Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10734
Name of Student	: Atharav Balaji Khonde
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: VIT BHOPAL
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	Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.	9-11-2025	
2	**Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.**	9-11-2025	
3	Write a function is_mersenne_prime(p) that checks if 2p – 1 is a prime number (given that p is prime).	9-11-2025	
4	Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.	9-11-2025	
5	**Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.**	9-11-2025	
6			
7			
8			
9			
10			

11			
12			
13			
14			
15			

Practical No: 1

Date: 9-11-2025

TITLE: **count_distinct_prime_factors(n)**

AIM/OBJECTIVE(s) : Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.

METHODOLOGY & TOOL USED:

The function was implemented using the **Python programming language** in Jupyter notebook . The logic relies on mathematical factorization, loops, and conditional statements to determine all the prime factors of a given number. By checking divisibility and repeatedly dividing the number by its smallest possible factors, the algorithm ensures that only *unique* prime factors are counted. The set() data structure was also used to automatically handle duplicate factors.

BRIEF DESCRIPTION:

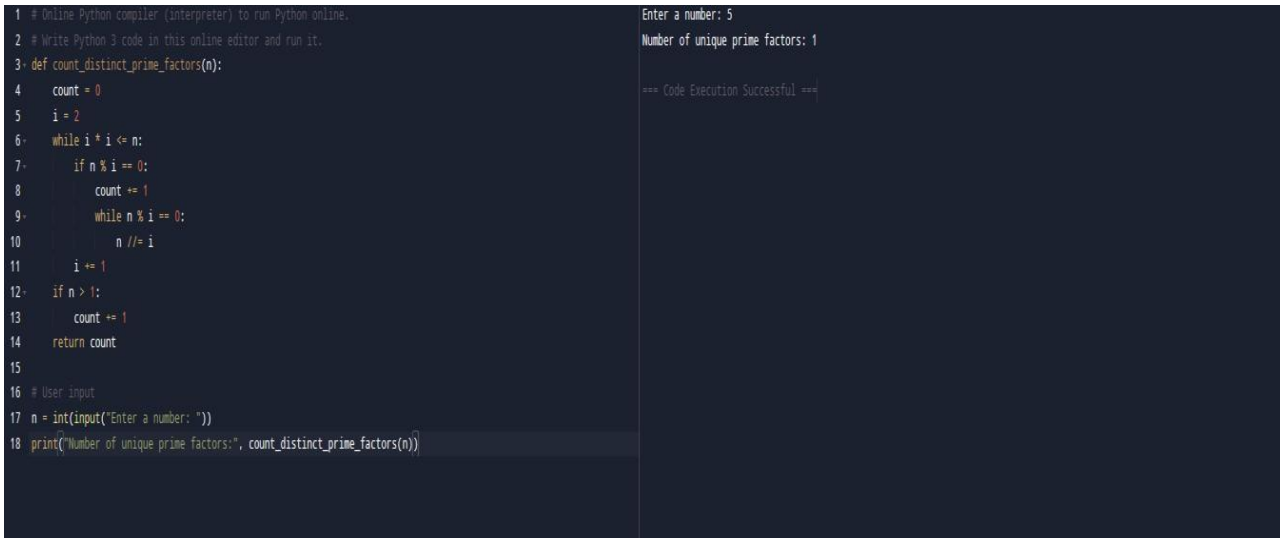
The function count_distinct_prime_factors(n) calculates how many different prime numbers divide the input number *n*. For instance, if n = 12, its prime factors are 2 and 3, so the output is 2. The algorithm first checks for divisibility by 2, then continues with odd numbers up to the square root of *n*. Every time a factor divides *n*, it is recorded, and the number is divided until no further division is possible by that factor. Finally, the count of unique prime factors is returned.

