Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	:25MIP10141
Name of Student	:SATVIK SRIVASTAVA
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:

VIT BHOPAL

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: 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): `lucas_sequence`

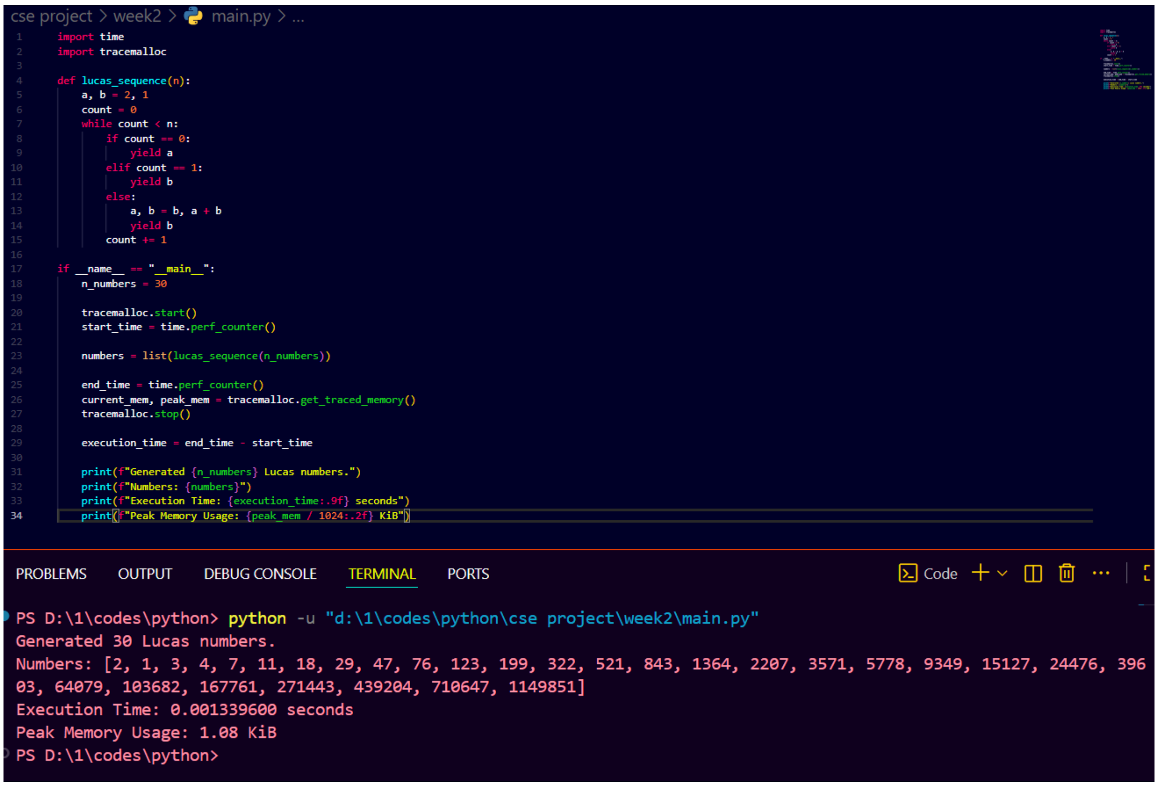
METHODOLOGY & TOOL USED:

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:



```python
import time
import tracemalloc

def lucas_sequence(n):
    a, b = 2, 1
    count = 0
    while count < n:
        if count == 0:
            yield a
        elif count == 1:
            yield b
        else:
            a, b = b, a + b
            yield b
        count += 1

if __name__ == "__main__":
    n_numbers = 30

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

    numbers = list(lucas_sequence(n_numbers))

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Generated {n_numbers} Lucas numbers.")
    print(f"Numbers: {numbers}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```

```
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, 396
03, 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.

Practical No: 2

Date: 16/11/2025

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.

