Lab Manual

Practical and Skills Development

CERTIFICATE

Registration No	: 25BAI10749
Name of Student	: Aniruddha Majumdar
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	Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.	23-11-2025	

2	Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.	**23-11-2025**	
3	Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.	**23-11-2025**	

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.	**23-11-2025**	

5			
6			
7			

8			
9			
10			
11			
12			

13			
14			
15			

Practical No: 1

Date: 23-11-2025

TITLE: **probabilistic Miller–Rabin**

AIM/OBJECTIVE(s) : Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.

METHODOLOGY & TOOL USED:

□
- Use modular exponentiation to compute power efficiently.

□
- Decompose n-1 into 2^r * d.

□
- Perform k rounds of testing with random bases.

□
- Check the conditions of strong probable prime.

Tool Used:

Programming Language: Python

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

BRIEF DESCRIPTION:

This function performs a randomized test to check whether a number n is prime. It repeatedly tests n using randomly selected bases, reducing the probability of error with each round.

RESULTS ACHIEVED:

Example Output:

Enter number: 6

Enter rounds k: 5

Probably Prime? False

Execution Time: 0.003022909164428711 seconds

Memory Used: 56.2578125 KB

```python
import random
import time
import tracemalloc

n = int(input("Enter number: "))
k = int(input("Enter rounds k: "))

tracemalloc.start()
start = time.time()

def is_prime_miller_rabin(n, k):
    if n < 2:
        return False
    small = [2,3,5,7,11,13]
    if n in small:
        return True
    if any(n % p == 0 for p in small):
        return False

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

    for _ in range(k):
        a = random.randrange(2, n-2)
        x = pow(a, d, n)
        if x in (1, n - 1):
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True



result = is_prime_miller_rabin(n, k)

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

print("Probably Prime?", result)
print("Execution Time:", end - start, "seconds")
print("Memory Used:", peak/1024, "KB")
```

```
Enter number: 6
Enter rounds k: 5
Probably Prime? False
Execution Time: 0.003022909164428711 seconds
Memory Used: 56.2578125 KB
```

DIFFICULTY FACED BY STUDENT:

- Understanding decomposition of n-1 into factors of 2.
- Handling modular exponentiation correctly.
- Ensuring that all edge cases (n < 2, even numbers) work properly.
- Logic complexity of multiple checks in each iteration.

SKILLS ACHIEVED:

. Strong understanding of **primality testing algorithms**.

. Implementation of **modular arithmetic** and **fast exponentiation**.

. Ability to convert mathematical algorithms into working Python code.

. Improved debugging and logical reasoning skills.

Practical No: 2

Date: 23-11-2025

TITLE: Pollard's Rho Algorithm

AIM/OBJECTIVE(s): To implement **Pollard's Rho integer factorization algorithm**, which finds non-trivial factors of large composite numbers efficiently.

METHODOLOGY & TOOL USED:

Select a pseudo-random polynomial function f(x) = x^2 + c mod n.

Use Floyd's cycle-finding method (tortoise–hare).

Keep computing gcd(|x - y|, n) until a non-trivial factor is found.

Tools used: Python built-ins (gcd), random number generation.

Tool Used:

Programming Language: Python

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

BRIEF DESCRIPTION:

The algorithm uses pseudo-random sequences to discover a factor of a composite number. It is fast, efficient, and widely used in cryptanalysis and number theory.

```python
import random
import math
import time
import tracemalloc

def pollard_rho(n):
    if n % 2 == 0:
        return 2
    x = random.randrange(2, n - 1)
    y = x
    c = random.randrange(1, n - 1)
    d = 1
    while d == 1:
        x = (x*x + c) % n
        y = (y*y + c) % n
        y = (y*y + c) % n
        d = math.gcd(abs(x - y), n)
        if d == n:
            return pollard_rho(n)
    return d

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

tracemalloc.start()
start = time.time()

result = pollard_rho(n)

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

print("Factor Found:", result)
print("Execution Time:", end - start, "seconds")
print("Memory Used:", peak/1024, "KB")
```

```
Enter number: 4
Factor Found: 2
Execution Time: 0.0005011558532714844 seconds
Memory Used: 19.1123046875 KB
```

RESULTS ACHIEVED:

Example Output:

Enter number: 4

Factor Found: 2

Execution Time: 0.0005011558532714844 seconds

Memory Used: 19.1123046875 KB

DIFFICULTY FACED BY STUDENT:

- Hard to understand cycle detection (tortoise and hare method).
- Managing infinite loops when factor is not found quickly.
- Selecting good values of c and initial seeds.
- Handling cases when the algorithm returns n again instead of a factor.