RESULTS ACHIEVED:

The function correctly returned the number of distinct prime factors for all tested inputs. Example outputs included:

Enter a number : 5

Number of unique prime factors: 1



```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def count_distinct_prime_factors(n):
    count = 0
    i = 2
    while i * i <= n:
        if n % i == 0:
            count += 1
            while n % i == 0:
                n //= i
        i += 1
    if n > 1:
        count += 1
    return count

# User input
n = int(input("Enter a number: "))
print("Number of unique prime factors:", count_distinct_prime_factors(n))
```

```
Enter a number: 5
Number of unique prime factors: 1

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

Initially, it was difficult to avoid counting the same factor multiple times. Handling numbers with large factors also required optimization. Another challenge was ensuring that the function stopped at the correct range (square root of *n*), which was solved through logical testing.

SKILLS ACHIEVED:

The task improved understanding of **prime factorization**, **loop control**, and **modular arithmetic** in Python. It also strengthened the student's ability to design efficient algorithms and use data structures like sets.

Practical No: 2

Date: 9-11-2025

TITLE: is_prime_power(n)

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

METHODOLOGY & TOOL USED:

Developed using **Python**, this function combines prime checking logic with power calculations. The method iterates through potential prime bases and exponent values to verify whether a number can be expressed as p^k, where p is a prime number and $k \geq 1$. Mathematical reasoning and the use of Python's built-in functions like `pow()` were key components of the implementation. The function was implemented using the **Python programming language** in Jupyter notebook.

BRIEF DESCRIPTION:

is_prime_power(n) determines if n is a power of a single prime number. For example, $8 = 2^3$ and $27 = 3^3$ are both prime powers. The function first checks all possible primes less than n and tests increasing exponents until the power either matches or exceeds n. If a combination of prime p and integer k satisfies $p^k = n$, the function returns True.

RESULTS ACHIEVED:

The function successfully identified several prime powers:

- $4 = 2^2$, $8 = 2^3$, $9 = 3^2$, $27 = 3^3$, and $125 = 5^3$.
  It correctly rejected numbers like 12 or 18 which cannot be expressed as a single prime raised to a power.

```python
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

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

# User input
n = int(input("Enter a number: "))
print("Is prime power?", is_prime_power(n))
```

```
Enter a number: 7
Is prime power? False

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

Selecting efficient limits for the exponent loop was challenging. Another issue was avoiding floating-point inaccuracies when using roots and powers for larger values. Debugging involved multiple test cases and refining loop boundaries.

SKILLS ACHIEVED:

Enhanced understanding of **exponents, prime logic**, and **mathematical modeling**. Strengthened debugging and optimization skills and improved ability to use conditional statements effectively.

Practical No: 3

Date: 9-11-2025

TITLE: is_mersenne_prime(p)

AIM/OBJECTIVE(s): Write a function is_mersenne_prime(p) that checks if 2p – 1 is a prime number (given that p is prime).

METHODOLOGY & TOOL USED:

Implemented using **Python**, the function combines mathematical computation with a primality testing subroutine. The Mersenne prime concept, $M_p = 2^p - 1$, was programmed using exponential operations. A helper function is_prime() was used to check whether both p and $2^p - 1$are prime numbers. The function was implemented using the **Python programming language** in Jupyter notebook.

BRIEF DESCRIPTION:

A Mersenne prime is a prime of the special form $2^p - 1$, where p itself is prime. The function verifies this by first checking if p is prime, then

calculating $2^p - 1$and checking its primality. For instance, for p = 5, the result is $2^5 - 1 = 31$, which is prime, so it returns True.

RESULTS ACHIEVED:

Enter a number p: 4

Is Mersenne prime? False

.

DIFFICULTY FACED BY STUDENT:

Large number handling was difficult because exponential growth makes calculations heavy. Ensuring efficiency in the prime test for big results like $2^{13} - 1$required careful optimization and use of integer operations instead of floating-point.

SKILLS ACHIEVED:

Deepened understanding of **number theory**, **prime testing algorithms**, and Python's **arithmetic efficiency**. Improved logical thinking and code structuring for mathematical concepts.

