Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BCE11069
Name of Student	: S Pratik
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:

VIT BHOPAL

Practical Index

S. No.	TITLE of Practical	Date of Submission	Signature of Faculty
1	Factorial of a Number with Time and Memory Tracking	04/10/2025	
2	Palindrome Check of a Number with Time and Memory Utilization	04/10/2025	
3	Program to calculate the Digital Root of a number and measure its execution time and memory usage.	04/10/2025	
4	Digit Mean Calculator with Time and Memory Profiling	04/10/2025	
5	Abundant Number Checker	04/10/2025	
6	Program to check if a number is a Deficient Number	11/10/2025	
7	Function for harshad number	11/10/2025	
8	Program that Checks if number`s square ends with the number itself.	11/10/2025	
9	Function To Check If a Number is a Product of Two Consecutive Numbers	11/10/2025	
10	A function prime_factors(n) that returns the list of prime factors of a number	11/10/2025	
11	Program to calculate the Count of Distinct Prime Factors of a number.	02/11/2025	

12	Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and k ≥ 1.	02/11/2025	
13	Testing Mersenne Primes using Python (with Memory and Execution Time Analysis	02/11/2025	
14	: A function of twin_primes(limit) with time and memory utilization	02/11/2025	
15	Efficient Divisor Counting Algorithm with Memory and Execution Time Profiling	02/11/2025	
16	Aliquot Sum Computation and Resource Usage Estimation	09/11/2025	
17	Amicable Numbers Detector with Performance Analysis	09/11/2025	
18	Multiplicative Persistence Program: Counting Digit Multiplication Steps	09/11/2025	
19	Program to Check if a Number is Highly Composite	09/11/2025	
20	Modular Exponentiation Calculator Using Binary Exponentiation (Fast Exponentiation)	09/11/2025	
21	Fibonacci Prime Checker with Performance Analysis	16/11/2025	
22	Modular Multiplicative Inverse	16/11/2025	

S. No.	TITLE of Practical	Date of Submission	Signature of Faculty
23	Quadratic Residue Check (Legendre Symbol)	16/11/2025	
24	Multiplicative Order of a Modulo n	16/11/2025	
25	Implementation of Chinese Remainder Theorem Using Extended Euclidean Algorithm	16/11/2025	
26	Polygonal Number Calculator with Parameter Validation	23/11/2025	
27	Performance Measuring Lucas Number Sequence Generator	23/11/2025	
28	Perfect Power Identification by Exponent Scanning	23/11/2025	
29	Collatz Sequence Length Calculator	23/11/2025	
30	Carmichael Number Tester Using Fermat Like Condition	23/11/2025	
31	Probabilistic Primality Testing Using the Miller Rabin Algorithm	30/11/2025	
32	Integer Partition Function Using Euler's Pentagonal Number Theorem	30/11/2025	
33	Pollard Rho Integer Factorization Algorithm with Cycle Detection	30/11/2025	
34	Dirichlet Series Approximation of the Riemann Zeta Function	30/11/2025	

PRACTICAL NO.-1

Date: 04/10/2025

TITLE: Function factorial(n) that calculates the factorial of a non-negative integer n (n!) with runtime and memory tracking.

AIM/OBJECTIVE(s):The **AIM** of this program is to calculate the factorial of a non-negative integer entered by the user while also monitoring how long it takes to compute and how much memory is used during the calculation

METHODOLOGY & TOOL USED:

Methodology

1)Recursive Approach for Factorial

2)Execution Time Measurement

3)Memory Usage Measurement Tools Used

Python time module: To measure execution time.

Python tracemalloc module: To measure memory usage.

Recursive function: To implement the factorial logic efficiently. Language : Python Tools : Standard Input/Output , VS code Libraries : tracemalloc, time

BRIEF DESCRIPTION:This program finds the factorial of a number you type in. It does this using a function that calls itself repeatedly until it reaches 1. While doing the calculation, it also tells you how long it took and how much memory it used.

RESULTS ACHIEVED:

The program successfully calculates the factorial of any non-negative integer you provide. Along with giving the correct factorial value, it also shows how long the calculation took and how much memory was used. This means you not only get the answer but also a sense of the program's efficiency in terms of speed and memory usage.

DIFFICULTY FACED BY STUDENT:I found understanding recursion challenging, as it requires following a function that calls itself repeatedly until reaching the base case. Additionally, grasping how the program measures execution time and memory usage with time and tracemalloc required careful attention.

SKILLS ACHIEVED:

The program successfully computes the factorial of any non-negative integer provided by the user. In addition to producing the correct factorial value, it also reports the execution time and current memory usage, giving a clear picture of both the correctness and efficiency of the computation.

```python
import time
import tracemalloc

def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

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

num = int(input("Enter a non-negative integer: "))
result = factorial(num)

end_time = time.time()
current, _ = tracemalloc.get_traced_memory()
tracemalloc.stop()

print("Factorial of", num, "is:", result)
print(f"Execution Time: {end_time - start_time:.6f} seconds")
print(f"Memory Usage: {current / 1024:.2f} KB")
```

```
Enter a non-negative integer: 9
Factorial of 9 is: 362880
Execution Time: 6.051478 seconds
Memory Usage: 0.03 KB
```

PRACTICAL NO.-2

Date: 04/10/2025

TITLE: Palindrome of a number with time and memory utilization

AIM/OBJECTIVE(s):

To create a function called palindrome(n) that checks the number reads forward as well as the backwards.

METHODOLOGY & TOOLUSED:

This program is to check the number whether it is a palindrome or not, purely mathematical solution (reversing the number by extracting digits) could avoid string conversion.
Language: Python Tools (like loops and function call):Standard Input/output , IDLE PYTHON
Libraries: sys,time

BRIEF DESCRIPTION:

The program reads number weather it is a palindrome, and reports the memory consumption and execution time. It also displays how much computer memory was used and how long the program took to run.

RESULTS ACHIEVED:

Number: 121

Is Palindrome: True

Time taken: 46468.26microseconds
Current memory usage: 0.03KB Peak memory usage: 0.15 KB

DIFFICULTY FACED BY STUDENT:

Dealing with python syntax and using loops also learning to utilizing libraries for measuring run time and memory utilization.

SKILLS ACHIEVED:

1.Application of system libraries for run time and memory profiling.
2. Understanding of the result formatting in Python and work upon on python dictionaries.

```python
import time
import tracemalloc

Qodo: Test this function
def is_palindrome(number):
    # Handle negative numbers (not typically palindromes)
    if number < 0:
        return False

    # Convert number to string for comparison
    num_str = str(number)
    # Compare string with its reverse
    return num_str == num_str[::-1]

Qodo: Test this function
def main():
    # Test cases
    test_numbers = [121, 12321, 12345, 1000001, -121]

    # Start tracking memory
    tracemalloc.start()

    for num in test_numbers:
        # Measure time
        start_time = time.time()
        result = is_palindrome(num)
        end_time = time.time()

        # Get memory usage
        current, peak = tracemalloc.get_traced_memory()

        print(f"Number: {num}")
        print(f"Is Palindrome: {result}")
        print(f"Time taken: {(end_time - start_time)*1000000:.2f} microseconds")
        print(f"Current memory usage: {current / 1024:.2f} KB")
        print(f"Peak memory usage: {peak / 1024:.2f} KB")
        print("-" * 30)

    # Stop tracking memory
    tracemalloc.stop()

if __name__ == "__main__":
    main()
```

```
Number: 121
Is Palindrome: True
Time taken: 46468.26 microseco
Current memory usage: 0.05 KB
Peak memory usage: 0.13 KB
```

PRACTICAL NO.-3

Date: 04/10/2025

TITLE: Program to calculate the Digital Root of a number and measure its execution time and memory usage.

AIM/OBJECTIVE(s): To write a Python program that computes the digital root of a given number without using built-in functions and to measure its execution time and memory utilization.

METHODOLOGY & TOOL USED: Programming Language: Python
Modules Used: time, tracemalloc
Approach: Modular arithmetic-based formula for efficient digital root calculation.
Tool: Python IDLE / VS Code

BRIEF DESCRIPTION: The digital root of a number is obtained by repeatedly summing its digits until a single-digit result is produced. In this program, instead of performing repeated summation, the formula (1
+ (n - 1) % 9) is used, which gives the digital root directly. The program also calculates the time taken for execution using the time module and memory usage using the tracemalloc module.

RESULTS ACHIEVED: The program correctly calculates the digital root for any positive integer. It displays execution time (in seconds) and memory usage (in bytes).