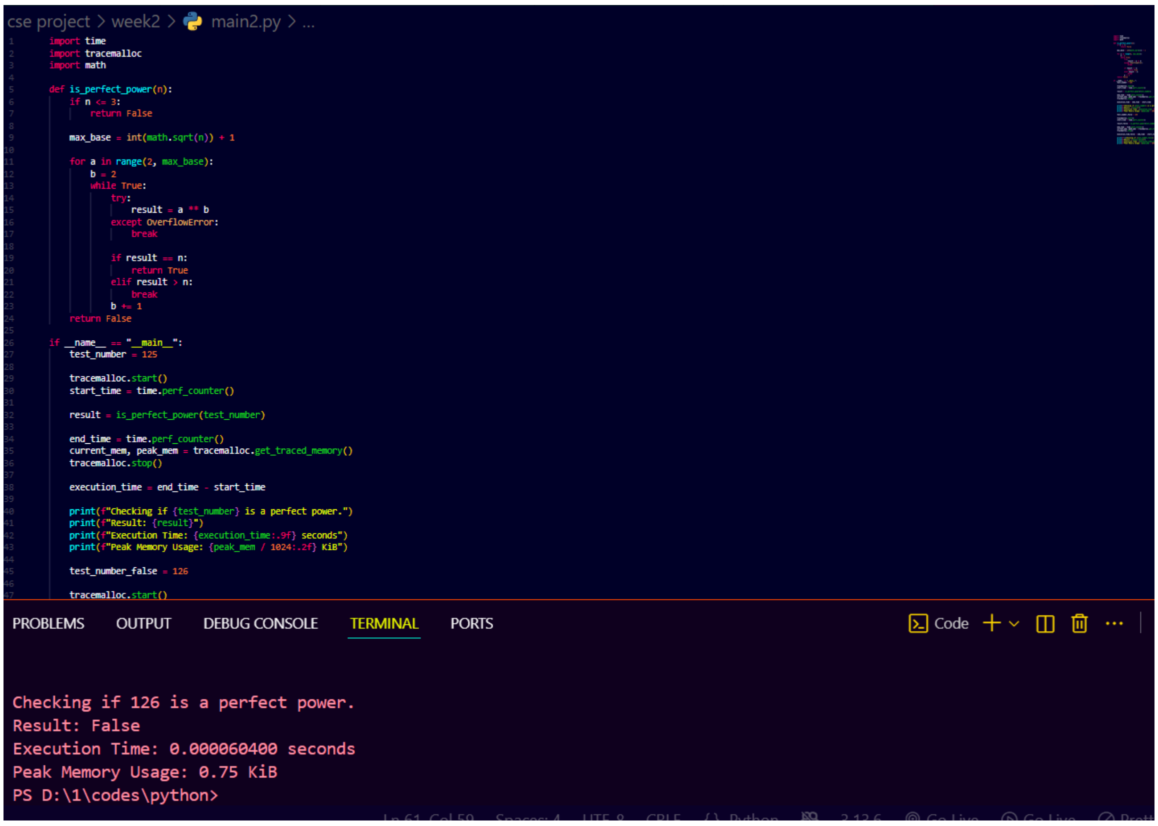
METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION:

