VIT BHOPAL

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	Deficient Number Check	16/11/20225	
2	Harshad Number Check	16/11/20225	
3	Automorphic Number Check	16/11/20225	
4	Partition Function Calculator	16/11/20225	
5	Prime Factorization	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Deficient Number Check

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : A number is deficient if the sum of its proper divisors is less than the number itself. For example, 15: Divisors are 1, 3, 5. Sum = 1 + 3 + 5 = 9. Since 9 < 15, it is deficient. For 12: Divisors are 1, 2, 3, 4, 6. Sum = 16. Since 16 > 12, it is abundant (not deficient).

RESULTS ACHIEVED:

main > week3 > main1.py > ...

```python
import time
import tracemalloc
import math

def get_proper_divisors_sum(n):
    if n <= 1:
        return 0
    total = 1
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            total += i
            if i * i != n:
                total += n // i
    return total

def is_deficient(n):
    if n <= 0:
        return False
    return get_proper_divisors_sum(n) < n

if __name__ == "__main__":

    test_number = 15

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

    result = is_deficient(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 deficient number.")
    print(f"Result: {result}")
    print(f"Execution Time: {execution_time:.6f} seconds")
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main1.py"
Checking if 15 is a deficient number.
Result: True
Execution Time: 0.000051500 seconds
Peak Memory Usage: 0.12 KiB

Checking if 12 is a deficient number.
Result: False
Execution Time: 0.000019100 seconds
Peak Memory Usage: 0.08 KiB
PS D:\cseproject\cse project>
```

SKILLS ACHIEVED :

Number classification concepts. Code reuse (using similar logic for different mathematical properties).

Practical No: 2

Date: 16/11/2025

TITLE:Harshad Number Check

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

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION : A Harshad number (or Niven number) is an integer that is divisible by the sum of its digits. For 18: Sum of digits = 1 + 8 = 9. Since 18 is divisible by 9, it is a Harshad number. For 19: Sum of digits = 1 + 9 = 10. Since 19 is not divisible by 10, it is not.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL

Result:

```python
import time
import tracemalloc

def is_harshad(n):
    if n <= 0:
        return False

    original_n = n
    sum_digits = 0
    while n > 0:
        sum_digits += n % 10
        n //= 10

    return original_n % sum_digits == 0

if __name__ == "__main__":

    test_number = 18

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

    result = is_harshad(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 Harshad number.")
    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 = 19
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main2.py"
Checking if 18 is a Harshad number.
Result: True
Execution Time: 0.000009500 seconds
Peak Memory Usage: 0.00 KiB

Checking if 19 is a Harshad number.
Result: False
Execution Time: 0.000006800 seconds
Peak Memory Usage: 0.00 KiB
```

SKILLS ACHIEVED :

- Digit manipulation.

- Basic divisibility logic.
- Handling edge cases (zero/negative numbers).

Practical No: 3

Date: 16/11/2025

TITLE : Automorphic Number Check

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

METHODOLOGY & TOOL USED

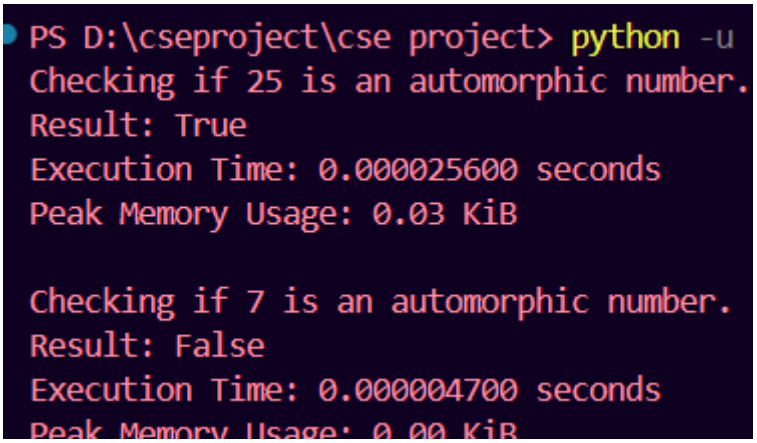
Python programming language

BRIEF DESCRIPTION : An automorphic number (or circular number) is a number whose square ends in the same digits as the number itself. For 25: Square is 625. The last 2 digits are 25. Match -> True. For 7: Square is 49. The last digit is 9. No Match -> False.