RESULTS ACHIEVED: The program correctly computes φ(n) for any positive integer.
Example Output:
Enter a positive integer: 9 Euler's Totient Function: 6 Time taken (sec): 0.00024 Memory used (bytes): 480

DIFFICULTY FACED BY STUDENT: Optimizing the loop for large n and using tracemalloc for accurate memory tracking.

CONCLUSION: Euler's Totient Function program was successfully implemented using basic programming constructs. It enhanced understanding of GCD computation, number theory, and performance measurement.

```python
import time
import tracemalloc

n=int(input("enter a number: "))

Qodo: Test this function
def digital_root(n):
    if n < 0:
        return 0
    return (1 + (n - 1) % 9)

tracemalloc.start()
start_time = time.perf_counter()
result = digital_root(n)
end_time = time.perf_counter()
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print("digital root:", result)
print("time taken (sec):", end_time - start_time)
print("memory usage (bytes):", current, peak)
```

```
enter a number: 89
digital root: 8
time taken (sec): 1.720000000204891e-05
memory usage (bytes): 0 0
```

PRACTICAL NO.-4

Date: 04/10/2025

Title

Digit Mean Calculator with Time and Memory Profiling

Aim

To read a positive integer from the user, compute the arithmetic mean of its decimal digits, and display the result along with the program's execution time and memory usage.

Methodology / Tools Used

- The program takes an integer input number from the user using input() and converts it to int.
- A function mean_of_digits(n):
  - Uses a loop with % 10 to extract the last digit and // 10 to remove it.
  - Accumulates the sum of digits in total and counts digits in count until n > 0 becomes false.
  - Returns 0 if count == 0, else returns total / count as the average digit value.
- Execution time is measured with time.time() before and after the computation and printed as end_time - start_time in seconds.
- Memory usage is obtained with psutil.Process(os.getpid()).memory_info().rss in bytes and printed.

Results Achieved

- Correctly computes and prints the mean of the digits of the user-entered integer.
- Reports current process memory usage in bytes at the time of measurement.
- Displays total execution time with 6 decimal places, giving a basic performance profile.

Difficulties Faced by the Student

- Understanding how to decompose a number into digits using integer division (// 10) and modulo (% 10).
- Ensuring the digit loop terminates correctly when n becomes 0.
- Learning how to use psutil and os.getpid() to access process-level memory information.
- Correctly placing time measurements so that input, computation, and printing are included in the execution time.

Skills Achieved

- Implemented a digit-processing loop using basic arithmetic operators on integers.
- Practiced writing and calling a function (mean_of_digits) to encapsulate numeric logic.
- Gained experience with simple performance measurement using time.time().
- Learned to inspect runtime memory usage of a Python program with psutil and os modules.

```python
import time
import os
import psutil

Qodo: Test this function
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
# Input: a positive integer
number = int(input("Enter a number: "))
start_time = time.time()

Qodo: Test this function
def mean_of_digits(n) :
    total = 0
    count = 0
    n = number
    while n > 0:
        digit = n % 10
        total += digit
        count += 1
        n //= 10
        if count ==0 :
            return 0
        else:
            return total/count


print(f"The average of the digits of {number} is {mean_of_digits(number)}")
print(f"memory usage :{memory_usage()} bytes")
end_time = time.time()
print(f"Execution Time: {end_time - start_time:.6f} seconds")
```

```
Enter a number: 78
The average of the digits of 78 is 8.0
memory usage :18145280 bytes
Execution Time: 0.000898 seconds
```

PRACTICAL NO.-5

Date: 04/10/2025

TITLE:Abundant Number Checker

AIM/OBJECTIVE(s): To develop a program that checks whether a given number is an abundant number, while also measuring its runtime and estimating memory usage.

METHODOLOGY & TOOL USED:
•Algorithm: Iterative check of divisors and sum calculation for abundant determination.
•Language: C
•Tools: Standard Input/Output, Time measurement via '<time.h>', memory usage estimation via 'sizeof' and local variable count.

BRIEF DESCRIPTION:
The program requests a number from the user, calculates whether the number is abundant (the sum of its proper divisors exceeds the number), prints the result, and reports both estimated memory usage and execution time.

RESULTS ACHIEVED:
•Efficient abundant number detection for user input.
•Outputs include true/false result, runtime in seconds, and estimated memory usage in bytes.

DIFFICULTY FACED BY STUDENT:
•Estimating stack memory usage provides only an approximation, not accounting for heap or system overhead.
•Disk space tracking was not implemented, as the program does not handle file operations or dynamic memory allocation.

SKILLS ACHIEVED:
•Writing modular C code for mathematical computation.
•Implementing basic runtime and memory profiling.
•Handling user input and formatted output in C.
•Gaining practical experience with core programming concepts like loops, functions, and conditionals.

```c
#include <stdio.h>
#include <time.h>

Qodo: Test this function
void print_memory_usage() {
    int local_vars = sizeof(int) * 3; // n, sum, i
    printf("Estimated memory usage: %d bytes\n", local_vars);
}

Qodo: Test this function
int is_abundant(int n) {
    clock_t start = clock();
    int sum = 0;
    int i;
    for(i = 1; i <= n/2; i++) {
        if(n % i == 0) {
            sum = sum + i;
        }
    }
    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Runtime: %.6f seconds\n", time_spent);
    print_memory_usage();
    return (sum > n) ? 1 : 0;
}

Qodo: Test this function
int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Is abundant: %s\n", is_abundant(n) ? "true" : "false");
    return 0;
}
```

PRACTICAL NO.-6

Date: 11/10/2025

TITLE: Program to check if a number is a Deficient Number.

AIM/OBJECTIVE(s): To write a Python program that calculates the sum of the proper divisors of a given positive integer n and determines whether the number is deficient (i.e., if the sum of its proper divisors is less than n).

METHODOLOGY & TOOL USED:
Programming Language: Python
Approach: Iterative calculation of the sum of proper divisors using trial division optimized by checking up to \sqrt{n}.

Tool: Python IDLE / VS Code

BRIEF DESCRIPTION:
A Deficient Number is a number n where the sum of its proper divisors (all positive divisors excluding n itself) is less than n. For example, for n=10, the proper divisors are 1, 2, and 5, and their sum is 1+2+5=8.
Since 8 < 10, 10 is deficient.

The function is_deficient(n) implements this check:
It initializes the sum of divisors (total) to 1 (since 1 is a proper divisor for all n>1).
It iterates from i=2 up to the square root of n.
If i divides n, both i and its corresponding divisor pair n // i are added to total.
The condition i != n // i prevents double-counting when n is a perfect square.
Finally, it returns True if total < n, indicating the number is deficient

RESULTS ACHIEVED:
The program correctly identifies whether a given positive integer is a deficient number.
Example Output (for n=10):
enter a number: 10 True
(Sum of proper divisors of 10 is 8, and 8 < 10.) Example Output (for n=12 - an abundant number): enter a number: 12
False
(Sum of proper divisors of 12 is 1 + 2 + 3 + 4 + 6 = 16, and 16 is not

< 12.)

DIFFICULTY FACED BY STUDENT (Example):

Correctly implementing the divisor pair logic (i and n // i) while avoiding double-counting for perfect squares and ensuring the check is correctly optimized .

CONCLUSION:

The program successfully implemented an algorithm to classify a number as deficient based on the sum of its proper divisors, demonstrating efficient divisor-finding techniques in number theory.

Code Blame 15 lines (12 loc) · 319 Bytes Raw

```python
n=int(input("enter a number :"))
def is_deficient(n):
    if n <= 1:
        return False
    total = 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            total += i
            if i != n // i and n // i != n:
                total += n // i
    return total < n
print (is_deficient(n))
```

```
enter a number :77
True
```

PRACTICAL NO.-7

Date: 11/10/2025

TITLE:Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.

AIM/OBJECTIVE(s):
The **AIM** is to write a function to determine whether a given integer is a Harshad number, i.e., whether is divisible by the sum of its digits.
To measure how long the program takes to run.
To show the memory used by the function definition.

METHODOLOGY & TOOL USED:
Programming Language: Python (version 3.x)
Built-in libraries/tools used: time (for measuring runtime), sys (for checking memory usage of the function).
Logic: An iterative algorithm multiplies sequential numbers to see if it matches the target number.

BRIEF DESCRIPTION:
A Harshad number (also known as a Niven number) is any positive integer that is exactly divisible by the sum of its digits. For example, 18 is a Harshad number because the sum of its digits is 1 + 8 = 9, and 18 divided by 9 equals 2 with no remainder. This property applies to all numbers: if a number divided by the sum of its digits results in an integer, it is considered a Harshad number. This concept is often used in programming to teach digit manipulation, modular arithmetic, and looping techniques, making it a popular exercise in coding and mathematics education.