SKILLS ACHIEVED:

- Understanding of **probabilistic factorization algorithms**.
- Use of **GCD**, modular arithmetic, and random functions.
- Learning cycle detection techniques used in algorithms.
- Improved algorithmic thinking and use of mathematical functions.

Practical No: 3

Date: 23-11-2025

TITLE: Approximate Riemann Zeta Function (zeta_approx(s, terms))

AIM/OBJECTIVE(s):

Write a function zeta_approx(s, terms) that approximates

the Riemann zeta function ζ(s) using the first 'terms' of

the series.

METHODOLOGY & TOOL USED:

- Use a loop to compute summation:

$$\sum_{n=1}^{k} \frac{1}{n^s}$$

- Ensure precision for fractional powers using Python’s floating-point arithmetic.

Tool Used:

Programming Language: Python

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

BRIEF DESCRIPTION:

The function calculates an approximate value of the zeta function by summing the first terms in the infinite series. Increasing terms improves accuracy.

RESULTS ACHIEVED:

Example Output:

Enter s: 4

Enter number of terms: 2

Zeta Approx: 1.0625

Execution Time: 0.00034904479980 46875 seconds

Memory Used: 19.0419921875 KB

```python
import time
import tracemalloc

def zeta_approx(s, terms):
    total = 0.0
    for n in range(1, terms+1):
        total += 1 / (n ** s)
    return total

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

tracemalloc.start()
start = time.time()

result = zeta_approx(s, terms)

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

print("Zeta Approx:", result)
print("Execution Time:", end - start, "seconds")
print("Memory Used:", peak/1024, "KB")
```

```
Enter s: 4
Enter number of terms: 2
Zeta Approx: 1.0625
Execution Time: 0.0003490447998046875 seconds
Memory Used: 19.0419921875 KB
```

DIFFICULTY FACED BY STUDENT:

- Handling floating-point precision issues.
- Slow computation for large terms.
- Understanding convergence behavior of series.
- Implementing exponent and summation efficiently.

SKILLS ACHIEVED:

- Knowledge of **infinite series approximation**.
- Use of loops with mathematical expressions.
- Understanding numerical precision.
- Improved mathematical programming skills.

Practical No: 4

Date: 23-11-2025

TITLE: Partition Function (partition_function(n))

AIM/OBJECTIVE(s): 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.

METHODOLOGY & TOOL USED:

- Use dynamic programming or recursive memoization.
- Generate partitions using previously computed values.
- Tool used: Python (recursion, lists, memoization dictionary).

Tool Used:

Programming Language: Python

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

BRIEF DESCRIPTION:

The partition function grows rapidly and cannot be computed by brute force for large n. Using dynamic programming, the function calculates p(n) using already-known values for smaller integers.

RESULTS ACHIEVED:

Example Output:

Enter n: 3
Partition p(n): 3
Execution Time: 0.0002810955047607422 seconds
Memory Used: 19.0419921875 KB

```python
import time
import tracemalloc

def partition_function(n):
    dp = [0] * (n + 1)
    dp[0] = 1
    for i in range(1, n + 1):
        for j in range(i, n + 1):
            dp[j] += dp[j - i]
    return dp[n]

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

tracemalloc.start()
start = time.time()

result = partition_function(n)

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

print("Partition p(n):", result)
print("Execution Time:", end - start, "seconds")
print("Memory Used:", peak/1024, "KB")
```

```
Enter n: 3
Partition p(n): 3
Execution Time: 0.0002810955047607422 seconds
Memory Used: 19.0419921875 KB
```

DIFFICULTY FACED BY STUDENT:

Understanding recurrence relation for partition numbers.

High time complexity for naïve implementations.

Handling large values of n without overflow or excessive recursion depth.

Designing efficient memoization.

Skills Achieved:

Understanding of combinatorics and number theory.

Ability to implement dynamic programming.

Handling recursive problems with optimization.

Experience with mathematically intensive algorithms.

VIT
BHOPAL