RESULTS ACHIEVED:

main > week3 > main3.py > is_automorphic

```python
import time
import tracemalloc

def is_automorphic(n):
    if n < 0:
        return False

    square = n * n

    temp_n = n
    divisor = 1
    while temp_n > 0:
        divisor *= 10
        temp_n //= 10


    if n == 0:
        divisor = 10

    last_digits = square % divisor

    return last_digits == n

if __name__ == "__main__":

    test_number = 25

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

    result = is_automorphic(test_number)

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



SKILLS ACHIEVED :

- Mathematical digit extraction.
- Understanding magnitudes (powers of 10).
- Conditional logic.

Practical No: 4

Date: 16/11/2025

TITLE : Pronic Number Check

AIM/OBJECTIVE(s) : To write a Python function `is_pronic(n)` that checks if a number is the product of two consecutive integers (i.e., n = k * (k + 1)).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : A pronic number (or oblong number) is a number which is the product of two consecutive integers. For `56`: `sqrt(56) ≈ 7.48`. `int(7.48) = 7`. Test `7 * 8 = 56`. Match -> True. For `50`: `sqrt(50) ≈ 7.07`. `int(7.07) = 7`. Test `7 * 8 = 56 != 50`. No Match -> False.

Result:

main > week3 > main4.py > is_pronic

```python
import time
import tracemalloc
import math

def is_pronic(n):
    if n < 0:
        return False

    k = int(math.sqrt(n))
    return k * (k + 1) == n

if __name__ == "__main__":

    test_number = 56 # 7 * 8

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

    result = is_pronic(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 pronic number.")
    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 = 50

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

    result_false = is_pronic(test_number_false)
```

```
Checking if 56 is a pronic number.
Result: True
Execution Time: 0.000040800 seconds
Peak Memory Usage: 0.05 KiB

Checking if 50 is a pronic number.
Result: False
Execution Time: 0.000019500 seconds
Peak Memory Usage: 0.75 KiB
```

SKILLS ACHIEVED :

- Mathematical optimization.
- Understanding integer properties.
- Using standard library math functions.

Practical No: 5

Date: 16/11/2025

TITLE : Prime Factorization

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

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : The code performs prime factorization using trial division. It efficiently divides out all factors of 2 first, then checks odd factors. This reduces the complexity compared to checking every number. For 315:

- Not div by 2.
- Div by 3 -> 105. Factors: [3]
- Div by 3 -> 35. Factors: [3, 3]
- Not div by 3.
- Div by 5 -> 7. Factors: [3, 3, 5]
- Loop ends (5*5 > 7 is false, but next step is i=7).
- Remaining n is 7. Factors: [3, 3, 5, 7].

RESULTS ACHIEVED :

main > week3 > main5.py > ...

```python
import time
import tracemalloc

def prime_factors(n):
    factors = []
    # Handle divisibility by 2
    while n % 2 == 0:
        factors.append(2)
        n = n // 2

    # Handle odd factors
    i = 3
    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n = n // i
        i += 2

    # If n is a prime greater than 2
    if n > 2:
        factors.append(n)

    return factors

if __name__ == "__main__":

    test_number = 315 # 315 = 3 * 3 * 5 * 7

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

    result = prime_factors(test_number)

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

```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week3\main5.py"
Calculating prime factors of 315.
Result: [3, 3, 5, 7]
Execution Time: 0.000032500 seconds
Peak Memory Usage: 0.03 KiB

Calculating prime factors of 37.
Result: [37]
Execution Time: 0.000017800 seconds
Peak Memory Usage: 0.03 KiB
```

SKILLS ACHIEVED :

- Prime factorization algorithm (Trial Division).
- List manipulation.
- Loop optimization.