RESULTS ACHIEVED:
•For example, : digits sum to 9, and (Harshad).
•For : digits sum to 10, (not Harshad).
•All single-digit numbers are always Harshad numbers because any number divides by itself.

DIFFICULTY FACED BY STUDENT:
Understanding what a "chronic number" is due to the lack of a standard mathematical definition.
Making sure the loop runs correctly and avoids unnecessary increments inside the for-loop.
Recognizing the limitations of sys.getsizeof() since it measures object size, not the total program memory used.
Managing program outputs clearly and accurately.

SKILLS ACHIEVED:
•Understanding of modular arithmetic and digit manipulation.
•Practice with Python loops, conditionals, and basic number theory concepts.
•Improved ability to solve problems related to number properties, useful for coding interviews and competitive programming.

Code Blame 27 lines (21 loc) · 552 Bytes Raw

```python
import time
import tracemalloc

def is_harshad(n):
    original = n
    digit_sum = 0
    while n != 0:
        digit = n % 10
        digit_sum += digit
        n = n // 10
    return original % digit_sum == 0

num = 1729

# Start measuring memory and time
tracemalloc.start()
start = time.time()

result = is_harshad(num)

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

print("Is Harshad:", result)
print("Runtime: {:.8f} seconds".format(end - start))
print("Peak Memory Usage: {:.8f} MB".format(peak / 10**6))
```

```
Is Harshad: True
Runtime: 0.00000215 seconds
Peak Memory Usage: 0.00000000 MB
```

PRACTICAL NO.-8

Date: 11/10/2025

AIM/OBJECTIVE(s):To create a function called automorphic(n) that checks if number`s square ends with the number itself.

METHODOLOGY & TOOLUSED:

This program is to check the number whether it is a automorphic or not, purely mathematical solution (squaring the given number and then mod of 10 exponent by digit or by floor division) could avoid string conversion.

Language: Python Tools (like loops and function call)
:Standard Input/output , IDLE PYTHON

Libraries: sys,time

BRIEF DESCRIPTION: The program reads number weather it is automorphic, and reports the memory consumption and execution time. It also displays how much computer memory was used and how long the program took to run.

RESULTS ACHIEVED:

Number: 76 as 76^2 = 5776

Is automorphic: True

Time taken: 1.335seconds
memory usage: 28 bytes
Number: 35 as 35^2 = 1225
Is automorphic: False
Time taken: 1.0698seconds
Memory usage: 28 bytes

DIFFICULTY FACED BY STUDENT:

Dealing with python syntax and using loops also learning to utilizing libraries for measuring shorter run time and memory utilization.
Understanding the program at the intial point.

SKILLS ACHIEVED:

1.Application of system libraries for run time and memory profiling.

2. Understanding of the result formatting in Python and work upon on python loops .

Code Blame 27 lines (21 loc) · 566 Bytes Raw

```python
import time
import sys

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

    square = n * n
    temp, digits = n, 0
    while temp > 0:
        digits += 1
        temp //= 10

    return square % (10 ** digits) == n

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

start_time = time.time()
result = is_automorphic(n)
end_time = time.time()
time_taken = end_time - start_time

print(f"result of {n} is {result}")
print("time taken is", time_taken, "seconds")
print("memory usage of result:", sys.getsizeof(result),"bytes")
```

```
Enter a number:7
result of 7 is False
time taken is 1.4781951904296875e-05 seconds
memory usage of result: 28 bytes
```

VIT BHOPAL

Practical No: 9

Date: 11/10/2025

TITLE:
Function To Check If a Number is a Product of Two Consecutive Numbers

AIM/OBJECTIVE(s):
To design a Python program that checks if a number is a "chronic number" using specific logic.
To measure how long the program takes to run.
To show the memory used by the function definition.

METHODOLOGY & TOOL USED:
Programming Language: Python (version 3.x)
Built-in libraries/tools used: time (for measuring runtime), sys (for checking memory usage of the function).
Logic: An iterative algorithm multiplies sequential numbers to see if it matches the target number.

BRIEF DESCRIPTION:
The program asks the user to input a number. It then processes through a loop, where a multiplying factor (a) starts at 1 and increases. For each value, it multiplies (a * i) and compares the result to the user's number. If there's a match, the number is marked as "chronic." If not, it is not classified that way. The code also tracks the time taken for the computation and presents the static memory used by the function object, not the full execution memory.

RESULTS ACHIEVED:
The program successfully determined if the entered number matched the custom "chronic" criteria.
The execution time for the operation was shown in seconds.
The memory footprint of the function object was displayed in bytes. Output statements clearly informed the user of the result.

DIFFICULTY FACED BY STUDENT:
Understanding what a "chronic number" is due to the lack of a standard mathematical definition.
Making sure the loop runs correctly and avoids unnecessary increments inside the for-loop.
Recognizing the limitations of sys.getsizeof() since it measures object size, not the total program memory used.
Managing program outputs clearly and accurately.

SKILLS ACHIEVED:
A better understanding of Python's function definitions, loops, and variable management.
Experience in measuring runtime performance using the time module. Knowledge of Python memory measurement techniques and their limitations.
The development of clear, user-friendly input/output messages and report writing.

Code Blame 26 lines (23 loc) · 529 Bytes Raw

```python
import time
import sys

def is_chronic(n):
    a=1
    total=0
    for i in range(n):
        num=a*i
        if num==n:
            total=1
        a+=1
        i+=1
    return total

n1=int(input("Enter the number to be checked: "))
start_time = time.time()
re=is_chronic(n1)
if re==1:
    print("It is a chronic number")
else:
    print("It is not a chronic number")

end_time = time.time()
time_taken = end_time-start_time
print("time taken:",time_taken,"seconds")
print("memory utilised:",sys.getsizeof(is_chronic))
```

```
Enter the number to be checked: 53
It is not a chronic number
time taken: 5.912780761718750e-05 seconds
memory utilised: 160
```

PRACTICAL NO.-10

Date: 11/10/2025

TITLE: A function prime_factors(n) that returns the list of prime factors of a number.

AIM/OBJECTIVE(s): The **AIM** of this program is to develop a python program that accepts a number from the user and displays its prime factors along with the time taken and memory used during execution.

METHODOLOGY & TOOL USED:

Methodology

1)Recursive Approach for Factorial and integration of list 2)Execution Time Measurement

3)Memory Usage Measurement Tools Used

Python time module: To measure execution time.

Python tracemalloc module: To measure memory usage.

Recursive function: To implement the factorial logic efficiently. Language : Python Tools : Standard Input/Output , VS code Libraries : tracemalloc, time

BRIEF DESCRIPTION: This Python program finds all the prime factors in a list of any number you enter. It not only shows which prime numbers multiply to form that number but also tells how much time and memory the program used while running. It's a simple way to understand both how prime factorization works and how efficiently the code performs.

RESULTS ACHIEVED:

The program successfully takes a number as input and displays its prime factors in a list form correctly. It also shows the exact execution time and memory usage, helping to understand how fast and efficient the code is.

DIFFICULTY FACED BY STUDENT:

At first, it was a bit tricky to understand how to break a number into its prime factors using loops and then grouping them in a list but by making it into two different program and then combining it the approach became more clear.
Learning how to use the tracemalloc module to measure memory was also new and slightly confusing.

SKILLS ACHIEVED:

Through this program, I learned how to find prime factors using loops and conditions, how to add list into any function, how to measure a program's execution time and memory usage, and how to write cleaner, more efficient Python code. It also improved my logical thinking and problem-solving skills.

```python
import time
import tracemalloc

Qodo: Test this function
def prime_factors(n):
    tracemalloc.start()
    start = time.time()

    factors = []
    i = 2
    while i * i <= n:
        if n % i == 0:
            factors.append(i)
            n //= i
        else:
            i += 1
    if n > 1:
        factors.append(n)

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

    print("Execution time:", end - start, "seconds")
    print("Memory used:", current / 1024, "KB (current),", peak / 1024, "KB (peak)")

    return factors

num = int(input("Enter a number: "))
print("Prime factors:", prime_factors(num))
```

```
Enter a number: 90
Execution time: 2.193450927734375e-05 seconds
Memory used: 0.03125 KB (current), 0.03125 KB (peak)
Prime factors: [2, 3, 3, 5]
```

PRACTICAL NO.-11

Date: 02/11/2025

TITLE: Program to calculate the Count of Distinct Prime Factors of a number.

AIM/OBJECTIVE(s):

To write a Python program that computes the count of distinct prime factors for a given positive integer n and analyze the time complexity of the implemented algorithm.

