_miller_rabin(n, k)` with k rounds.
2. Program 2: Implement `pollard_rho(n)` for integer factorization using Pollard's rho algorithm.
3. Program 3: Write a function `zeta_approx(s, terms)` that approximates the Riemann zeta function $\zeta(s)$ using the first 'terms' of the series.
4. Program 4: 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.

• Program 1: is_prime_miller_rabin(n, k) with k rounds

.1 Program Assigned

Implement the probabilistic Miller-Rabin test `is_prime_miller_rabin(n, k)` with k rounds.

.2 Code

```

n = int(input("Enter number to test primality: "))
k = int(input("Enter number of rounds: "))

if n < 2:
    print("Composite")
    exit()
if n in (2, 3):
    print("Prime")
    exit()
if n % 2 == 0:
    print("Composite")
    exit()

d = n - 1
r = 0
while d % 2 == 0:
    d //= 2
    r += 1

bases = [2, 3, 5, 7, 11, 13, 17]
is_probably_prime = True
base_index = 0

for _ in range(k):
    a = bases[base_index]
    base_index += 1
    if a >= n - 1:
        break

    x = pow(a, d, n)
    if x == 1 or x == n - 1:
        continue

    composite_flag = True
    for _ in range(r - 1):
        x = (x * x) % n
        if x == n - 1:
            composite_flag = False
            break

    if composite_flag:
        is_probably_prime = False
        break

if is_probably_prime:
    print("Probably Prime")
else:
    print("Composite")

```

```

Enter number to test primality: 71
Enter number of rounds: 4
Probably Prime
>>>

```

○ Skills Acquired

- Understanding, implementing, and utilizing **probabilistic algorithms**.
- Applying complex number theory theorems to computational problems.

• Program 2: pollard_rho(n) Function

.1 Program Assigned

Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.

.2 Code

```
n = int(input("Enter number to factor: "))

if n % 2 == 0:
    print("Factor:", 2)
    exit()

x = 2
y = 2
c = 1
d = 1

while d == 1:
    x = (x * x + c) % n
    y = (y * y + c) % n
    y = (y * y + c) % n

    diff = x - y
    if diff < 0:
        diff = -diff

    a = diff
    b = n
    while b != 0:
        a, b = b, a % b

    d = a

if d == n:
    print("Failure factorizing")
else:
    print("Non-trivial factor:", d)

Enter number to factor: 100
Factor: 2

>>> |
```

2.5 Skills Acquired

- Implementing a **probabilistic (Monte Carlo)** and **non-deterministic** algorithm.
- Applying graph theory algorithms (specifically, Floyd's Cycle-Finding Algorithm, or "Tortoise and Hare") to number theory problems.

- **Program 3: zeta_approx(s, terms) Function**

Program Assigned

Write a function `zeta_approx(s, terms)` that approximates the Riemann zeta function $\zeta(s)$ using the first 'terms' of the series.

Code

```
s = float(input("Enter s: "))
terms = int(input("Enter number of terms: "))

zeta = 0.0
n = 1

while n <= terms:
    zeta += 1 / (n ** s)
    n += 1

print("Zeta approximation =", zeta)
```

```
>>> Enter s: 3.14
Enter number of terms: 3
Zeta approximation = 1.1451968791422344
```

Skills Acquired

- Confidently handling and performing calculations involving complex numbers.
- Implementing numerical methods for approximating infinite mathematical concepts.
- Implementing dynamic exponentiation where both the base and the exponent are variable (and the exponent may be complex).

4.1 Program 4: partition_function(n) Function

4.1 program assigned:

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.

4.2 Code

```

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

# dp[i] = number of partitions of i
dp = [0] * (n + 1)
dp[0] = 1

num = 1
while num <= n:
    i = num
    while i <= n:
        dp[i] += dp[i - num]
        i += 1
    num += 1

print("Number of partitions p(n):", dp[n])

```

```

Enter n: 10
Number of partitions p(n): 42
>>>

```

4.3 Skills acquired

- Understanding and applying the Dynamic Programming paradigm to solve combinatorial problems.
- Translating a complex mathematical recurrence relation into nested loop structures.

5. Overall Skills Acquired & Conclusion

- Algorithmic Efficiency and Complexity Management
- Applied Number Theory and Modular Arithmetic
- Numerical & Complex Analysis