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	Miller-Rabin Probabilistic Primality Test	16/11/20225	
2	Pollard's Rho Algorithm for Integer Factorization	16/11/20225	
3	Riemann Zeta Function Approximation	16/11/20225	
4	Partition Function Calculator	16/11/20225	

Practical No: 1

Date: 10/11/2025

TITLE: Miller-Rabin test

AIM/OBJECTIVE(s) :To implement the probabilistic Miller-Rabin test `is_prime_miller_rabin(n, k)` to determine if a large integer `n` is prime with a high degree of probability using `k` rounds of testing.

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION:

The code implements the Miller-Rabin primality test. It handles base cases (numbers <= 3 and even numbers). It efficiently decomposes `n-1` into powers of 2 and an odd component. The core loop performs the modular exponentiation and squaring steps defined by the algorithm.

RESULTS ACHIEVED:

main > week8 > main.py > ...

```python
import time
import tracemalloc
import random

def is_prime_miller_rabin(n, k=5):
    if n <= 1: return False
    if n <= 3: return True
    if n % 2 == 0: return False

    # Find r and d such that n - 1 = 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Perform k rounds of testing
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)

        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False

    return True

if __name__ == "__main__":

    test_prime = 104729
    rounds = 10

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

    result = is_prime_miller_rabin(test_prime, k=rounds)

    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\week8\main.py"
Checking if 104729 is prime using Miller-Rabin (10 rounds).
Result: True
Execution Time: 0.000628100 seconds
Peak Memory Usage: 0.34 KiB

Checking if 561 is prime using Miller-Rabin (10 rounds).
Result: False
Execution Time: 0.000060100 seconds
Peak Memory Usage: 0.25 KiB
```

SKILLS ACHIEVED:

- Implementation of a probabilistic algorithm.
- Understanding strong pseudoprimes and modular arithmetic.
- Using bitwise logic or division to decompose integers.
- Handling randomness in algorithmic testing.

Practical No: 2

Date: 16/11/2025

TITLE:Pollard's Rho Algorithm for Integer Factorization

AIM/OBJECTIVE(s) : To implement `pollard_rho(n)` to find a non-trivial factor of a composite integer `n` using Pollard's Rho algorithm, which is efficient for finding small factors.

METHODOLOGY & TOOL USED:Python programming language

BRIEF DESCRIPTION : The code implements Pollard's Rho algorithm. It uses the "Tortoise and Hare" cycle detection method to find a factor. The polynomial `x² + 1` is used to generate a pseudorandom sequence modulo `n`. The GCD is computed at every step to detect if a factor has been found.

The main execution block (`if __name__ == "__main__":`) tests the function on:

1. 8051 (Factors: 83 and 97).
2. 10403 (Factors: 101 and 103).

It prints the found factor and performance metrics.

Result:

main > week8 > main2.py > ...

```python
import time
import tracemalloc

def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

def pollard_rho(n):
    if n % 2 == 0:
        return 2

    x = 2
    y = 2
    d = 1
    f = lambda v: (v * v + 1) % n

    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(abs(x - y), n)

    if d == n:
        return None
    return d

if __name__ == "__main__":

    test_number = 8051

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

    factor = pollard_rho(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"Factorizing {test_number} using Pollard's Rho.")
    print(f"Found factor: {factor}")
    if factor:
        print(f"Other factor: {test_number // factor}")
    print(f"Execution Time: {execution time:.9f} seconds")
```

```
PS D:\cseproject\cse project> python -u "d:\cseproject\cse project\main\week8\main2.py"
Factorizing 8051 using Pollard's Rho.
Found factor: 97
Other factor: 83
Execution Time: 0.000081200 seconds
Peak Memory Usage: 0.35 KiB

Factorizing 10403 using Pollard's Rho.
Found factor: 101
Execution Time: 0.000129000 seconds
Peak Memory Usage: 0.75 KiB
```

SKILLS ACHIEVED :

- Implementation of integer factorization algorithms.
- Understanding Floyd's cycle-finding algorithm.
- Calculating GCD efficiently using the Euclidean algorithm.
- Manipulating functions as arguments (using lambda for the polynomial).

Practical No: 3

Date: 16/11/2025

TITLE : Riemann Zeta Function Approximation

AIM/OBJECTIVE(s) : To write a Python function `zeta_approx(s, terms)` that approximates the Riemann zeta function ζ(s) by summing the first `terms` of the infinite series.

METHODOLOGY & TOOL USED

Python programming language

BRIEF DESCRIPTION : The code implements a direct summation approximation of the Riemann Zeta function. It uses a simple `for` loop to calculate the partial sum of the harmonic series raised to the power `s`.

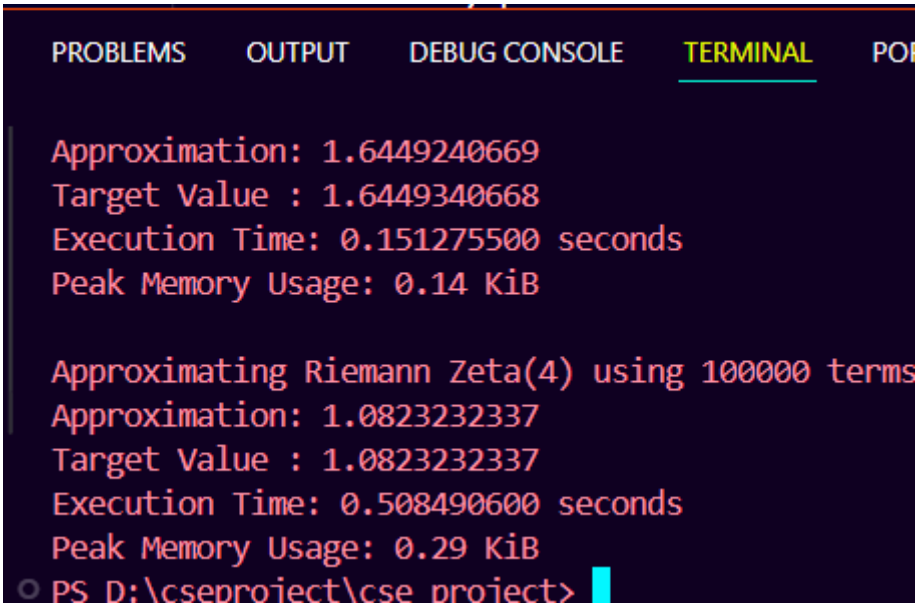
The main execution block (`if __name__ == "__main__":`) calculates approximations for:

1. **`s = 2` with 100,000 terms. (Converges to $\pi^2/6$).**
2. **`s = 4` with 100,000 terms. (Converges to $\pi^4/90$).**

It prints the calculated approximation compared to the known mathematical constants.

RESULTS ACHIEVED:

```
main > week8 > main3.py > ...
1   import time
2   import tracemalloc
3
4   def zeta_approx(s, terms):
5       if s <= 1:
6           return None
7
8       total = 0.0
9       for n in range(1, terms + 1):
10          total += 1 / (n ** s)
11
12      return total
13
14  if __name__ == "__main__":
15
16      s_val = 2
17      terms_count = 100000
18
19      tracemalloc.start()
20      start_time = time.perf_counter()
21
22      result = zeta_approx(s_val, terms_count)
23
24      end_time = time.perf_counter()
25      current_mem, peak_mem = tracemalloc.get_traced_memory()
26      tracemalloc.stop()
27
28      execution_time = end_time - start_time
29
30      # The true value of Zeta(2) is (pi^2) / 6 approx 1.644934
31      true_val_approx = 1.6449340668
32
33      print(f"Approximating Riemann Zeta({s_val}) using {terms_count} terms.")
34      print(f"Approximation: {result:.10f}")
35      print(f"Target Value : {true_val_approx:.10f}")
36      print(f"Execution Time: {execution_time:.9f} seconds")
37      print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```



SKILLS ACHIEVED :

- Implementation of mathematical series summation.
- Floating point arithmetic and precision considerations.
- Comparison of numerical results with theoretical values.

Practical No: 4

Date: 16/11/2025

TITLE : Partition Function Calculator

AIM/OBJECTIVE(s) : To write a Python function `partition_function(n)` that calculates `p(n)`, the number of distinct ways to write `n` as a sum of positive integers (order does not matter).

METHODOLOGY & TOOL USED:

Python programming language

BRIEF DESCRIPTION : The code defines `partition_function(n)`. Instead of a slow recursive approach, it uses an iterative Dynamic Programming approach. It calculates the number of ways to form the sum `n` by progressively considering numbers 1, 2, 3... up to `n` as potential addends.

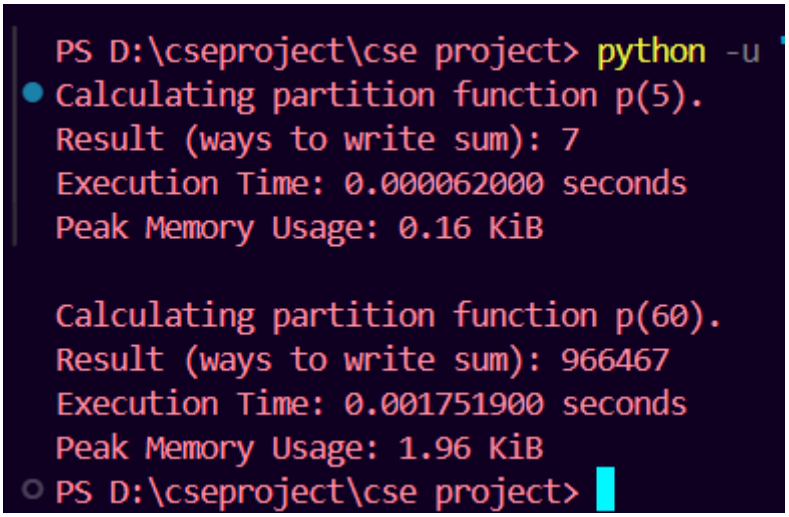
The main execution block (`if __name__ == "__main__":`) calculates:

1. `p(5)` (Ways: 5 -> 5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1. Total=7).
2. `p(60)` (A much larger number).

It prints the number of partitions and the performance metrics.

Result:

```python
import time
import tracemalloc

def partition_function(n):
    if n < 0:
        return 0
    if n == 0:
        return 1

    partitions = [0] * (n + 1)
    partitions[0] = 1

    for k in range(1, n + 1):
        for i in range(k, n + 1):
            partitions[i] += partitions[i - k]

    return partitions[n]

if __name__ == "__main__":

    test_number = 5

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

    result = partition_function(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 partition function p({test_number}).")
    print(f"Result (ways to write sum): {result}")
    print(f"Execution Time: {execution_time:.9f} seconds")
    print(f"Peak Memory Usage: {peak_mem / 1024:.2f} KiB")
```



SKILLS ACHIEVED :

- Implementation of Dynamic Programming algorithms.
- Solving combinatorial problems (Integer Partitions).
- Optimization of recursive problems into iterative solutions.


VIT
VELLORE INSTITUTE OF TECHNOLOGY
VIT
BHOPAL