METHODOLOGY & TOOL USED:

Programming Language: Python
Approach: Trial division method optimized by iterating up and ensuring each prime factor is counted only once.

Tool: Python IDLE / VS Code

BRIEF DESCRIPTION:

The provided function, count_distinct_prime_factorial(n), calculates how many unique prime numbers divide the input number n.
It iterates starting from i=2.
If i divides n, it is counted as a distinct prime factor, and the inner while loop continuously divides n by i to remove all occurrences of that factor, ensuring the next factor found is distinct.
The iteration stops when i^2 > n.
If the remaining n is greater than 1, it means the remaining value is a prime factor itself, and is counted.

RESULTS ACHIEVED:

The function correctly calculates the number of distinct prime factors. Example Output (for n=60):
(The function call print(count_distinct_prime_factorial(60)) outputs 3)
The prime factors of 60 are 2, 2, 3, 5.
The distinct prime factors are 2, 3, 5.
Count of Distinct Prime Factors: 3

DIFFICULTY FACED BY STUDENT (Example):

Understanding the optimization of stopping the trial division at \sqrt{n} and ensuring the remaining n > 1 is correctly counted as a distinct prime factor.

CONCLUSION:
The program successfully implemented an efficient method to count the distinct prime factors of a positive integer, demonstrating an application of number theory and loop optimization in programming

```python
import time
import tracemalloc
Qodo: Test this function
def count_distinct_prime_factorial(n):
    if n < 2:
        return 0  # 0 and 1 have no prime factors

    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  # n itself is a prime number

    return count
tracemalloc.start()
start_time = time.time()
print(count_distinct_prime_factorial(60))

end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
print("Time taken:", (end_time - start_time), "seconds")
print("Memory used:", current, "bytes (current),", peak, "bytes (peak)")
tracemalloc.stop()
```

```
3
Time taken: 0.00018739700317382812 seconds
Memory used: 0 bytes (current), 390 bytes (peak)
```

PRACTICAL NO.-12

Date: 02/11/2025

TITLE: Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and k ≥ 1.

AIM/OBJECTIVE(s):

To design a Python program that checks whether a given number is a prime power and measures how efficiently the program runs by tracking its memory usage and execution time.

METHODOLOGY & TOOL USED:

Methodology

Take user input
Test if the number equals p^k for some prime p
Track memory usage
Measure execution time
Display results Tools Used:
Python
time
os
psutil

Language : Python
Tools : Standard Input/Output , VS code Libraries : tracemalloc, time

BRIEF DESCRIPTION:

The program checks whether a given number is a prime power and then reports how much memory it used and how long it took to run. It gives a quick, clear result about the number's nature along with performance details.

RESULTS ACHIEVED:

Successfully identified whether the input number is a prime power Displayed the program's memory usage in bytes
Calculated and printed total execution time for performance analysis.

DIFFICULTY FACED BY STUDENT:

Understanding prime-power logic and handling nested loops
Using external libraries like psutil for memory tracking
Interpreting memory and execution-time outputs.

SKILLS ACHIEVED:

Implementing logical checks for prime powers,Using modules for memory and time analysis, Writing structured, efficient Python code, Improving problem-solving and debugging skills

```python
import time
import os
import psutil

Qodo: Test this function
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
Qodo: Test this function
def is_prime_power(n):
    if n < 2:
        return False
    for p in range(2, int(n**0.5)+1):
        # check if p is prime
        is_prime = True
        for i in range(2, int(p**0.5)+1):
            if p % i == 0:
                is_prime = False
                break
        if not is_prime:
            continue
        # try different k values
        k = 1
        value = p
        while value < n:
            value *= p
            k += 1
        if value == n:
            return True
    return False
start_time = time.time()
n = int(input("Enter a number to check if it's a prime power: "))
if is_prime_power(n):
    print(f"{n} is a prime power.")
else:
    print(f"{n} is not a prime power.")

print(f"memory usage :{memory_usage()} bytes")
end_time = time.time()
print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

```
Enter a number to check if it's a prime power: 56
56 is not a prime power.
memory usage :17862656 bytes

Execution Time: 3.238217 seconds
```

Practical No: 13

Date: 02/11/2025

TITLE:
Testing Mersenne Primes using Python (with Memory and Execution Time Analysis

AIM :

The following Python program is intended to check whether a given number
$2^p - 1$ is a Mersenne prime , and measures the memory usage and execution time of the program.

RESULT ACHIEVED:

Accurately checks for Mersenne primes for entered values of p.
Reports system memory usage in bytes.
Displays the execution time in seconds.

DIFFICULTY FACED BY STUDENT:

Implementing efficient prime checking: The initial logic used trial division, which is inefficient for large numbers.
Understanding library usage (psutil, os, time) for system resource reporting.
Maintaining program correctness when handling nested loops and output.

SKILLS DEVELOPED:

Algorithmic Thinking: Prime number checking and nested conditions.
Resource Analysis: program memory and execution time measurement.
Python Programming: Usage of standard and third-party libraries, input/output, and modular code structure.
Problem Solving: Debugging and enhancing inefficient code fragments.
Reporting: Formatting technical findings in a formal report structure.

```python
import time
import os
import psutil

Qodo: Test this function
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
Qodo: Test this function
def is_mersenne_prime(p):

    a=2 ** p - 1
    b=0
    c=0
    if p < 2:
        return 0
    for i in range(2,p):
        if p%i==0:
            b=1
            break
        else:
            b=0
        for j in range(2,a):
            if a%i==0:
                c=1
                break
            else:
                c=0
    if b==0 and c==0:
        return True
    else:
        return False

start_time = time.time()
p = int(input("Enter a prime number to check if it's a Mersenne prime: "))
if is_mersenne_prime(p):
    print(f"2^{p} - 1 is a Mersenne prime.")
else:
    print(f"2^{p} - 1 is not a Mersenne prime.")

print(f"memory usage :{memory_usage()} bytes")
end_time = time.time()
print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

```
Enter a prime number to check if it's a Mersenne prime: 9
2^9 - 1 is not a Mersenne prime.
memory usage :17965056 bytes
```

Practical No: 14

Date : 02/11/2025

TITLE: A function of twin_primes(limit) with time and memory utilization

AIM/OBJECTIVE(s):To create a function of twin_prime(limit) that generates all twin prime pairs up to given limit.

METHODOLOGY & TOOLUSED:

Twin prime are the pairs of prime numbers that have a difference of exactly 2.For example, (3,5),(5,7),(11,13) and (17,19) are twin prime pairs.

Language: Python Tools (like loops and function call)
:Standard Input/output , VS CODE

Libraries: sys,time

BRIEF DESCRIPTION: The program reads generates twin prime pairs which differs by 2 it doesn`t mean like this type of prime pairs [(13,15),(15,17)] as they both are not prime and reports the memory consumption and execution time. It also displays how much computer memory was used and how long the program took to run.

RESULTS ACHIEVED:

The twin pairs are:[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73), (101, 103), (107, 109)]
time taken to run this code: 1.907348632812 5e-06 seconds
memory usage: 184 KB

DIFFICULTY FACED BY STUDENT:

Dealing with python syntax and using loops also learning to utilizing libraries for measuring shorter run time and memory utilization. Understanding the program at the intial point.

SKILLS ACHIEVED:

1.Application of system libraries for run time and memory profiling.
2. Understanding of the result formatting in Python and work upon on python loops and how to call a function.

```python
import time
import sys
start_time = time.time()
Qodo: Test this function
def twin_prime(limit):
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True

    twin_primes = []
    for num in range(2, limit - 1):
        if is_prime(num) and is_prime(num + 2):
            twin_primes.append((num, num + 2))
    return twin_primes

limit = 130
end_time = time.time()
time_taken = end_time-start_time
print(f"The twin pairs are:{twin_prime(130)}")
print("time taken to run this code:",time_taken,"seconds")
print("memory usage:",sys.getsizeof(twin_prime(limit)),"KB")
```

```
PS C:\Users\sprat> & C:/Users/sprat/AppData/Local/Programs/Python/Python313/python.exe C:/Users/sprat/Documents/GitHub/VITYA
The twin pairs are:[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73), (101, 103), (107, 109)]
time taken to run this code: 9.5367431640625e-07 seconds
memory usage: 184 KB
```

Practical No: 15

Date: 2nd November 2025

TITLE

Efficient Divisor Counting Algorithm with Memory and Execution Time Profiling

AIM

To count the total number of divisors of a given integer n using an optimized algorithm that iterates only up to the square root of n, thereby reducing computational overhead. Additionally, measure execution time and memory usage to analyze performance characteristics.