`is_perfect_power(n)`. It first handles edge cases where $n \le 3$. 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:



```python
import time
import tracemalloc
import math

def is_perfect_power(n):
    if n <= 3:
        return False

    max_base = int(math.sqrt(n)) + 1

    for a in range(2, max_base):
        b = 2
        while True:
            try:
                result = a ** b
            except OverflowError:
                break

            if result == n:
                return True
            elif result > n:
                break
            b += 1
    return False

if __name__ == "__main__":
    test_number = 125

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

    result = is_perfect_power(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Checking if {test_number} is a perfect power.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

    test_number_false = 126

    tracemalloc.start()
```

```
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.

Practical No: 3

Date: 16/11/2025

TITLE:Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps

for n to reach 1 in the Collatz conjecture.

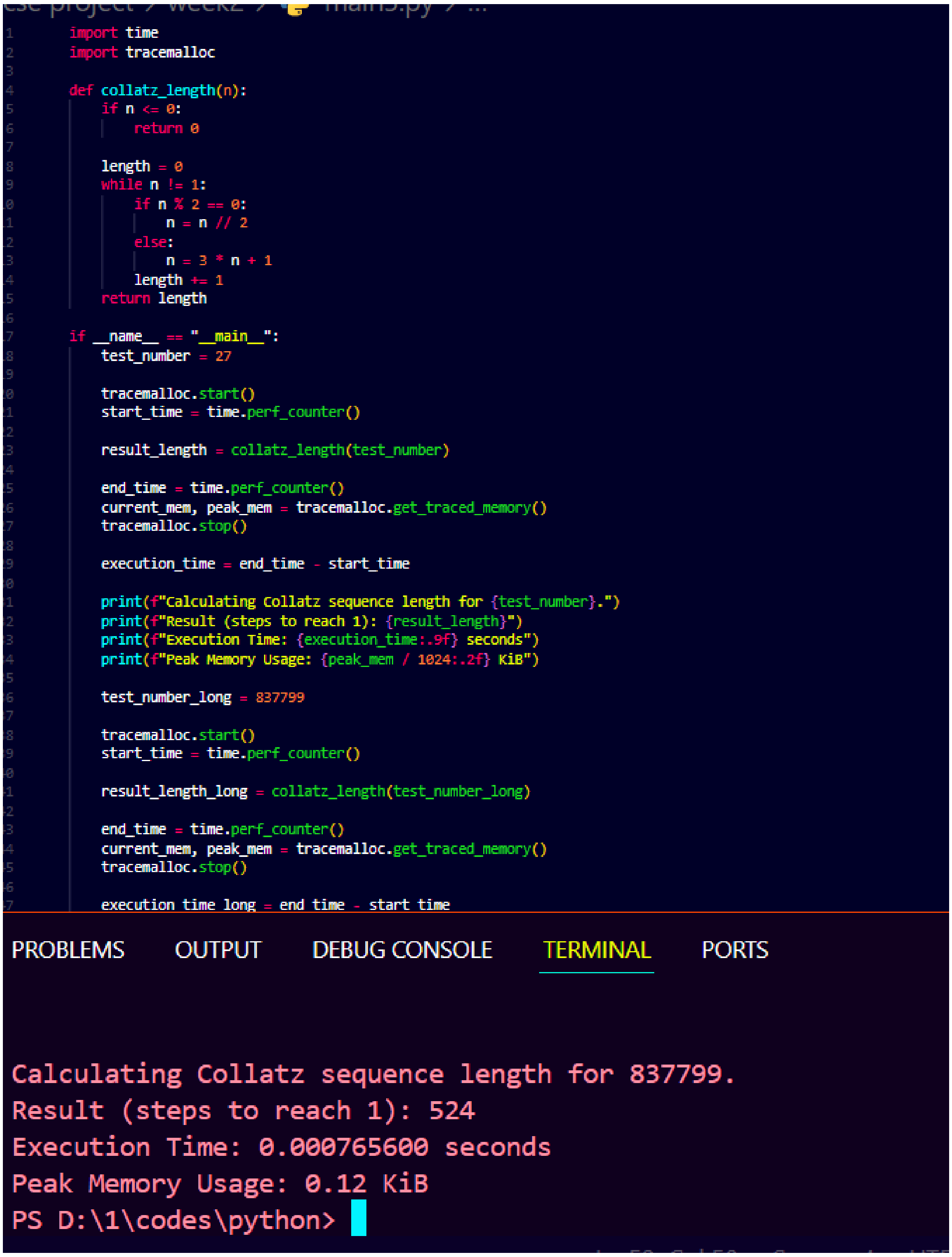
AIM/OBJECTIVE(s):To write a Python function `collatz_length(n)` that returns the number of steps required for a given integer `n` to reach 1 by following the rules of the Collatz conjecture .

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:



```python
import time
import tracemalloc

def collatz_length(n):
    if n <= 0:
        return 0

    length = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        length += 1
    return length

if __name__ == "__main__":
    test_number = 27

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

    result_length = collatz_length(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Calculating Collatz sequence length for {test_number}.")
    print(f"Result (steps to reach 1): {result_length}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")

    test_number_long = 837799

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

    result_length_long = collatz_length(test_number_long)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time_long = end_time - start_time
```

```
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 :

