



# 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:

**Practical Index**

<b>S. No.</b>	<b>Title of Practical</b>	<b>Date of Submission</b>	<b>Signature of Faculty</b>
<b>1</b>	Function to Calculate Factorial	4/10/2025	
<b>2</b>	Function to Check for Palindrome Number.	4/10/2025	
<b>3</b>	Function mean_of_digits(n) that returns the average of all digits in a number.	4/10/2025	
<b>4</b>	Function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	4/10/2025	
<b>5</b>	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.



**Practical No: 3****Date: 4/10/2025**

**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.