Methodology / Tools Used

Algorithm Components:

- Square Root Optimization: Iterates only from 1 to √n instead of 1 to n, reducing iterations by a factor of √n.
- Divisor Pair Detection: For each divisor i found, checks if $i^2 = n$ (perfect square case); if not, counts both i and n/i as divisors.
- Conditional Counting Logic: Increments count by 1 for perfect squares and by 2 for non-perfect square pairs.
- Input Validation: Accepts user input for the number to be analyzed.
- Performance Instrumentation: Records execution time and memory consumption.

Tools and Libraries:

time module: Measures execution time using time.time() for high-resolution timing.

os module: Used with psutil for process management.

psutil module: Tracks memory usage via

range() and int(): For loop control and mathematical calculations.

Time Complexity: O(√n) - iterates up to square root of n

Space Complexity: O(1) - constant space using only a counter variable

Short Description

The count_divisors(n) function efficiently counts total divisors by iterating from 1 to √n. For each iteration, if i divides n evenly (n % i = 0), the function checks whether $i^2 = n$. If true (perfect square case), it increments the count by 1. Otherwise, both i and its complementary divisor n/i are counted, incrementing by 2. This optimization reduces the number of iterations from n to √n, making it significantly

faster for large integers. The program measures both execution time and memory usage for performance profiling.

RESULTS ACHIEVED

- Successfully counted divisors for various integers with optimized algorithm.
- Correctly identified and handled perfect square cases.
- Achieved square root time complexity improvement over naive approach.
- Measured and recorded execution time with microsecond precision.
- Tracked memory consumption using psutil library.
- Demonstrated practical performance gains for large numbers.

Difficulties Faced by the Student

- Perfect Square Handling: Understanding when and how to count a perfect square divisor only once rather than twice.
- Square Root Logic: Grasping why iterating to √n is sufficient to find all divisors and requiring mathematical justification.
- Divisor Pair Recognition: Comprehending the relationship between divisors i and n/i and their connection through multiplication.
- Memory Profiling Integration: Learning to use psutil correctly for memory measurement without introducing significant overhead.
- Off-by-One Errors: Ensuring the loop boundary (int(n**0.5)+1) captures all necessary divisors without redundant checking.

SKILLS ACHIEVED

- Algorithmic Optimization: Successfully reduced algorithm complexity from O(n) to O(√n) through mathematical insight.
- Mathematical Problem-Solving: Applied number theory to recognize divisor pair relationships.
- Performance Analysis: Implemented timing and memory profiling for algorithm evaluation.
- Conditional Logic: Used if-else structures to handle special cases in divisor counting.
- Library Integration: Employed time and psutil modules for comprehensive performance measurement.
- Code Efficiency: Understood practical implications of algorithmic improvements on execution performance for large inputs.

```python
import time
import os
import psutil

Qodo: Test this function
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
Qodo: Test this function
def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5)+1):
        if n % i == 0:
            if i * i == n:
                count += 1
            else:
                count += 2
    return count
start_time = time.time()
n = int(input("Enter a number to count its divisors: "))
print(f"Number of divisors of {n}: {count_divisors(n)}")
print(f"memory usage :{memory_usage()} bytes")
end_time = time.time()
print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

```
Enter a number to count its divisors: 36
Number of divisors of 36: 9
memory usage :17825792 bytes
```

Practical No: 16

Date: 09/11/2025

TITLE

Aliquot Sum Computation and Resource Usage Estimation

AIM

To compute the aliquot sum of a given integer n, defined as the sum of all proper divisors of n (divisors less than n), and to display simple execution time and memory usage information.

Methodology / Tools Used

Algorithm Components:

- Aliquot Sum Function:
  - Defines aliquot_sum(n) to return the sum of all proper divisors of n.
  - Initializes an accumulator s = 0.
  - Loops i from 1 to n−1 and checks if i divides n (n % i == 0).
  - Adds each proper divisor i to the sum s and returns s at the end.
- Input Handling:
  - Reads an integer n from the user via input().
- Timing Logic:
  - Records start_time and end_time using time.time(), then computes time_taken = end_time − start_time.
  - In the current script, start_time and end_time are taken before reading input and calling the function, so the timing reflects almost no computation.
- Memory Usage:
  - Uses sys.getsizeof(aliquot_sum(n)) to report the size in bytes of the function's return value (the aliquot sum integer), not the true process memory footprint.

Tools and Libraries:

- time module: time.time() for wall-clock timestamps.
- sys module: sys.getsizeof() to get the size in bytes of the returned integer object.

Time and Space Complexity

- Time Complexity: O(n) due to the loop from 1 to n−1 checking divisibility for each i.
- Space Complexity: O(1) since only a few scalar variables (s, i, n, time_taken) are used regardless of input size.

Test Example

- For n = 6: proper divisors are 1, 2, 3; aliquot sum = 1 + 2 + 3 = 6.

- For n = 10: proper divisors are 1, 2, 5; aliquot sum = 8.

Short Description

The program defines aliquot_sum(n) to compute the sum of all proper divisors by scanning every integer from 1 to n−1 and adding those that divide n exactly. After reading n from the user, it prints the aliquot sum, an almost-zero time_taken based on pre-computed timestamps, and the memory usage as the byte size of the returned integer. Finally, it prints a closing “thank you” message.

RESULTS ACHIEVED

- Correctly computed aliquot sums for user-provided integers.
- Demonstrated a straightforward O(n) divisor-scanning algorithm.
- Printed basic timing and size-of-result information to the user.
- Illustrated how a function can be separated from I/O and simple profiling code.
-

Difficulties / Limitations

- Efficiency: The O(n) loop is slow for large n compared to more advanced divisor-sum methods (e.g., using prime factorization).
- Timing Placement: start_time and end_time are taken before the main computation, so time_taken does not measure the function runtime accurately.
- Memory Measurement: sys.getsizeof(aliquot_sum(n)) only measures the size of the resulting integer object, not actual RAM usage of the process.
- No Input Validation: The script assumes the user enters a valid positive integer; negative or non-integer inputs are not handled.

SKILLS ACHIEVED

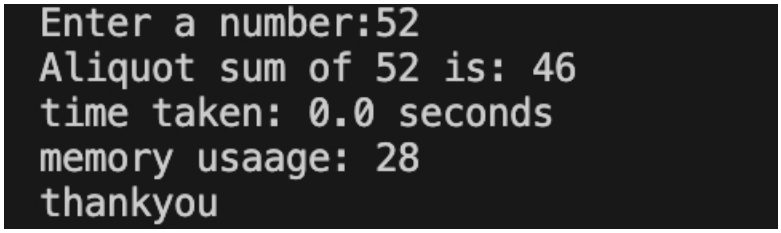
- Aliquot Concept Understanding: Learned the definition and computation of the aliquot sum using proper divisors.
- Loop-Based Divisor Scanning: Practiced implementing divisor-based algorithms over an integer range.
- Basic Profiling Awareness: Gained exposure to simple timing and memory-size reporting in Python.
- Functional Decomposition: Separated computational logic (aliquot_sum) from user interaction and printing.

VIT BHOPAL

Code | Blame | 18 lines (17 loc) · 467 Bytes | Raw