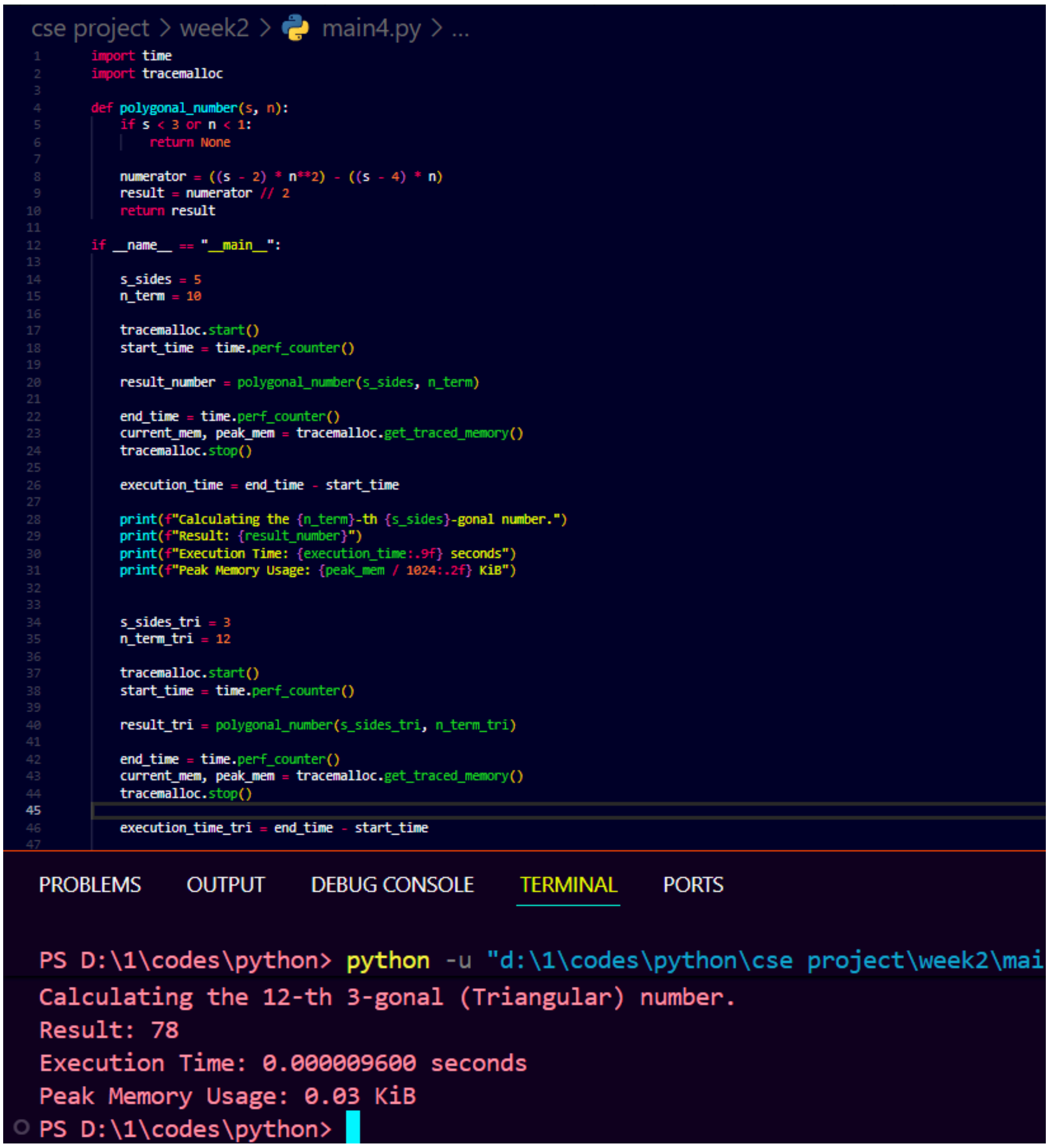
The code defines a function `polygonal_number(s, n)` that implements the mathematical formula for the n-th s-gonal number. It includes a check to ensure `s` (sides) is at least 3 and `n` (term) is at least 1.

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:



```python
import time
import tracemalloc

def polygonal_number(s, n):
    if s < 3 or n < 1:
        return None

    numerator = ((s - 2) * n**2) - ((s - 4) * n)
    result = numerator // 2
    return result

if __name__ == "__main__":

    s_sides = 5
    n_term = 10

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

    result_number = polygonal_number(s_sides, n_term)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Calculating the {n_term}-th {s_sides}-gonal number.")
    print(f"Result: {result_number}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")


    s_sides_tri = 3
    n_term_tri = 12

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

    result_tri = polygonal_number(s_sides_tri, n_term_tri)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time_tri = end_time - start_time