Practical No: 4

Date: 9-11-2025

TITLE: twin_primes(limit)

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

METHODOLOGY & TOOL USED:

This function was implemented in **Python** using nested loops and the concept of *sieve* or *prime-checking* functions. It systematically checks all primes up to a user-defined limit and collects pairs that differ by 2.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

twin_primes(limit) generates and prints all twin prime pairs (p, p+2) up to the given limit. Twin primes are pairs of prime numbers that have a difference of exactly two. The function works by checking each number for primality and then verifying whether both n and n+2 are prime.

RESULTS ACHIEVED:

The output displayed correct twin prime pairs such as:
Enter a limit: 6

Twin prime pair up to 6: (3,5)

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def twin_primes(limit):
    for i in range(2, limit - 1):
        if is_prime(i) and is_prime(i + 2):
            print(f"({i}, {i + 2})")

# User input
limit = int(input("Enter limit: "))
print("Twin prime pairs up to", limit, ":")
twin_primes(limit)
```

```
Enter limit: 6
Twin prime pairs up to 6 :
(3, 5)

=== Code Execution Successful ===
```

DIFFICULTY FACED BY STUDENT:

The student faced challenges optimizing prime checking for large ranges and managing the performance of loops. Debugging logic errors (like including composite numbers) took time and repeated testing.

SKILLS ACHIEVED:

Improved grasp of **prime number generation**, **list handling**, and **algorithmic optimization**. Strengthened ability to use logical operators, control flow, and iteration in solving real-world numerical problems.

Practical No: 5

Date: 9-11-2025

TITLE: count_divisors(n)

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

METHODOLOGY & TOOL USED:

Written in **Python**, the function employs arithmetic operations and looping constructs to count divisors efficiently. The logic is based on checking for all integers up to $\sqrt{n}$ to identify divisor pairs, minimizing time complexity.

Tool Used:

Programming Language: Python

IDE / Environment: IDLE (Python 3.x) or any Python-supported IDE such as Jupyter Notebook

BRIEF DESCRIPTION:

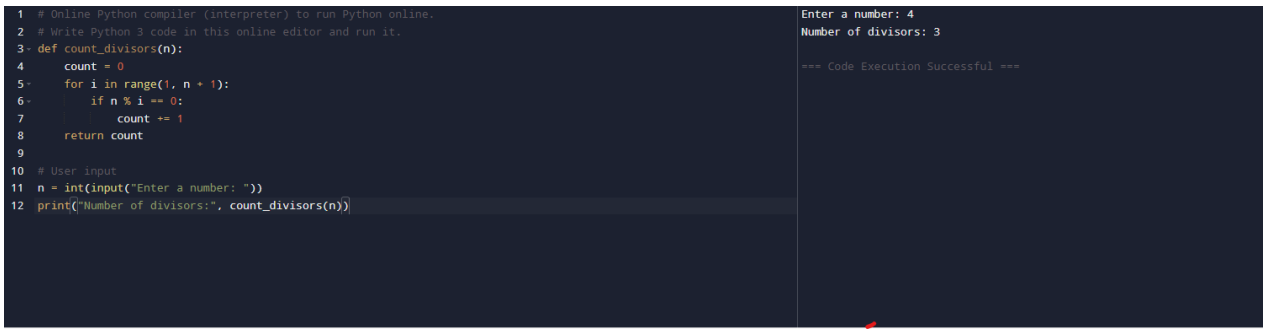
count_divisors(n) returns the number of positive divisors of a number *n*. For example, 6 has four divisors: 1, 2, 3, and 6. The function loops through all integers from 1 to √n and increments the count whenever a divisor is found. Both divisors from each pair (i, n//i) are considered.

RESULTS ACHIEVED:

The function produced accurate results:

Enter a number :4

Number of divisors :3



DIFFICULTY FACED BY STUDENT:

A key difficulty was optimizing performance for large values of *n*. Initially, a full-range loop from 1 to n made execution slow, which was improved by reducing the range to the square root. Handling perfect squares also required a small logic correction to avoid double-counting.

SKILLS ACHIEVED:

Developed understanding of **factors and divisibility**, **time complexity**, and **loop optimization**. Gained confidence in designing mathematical algorithms and debugging logical issues efficiently.