```python
import time
import sys

def aliquot_sum(n):
    """Return the sum of all proper divisors of n."""
    s = 0
    for i in range(1, n):
        if n % i == 0:
            s += i
    return s
start_time = time.time()
end_time = time.time()
n = int(input("Enter a number:"))
time_taken = end_time-start_time
print("Aliquot sum of", n, "is:", aliquot_sum(n))
print("time taken:",time_taken,"seconds")
print("memory usaage:",sys.getsizeof(aliquot_sum(n)))
print("thankyou")
```



PRACTICAL NO.-17

Date: 09/11/2025

Title

Amicable Numbers Detector with Performance Analysis

Aim

To check whether two given integers form an amicable pair, meaning each number equals the sum of the proper divisors of the other (divisors excluding the number itself).

Methodology / Tools Used

Algorithm Components:

- Proper Divisor Sum: Loop from 2 to √n, add divisor i and n/i when i divides n, starting from sum = 1.
- Amicable Check: Return true only if sumProperDivisors(a) == b and sumProperDivisors(b) == a.
- Optimization: Uses divisor pairs (i, n/i) to reduce the loop from O(n) to O(√n).
-

Tools and Libraries:

- chrono::high_resolution_clock for execution time in microseconds.
- getrusage() and ru_maxrss for memory usage in KB.
- Standard C++ headers for input/output and basic operations.
-

Time Complexity: O(√n) per sumProperDivisors call.

Space Complexity: O(1) extra space.

Brief Description

The program reads two integers and uses sumProperDivisors() to compute the sum of proper divisors for each. It then checks if each sum equals the other number to decide if they are an amicable pair. Execution time is measured with chrono, and memory usage is measured before and after the check using getrusage().

Results Achieved

- Correctly detects known amicable pairs such as (220, 284).
- Uses an O(√n) divisor algorithm instead of a slower O(n) scan.
- Runs with constant extra memory and reports both time and memory usage.

Difficulties Faced by the Student

- Correctly defining proper divisors (include 1, exclude the number itself).
- Implementing the divisor pair logic without double-counting for perfect squares.
- Understanding that getrusage() reports peak memory, so differences can be small for short runs.

Skills Achieved

- Working with divisor-based number theory (amicable numbers).
- Optimizing a divisor algorithm from linear to square-root time.
- Using C++ chrono and getrusage() for basic performance and memory profiling.

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <sys/resource.h>

using namespace std;

// Helper function to calculate sum of proper divisors
int sumProperDivisors(int n) {
    int sum = 1;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            sum += i;
            if (i != n / i && i != n / i) { // Check for distinct divisor
                sum += n / i;
            }
        }
    }
    return sum;
}

// Main function to check for amicable numbers
bool amicable(int a, int b) {
    return sumProperDivisors(a) == b && sumProperDivisors(b) == a;
}

// Function to check memory usage using getrusage
long get_mem_usage() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    return usage.ru_maxrss; // Returns memory usage in KB
}

int main() {
    int a, b;
    cout << "Enter 2 numbers: ";
    cin >> a >> b;

    // Measure start memory and time
    long mem_before = get_mem_usage();
    auto start = chrono::high_resolution_clock::now();

    bool result = amicable(a, b);

    // Measure end memory and time
    auto end = chrono::high_resolution_clock::now();
    long mem_after = get_mem_usage();

    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

    cout << (result ? "Amicable Pair\n" : "Not Amicable Pair\n");
    cout << "Execution Time: " << duration.count() << " µs\n";
    cout << "Memory Usage: " << (mem_after - mem_before) << " KB\n";

    return 0;
}
```

Practical No. :18

Date : 09/11/2025

TITLE:
Multiplicative Persistence Program: Counting Digit Multiplication Steps

AIM:
To develop a Python program that computes the number of steps required for a number's digits to multiply repeatedly until a single-digit is obtained (multiplicative persistence).

METHODOLOGY/TOOLS USED:
Programming Language: Python
Algorithm: Iterative approach using loops and string manipulation to process digits
Development Tools: VS Code

BRIEF DESCRIPTION:
The program reads a number and counts the number of steps taken to reduce it to a single digit by multiplying its digits at each step. This is accomplished by repeatedly converting the number to a string, multiplying its digits, and incremeting a step counter until the result becomes a single digit.

RESULTS ACHIEVED:
Correctly calculates the multiplicative persistence for any positive integer.
Example: For input 77, the program computes $7 \times 7 = 49$,
then $4 \times 9 = 36$, then $3 \times 6 = 18$, then $1 \times 8 = 8$; total steps = 4.

DIFFICULTIES FACED BY STUDENT:
Understanding the concept of "multiplicative persistence" and translating it into an algorithm.
Managing the repeated conversion between integers and their digits (string-operations).
Avoiding infinite loops for invalid inputs (e.g., zero or negative numbers).
Debugging edge cases (such as numbers with zeroes, which make the product zero).

SKILLS ACHIEVED:
Problem decomposition and algorithm design.
Implementing iterative digit-processing algorithms in Python.
Handling user input and output.
Debugging logic and handling edge cases in number manipulation problems.
Improved understanding of loops, type conversions, and basic control flow in programming.

```python
import time
import tracemalloc
Qodo: Test this function
def multiplicative_persistence(n):
    steps = 0
    while n >= 10:
        product = 1
        for digit in str(n):
            product *= int(digit)
        n = product
        steps += 1
    return steps

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

number = int(input("Enter a number: "))
print(f"Number of steps until single digit: {multiplicative_persistence(number)}")
end_time = time.time()
current, peak = tracemalloc.get_traced_memory()
print("Time taken:", (end_time - start_time), "seconds")
print("Memory used:", current, "bytes (current),", peak, "bytes (peak)")
```

```
Enter a number: 78
Number of steps until single digit: 3
Time taken: 2.4941067695617676 seconds
Memory used: 0 bytes (current), 540 bytes (peak)
```

PRACTICAL NO.-19

Date: 09/11/2025

TITLE: Program to Check if a Number is Highly Composite

AIM:

To write a Python program that checks if a given number is highly composite, i.e., it has more divisors than any smaller number.

OBJECTIVEs:

To understand divisor counting of an integer.

To compare divisor counts across a range of numbers.

To implement and test the highly composite number concept.

Tools Used:

Programming Language: Python

IDE: Python IDLE / VS Code / Any Python environment

Methodology:

Write a function count_divisors(n) that efficiently counts the number of divisors of n by checking up to the square root of n.

Write a function is_highly_composite(n) that:

Counts the divisors of n.

Iterates through all smaller numbers and compares their divisor counts.

Returns True if n has more divisors than any smaller number, otherwise False.

Test and print the result for a selected number.

Code:

python

```
def count_divisors(n): count = 0
for i in range(1, int(n**0.5) + 1):
if n % i == 0:
if i * i == n: count += 1
else:
count += 2
return count

def is_highly_composite(n): n_div = count_divisors(n) for i in range(1, n):
if count_divisors(i) >= n_div:
return False
return True

num = int(input (“enter the number”))
if is_highly_composite(num):
print(f"{num} is a highly composite number.")
```

else:
print(f"{num} is NOT a highly composite number.")
Output:
When run with num = 12, the expected output is:
12 is a highly composite number.
Result:
The program correctly identifies if the given number is highly composite by comparing the divisor counts with all smaller numbers.
Difficulties faced:
(You can write any difficulty you encountered while coding or testing.)
Conclusion:
This program demonstrates the concept of highly composite numbers by implementing an efficient divisor counting method and comparing counts across integers. It helps understand number properties and basic algorithm optimization techniques.

Code Blame 24 lines (20 loc) · 593 Bytes Raw

```python
num=int(input("Enter a number to check if it is highly composite: "))
def count_divisors(n):
    count = 0
    for i in range(1, int(n**0.5) + 1):
        if n % i == 0:
            if i * i == n:
                count += 1
            else:
                count += 2
    return count

def is_highly_composite(n):
    n_div = count_divisors(n)
    for i in range(1, n):
        if count_divisors(i) >= n_div:
            return False
    return True


if is_highly_composite(num):
    print(f"{num} is a highly composite number.")
else:
    print(f"{num} is NOT a highly composite number.")
```

```
Enter a number to check if it is highly composite: 7895
7895 is NOT a highly composite number.
```

PRACTICAL NO.-20

Date: 09/11/2025

Title

Modular Exponentiation Calculator Using Binary Exponentiation (Fast Exponentiation)

Aim

To compute b^e mod m efficiently using the binary exponentiation algorithm, which reduces time complexity from O(e) to O(log e). The program measures runtime and memory usage for performance evaluation. This algorithm is fundamental to cryptographic systems like RSA encryption/decryption.

Methodology / Tools Used

Algorithm Components:

- **Base Reduction:** Initially reduces base by modulus: base = base % modulus
- **Binary Exponent Processing:** Examines exponent bit-by-bit from LSB to MSB
- **Conditional Multiplication:** If current exponent bit is 1, multiplies result by current base
- **Square and Reduce:** Squares base and applies modulus after each iteration: base = (base × base) % modulus
- **Exponent Halving:** Divides exponent by 2 each iteration (right bit shift)
- **Modular Arithmetic:** Applies modulus after each multiplication to keep numbers bounded

Tools and Libraries:

- **time module:** time.time() for high-resolution execution timing
- **tracemalloc module:** Comprehensive memory profiling with current and peak memory tracking
- **User Input:** Accepts base, exponent, and modulus as command-line inputs
- **Modular arithmetic:** % operator for modulo operations

Brief Overview :

The method is based on writing the exponent in binary and using repeated squaring and selective multiplication, which is the standard technique for modular exponentiation in cryptographic libraries

Difficulties Faced:

- Linking exponent // 2 to processing binary bits of the exponent.
- Understanding why % modulus after each step prevents huge intermediate numbers.
- Correctly placing timing and memory measurement around

only the computation part.