```

```
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

VIT BHOPAL

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 an−1 ≡ 1 mod n for all a coprime to n.

AIM/OBJECTIVE(s):

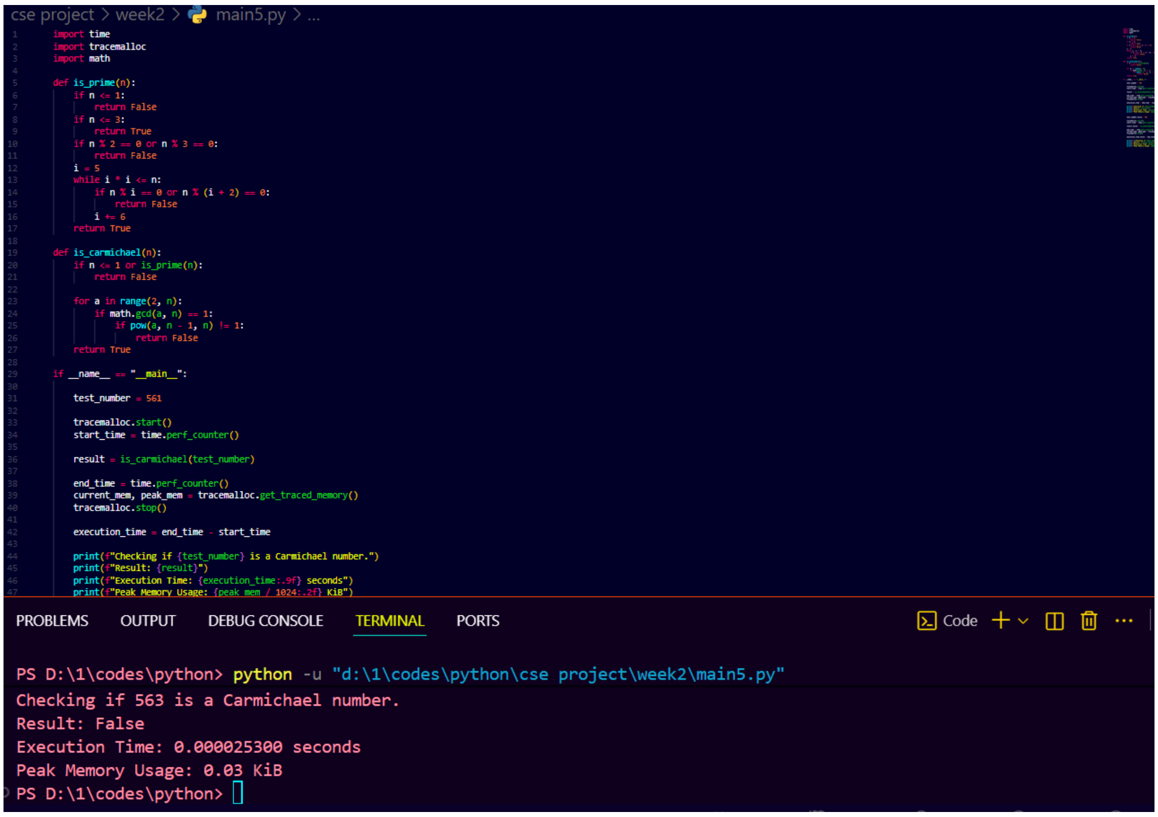
To write a Python function `is_carmichael(n)` that checks if a given composite number `n` is a Carmichael number. A number is a Carmichael number if it is composite and satisfies $a^{n-1} \equiv 1 \pmod n$ for all integers `a` that are coprime to `n`.

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 > ...

```python
import time
import tracemalloc
import math

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_carmichael(n):
    if n <= 1 or is_prime(n):
        return False

    for a in range(2, n):
        if math.gcd(a, n) == 1:
            if pow(a, n - 1, n) != 1:
                return False
    return True

if __name__ == "__main__":

    test_number = 561

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

    result = is_carmichael(test_number)

    end_time = time.perf_counter()
    current_mem, peak_mem = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    execution_time = end_time - start_time

    print(f"Checking if {test_number} is a Carmichael number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```

```
PS D:\1\codes\python> python -u "d:\1\codes\python\cse project\week2\main5.py"
Checking if 563 is a Carmichael number.
Result: False
Execution Time: 0.000025300 seconds
Peak Memory Usage: 0.03 KiB
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.