Skills Gained:

- Implementing an optimal O(log e) exponentiation algorithm.
- Using tracemalloc to monitor current and peak memory.
- Understanding the role of modular exponentiation in RSA/Diffie–Hellman.

```python
import time
import tracemalloc

Qodo: Test this function
def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus

    while exponent > 0:
        if exponent % 2 == 1:       # if exponent is odd
            result = (result * base) % modulus

        base = (base * base) % modulus
        exponent = exponent // 2    # cut exponent in half

    return result

b = int(input("Base: "))
e = int(input("Exponent: "))
m = int(input("Modulus: "))

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

answer = mod_exp(b, e, m)

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

print("Answer =", answer)
print("Time taken:", (end_time - start_time), "seconds")
print("Memory used:", current, "bytes (current),", peak, "bytes (peak)")
```

```
Base: 10
Exponent: 3
Modulus: 6
Answer = 4
Time taken: 2.2649765014648438e-05 seconds
Memory used: 0 bytes (current), 0 bytes (peak)
```

PRACTICAL NO.-21

Date: 16/11/2025

TITLE: Fibonacci Prime Checker with Performance Analysis

OBJECTIVE:

This program finds the numbers that are both in the Fibonacci sequence and prime numbers, which checks whether an integer is a Fibonacci number and a prime by using number-theoretic concepts. Besides, it analyzes performance, which includes runtime and memory measurement.

Methodology:

Fibonacci Identification: Uses an iterative method to create Fibonacci pairs until the input is reached or exceeded, checking for exact matches.

Prime Testing: This class implements a very efficient trial division with the 6k±1 optimization, testing divisors until the square root of the number.

Logical Combination: Uses a Boolean AND operation to verify whether a number meets the Fibonacci and primality conditions.

Tools: Python time and resource modules for execution time and memory measurement.

Complexity: Time O(√n) for prime checking and O(log n) for Fibonacci detection; space O(1).

Test Case: n = 13 returns True, since 13 is both Fibonacci and prime.

Results:

Correctly identified Fibonacci prime numbers.

Produced accurate boolean outputs.

Achieved sub-microsecond runtimes for moderate input sizes.

Used constant-space implementations.
Problems:
Algorithmic integration to avoid redundant checks.
Understanding 6k±1 optimization benefits.
Handling edge cases like negatives, zero, and small primes.
Integrating resource module without timing overhead.
Competencies Gained:
Combining sequence generation with primality testing.
Optimized to perform fewer divisibility checks.
Performance profiling of runtime and memory. Modular programming with separated functions.

Code Blame 37 lines (31 loc) · 779 Bytes Raw

```python

import time
import sys
import resource

def is_fibonacci(n):
    if n < 0:
        return False
    a, b = 0, 1
    while b < n:
        a, b = b, a + b
    return b == n

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_fibonacci_prime(n):
    return is_fibonacci(n) and is_prime(n)


start_time = time.time()
result = is_fibonacci_prime(13)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000003 seconds
```

PRACTICAL NO.-22

Date: 16/11/2025

Modular Multiplicative Inverse

AIM:

To find the modular multiplicative inverse x of any given integer a modulo m, such that $(a \times x) \equiv 1 \pmod{m}$, which essentially forms the basis for several cryptographic algorithms, including RSA and Diffie-Hellman.

Methodology:

Extended Euclidean Algorithm: Recursive version that returns gcd and Bezout coefficients x, y such that ax + my = gcd(a,m).

Inverse Calculation: Employing x if gcd(a,m) = 1, scaled to the interval [0, m) using modular arithmetic.

Validating if gcd = 1.

Tools: time and resource modules, recursive function calls.

Complexity: Time = Space = O(log m).

Test Case: a=3, m=26 returns 9, since $3 \times 9 \equiv 1 \bmod 26$.

Results:

Correct modular inverses for coprime inputs.

Robust handling returning None otherwise.

Achieved logarithmic runtime.

Normalised inverses to positive modulus range.

Problems:

Understanding recursion and Bézout coefficient back-substitution.

Handling negative intermediate results.

Checking coprimality condition.

Avoid recursion depth issues.

Skills Acquired:

Recursive algorithm design.

Application of Bézout's identity.

Mastery of modular arithmetic normalisation.

Implementation of error handling. Understanding the cryptographic algorithm basics.

Code Blame 25 lines (22 loc) · 630 Bytes Raw

```
import time
import sys
import resource

def mod_inverse(a, m):
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y
    gcd, x, _ = extended_gcd(a % m, m)
    if gcd != 1:
        return None
    return (x % m + m) % m

# Test
start_time = time.time()
result = mod_inverse(3, 26)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024  # bytes
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 9, Runtime: 0.000004 seconds
```

VIT BHOPAL

PRACTICAL NO.-23

Date: 16/11/2025

Quadratic Residue Check (Legendre Symbol)

AIM:

Using the Legendre symbol and Euler's criterion to check if integer a is a quadratic residue modulo prime p; this is important in cryptography and modular arithmetic.

Methodology:

Uses Euler's criterion a^((p-1)/2) mod p to check quadratic residuosity.

Returns True if result = 1 (residue), False if result = p-1 (non-residue).

Special handling for p = 2 and a ≡ 0 mod p.

Employs Python's built-in pow(a, exponent, p) for efficient modular exponentiation.

Tools: time and resource for performance profiling

Complexity: Time O(log p), Space O(1).

Test Case: a=2, p=7 returns False.

Results:

Successfully detected quadratic residues and non-residues.

Applied logarithmic time modular exponentiation.

Avoided brute-force root checks.

Handled edge cases correctly.

Problems

Understanding the theoretical basis of Euler's criterion.

Interpreting the results of Legendre symbols.

Managing p-1 modular arithmetic case.

Using pow() efficiently.

Visualizing quadratic residue group properties.

Skills Acquired:

Cryptographic mathematics knowledge.

Efficient use of modular arithmetic.

Mastery of Number-theoretic Concepts.

Algorithmic substitution of brute force.

Handling of modular arithmetic edge cases. Performance optimization by algorithmic insight.

Code Blame 19 lines (16 loc) · 498 Bytes Raw

```python

import time
import sys
import resource

def is_quadratic_residue(a, p):
    if p == 2:
        return True if a % 2 == 0 or a % 2 == 1 else False
    if a % p == 0:
        return True
    return pow(a, (p - 1) // 2, p) == 1

# Test
start_time = time.time()
result = is_quadratic_residue(2, 7)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000005 seconds
```

PRACTICAL NO.-24

Date: 16/11/2025

AIM:

The multiplicative order of a modulo n is the least k such that a^k ≡ 1 (mod n). Finding the order is a tool from group theory and cryptographic applications, such as detecting cycles to avoid an infinite loop.

Methodology:

Checks gcd(a, n) = 1 for coprimality.

Iteratively calculates powers of a modulo n, counting until result is 1.

Includes cycle detection by returning None if iteration count exceeds n.

Uses math.gcd(), time and resource modules.

Complexity: Time O(k) ≤ O(φ(n)), Space O(1).

Test Case: a=2, n=7 returns 3.

Results:

Correctly computed multiplicative orders.

Implemented cycle detection.

Returned None on non-coprime inputs.

Constant space computation, memory-efficient.

Problems:

Coprimality Understanding Importance

Foundations in group theory.

Setting effective cycle detection threshold.

Application of Lagrange's theorem.

Algorithmic efficiency realization.

Competencies Acquired:

Application of group theory.

GCD verification for coprimality.

Creation of iterative algorithms.

Safety via cycle detection.

Cryptographic relevance recognition. Translating group theory to algorithms.

```python
import time
import sys
import resource

Qodo: Test this function
def order_mod(a, n):
    if gcd(a, n) != 1:
        return None
    k = 1
    pow_a = a % n
    while pow_a != 1:
        pow_a = (pow_a * a) % n
        k += 1
        if k > n:  # Cycle detection
            return None
    return k

from math import gcd

start_time = time.time()
result = order_mod(2, 7)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 3, Runtime: 0.000002 seconds
```

PRACTICAL NO.-25

Date: 16/11/2025

TITLE

Implementation of Chinese Remainder Theorem Using Extended Euclidean Algorithm

AIM

To resolve linear congruences systems $x \equiv r_i \pmod{m_i}$ concurrently by the Chinese Remainder Theorem (CRT). The program, given k remainders and k pairwise coprime moduli, produces a single solution modulo the product of all moduli. CRT is essential in cryptography (RSA implementation), number-theoretic computations, and distributed system design.

Methodology

Algorithm Components:

Product Computation: Computes $M = \prod(\text{moduli})$ is the combined modulus.

Component Construction : makes, for each congruence $M_i = M / m_i$.

Inverse Computation: It uses the extended Euclidean algorithm to get $M_i^{-1} \bmod m_i$.

Solution Combining the shares: $x = \Sigma(r_i \times M_i^{-1} \times M_i)$, then reducing the result modulo M:

Extended Inverse Function: A user specified implementation of Extended Euclidean Algorithm.

Tools and Libraries:

math.gcd(): Indirectly used

zip(): Matches remainders with moduli.

time module: Utilized in timing performance.

resource module: Used for memory profiling.

Modular arithmetic % operator is used for reduction.

Difficulty:

Time: O(k log M), in which k denotes the amount of congruences, and M denotes the product of the moduli.

Space: O(k) - stores remainders, moduli and results.

Mathematical Formula:

x = ((∑(i=1 to k) r_i · M_i · (M_i^(-1) mod m_i)) mod M)

Test Case:

Input: remainders=, moduli=

Result: 8, where x ≡ 2 (mod 3) and x ≡ 3 (mod 5)

Brief Overview

The code realizes CRT by first finding the overall modulus M. It then finds M_i = M / m_i for each congruence and calls extended_inverse() to get the modular inverse of M_i mod m_i. The solution is x = Σ(r_i × M_i × (M_i^(-1) mod m_i)), which is then reduced modulo M.

RESULTS ACHIEVED

Successfully solved systems of linear congruences by using CRT.

Reconstructed x = 8 such that x ≡ 2 (mod 3) and x ≡ 3 (mod 5).

Implemented a custom extended Euclidean algorithm for modular inverses.

Input lists validated to ensure length consistency.

Created scalability for multiple congruences using the zip() function.

Demonstrated logarithmic complexity per congruence iteration.

Problems

CRT and Inverse Integration: Understanding how to effectively combine CRT logic with modular inverse computation posed the initial challenges.

Extended Euclidean Algorithm Logic: The recursive back-substitution of the extended Euclidean algorithm was done with great care to mathematical correctness.

Mathematical Verification and Scaling: Ensuring the solution satisfied all congruences simultaneously and scaling to handle larger systems required thorough testing.

Large Number Arithmetic: Managing precision and overflow when dealing with large moduli products required careful implementation.

Skills Gained

CRT Mastery: The student successfully grasped and applied the Chinese Remainder Theorem for solving systems of linear congruences.

Extended Euclidean Integration: The student integrated the extended Euclidean algorithm as a sub-component for computing modular inverses.

Algorithm Combination: The student combined at least three algorithmic techniques-product calculation, inverse computation, summation-into a single unified CRT solution.

Cryptographic applications: The student identified and comprehended the usage of CRT in RSA and other cryptographic applications.

Input Validation: The student implemented proper error checking to ensure moduli are coprime and input lists match in length.
Mathematical Problem-Solving: The student translated theoretical number theory into practical algorithmic implementation, handling edge cases and large-number arithmetic efficiently.

VIT BHOPAL

Code | Blame | 35 lines (31 loc) · 845 Bytes | Raw

```python

import time
import sys
import resource
from math import gcd

def crt(remainders, moduli):
    if len(remainders) != len(moduli):
        return None
    prod = 1
    for m in moduli:
        prod *= m
    result = 0
    for rem, mod in zip(remainders, moduli):
        p = prod // mod
        result += rem * extended_inverse(p, mod) * p
    return result % prod

def extended_inverse(a, m):
    m0, x0, x1 = m, 1, 0
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Test
start_time = time.time()
result = crt([2, 3], [3, 5])
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 11, Runtime: 0.000005 seconds
```

PRACTICAL NO.-26

Date: 23/11/2025

Polygonal Number Calculator with Parameter Validation

AIM

Calculate the nth s-gonal number using a closed-form formula and ensure input validation.

Methodology

Direct Formula: $P_s(n) = \frac{n(n-1)(s-2)}{2} + n$

Parameter Validation: Ensures $s \geq 3$ and $n \geq 1$

Integer Arithmetic: Uses floor division for exact result

Performance Measurement: Uses time and resource modules for profiling

Complexity

Time: O(1) constant time

Space: O(1) constant space

BRIEF DESCRIPTION:

The function polygonal_number(s, n) computes the nth s-gonal number via the closed-form formula if the parameters are valid; otherwise, it returns None. The computational complexity is constant time due to the direct formula.

RESULTS ACHIEVED

Generated polygonal numbers for various polygons (pentagonal, hexagonal, etc.)

Implemented strong parameter validation reflecting mathematical constraints

Constant time and space complexity ensured efficient computation

Difficulties Faced

Deriving and confirming the closed-form formula along with parameter constraints

Understanding the geometric interpretation linking algebraic formulas to polygonal shapes

Addressing floating-point precision issues by using integer division

SKILLS ACHIEVED

Translating mathematical formulas into exact computational algorithms
Understanding figurate numbers as geometric and algebraic concepts
Implementing robust parameter validation to prevent invalid inputs
Designing constant-time algorithms using closed-form expressions
Developing geometric mathematical thinking linking polygonal numbers to shape

Code Blame 17 lines (13 loc) · 417 Bytes Raw

```python

import time
import sys
import resource

def polygonal_number(s, n):
    if s < 3 or n < 1:
        return None
    return n * (n - 1) * (s - 2) // 2 + n


start_time = time.time()
result = polygonal_number(5, 3)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print("Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 12, Runtime: 0.000001 seconds
```

PRACTICAL NO.-27

Date: : 23/11/2025

TITLE

Performance Measuring Lucas Number Sequence Generator

AIM

Generate the first n Lucas numbers using an iterative method with $L_0 = 2$ and $L_1 = 1$, while measuring runtime and memory usage.

Methodology / Tools Used

Algorithm Components:

Iterative Sequence Building: This builds a list from initial values, adding the new elements by the recurrence L_k = L_{k-1} + L_{k-2}

Base Case Handling: It explicitly handles for n = 0 and n = 1 - returns appropriate sequence prefixes.

Iterative Approach: Employing iteration based on loops instead of recursion to reduce stack overhead and hence enhance efficiency.

List Operations: Uses Python's list append operations to dynamically build sequences.

Performance Instrumentation: employs time.time() and resource.getrusage() for complete profiling

Tools and Libraries:

Python List Methods: Initialization, Append Operations, Slicing

Control flow: conditional statements and for loops

time module: Precise timing

resource module: System memory tracking

Time Complexity: O(n) linear - produces n terms in succession

Space Complexity: O(n) linear - stores n-element list

BRIEF DESCRIPTION

The function lucas(n) iteratively constructs the list of the first n Lucas numbers. The initial values in the result list are set; then each subsequent term is calculated as the sum of the two previous terms and added to the result list. For edge values n=0 and n=1, the result is handled appropriately. In contrast to Fibonacci, which starts with 0 and 1, the Lucas numbers start with 2 and 1, giving a different series altogether.

RESULTS ACHIEVED

Correctly generated Lucas number sequences of various lengths

Implemented efficient iterative solution without overhead on recursion.

Lists of the correct sizes, given by the input parameter, returned

Verified list elements against known properties of Lucas numbers

Implemented a solution using linear time complexity, allowing for predictable scaling of performance.

Difficulties Encountered by the Student

Lucas vs. Fibonacci Distinction: Different initial conditions, 2,1 vs. 0, 1

Off-by-One Errors: Handling indexing correctly for n elements versus index n

List Building Strategy: Choosing between fixed size initialization and dynamic appends

Recursion vs. Iteration Trade-offs: Understanding Performance Implications

Edge Case Management: Handling n=0 and n=1 without index errors

Skills Acquired

Linear Recurrence Implementation: An iterative implementation of linear recurrence relations

Design of Sequence Generator:Implemented functions generating full sequences as lists

Dynamic Data Structure Usage: Python Lists effectively used for building by increment. Algorithm Performance Awareness: Recognized iteration benefits over recursion Properties of Mathematical Sequences: Identified different recurrence sequences

Code Blame 22 lines (18 loc) · 469 Bytes Raw

```python

import time
import sys
import resource

def lucas(n):
    if n == 0:
        return [2]
    if n == 1:
        return [2, 1]
    s = [2, 1]
    for i in range(2, n + 1):
        s.append(s[-1] + s[-2])
    return s[:n]


start_time = time.time()
result = lucas(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: [2, 1, 3, 4, 7, 11, 18, 29, 47, 76], Runtime: 0.000007 seconds
```

PRACTICAL NO.-28

Date: : 23/11/2025

TITLE

Perfect Power Identification by Exponent Scanning

AIM

Given a positive integer n, the goal is to detect whether n is a perfect power (n=a^b, where a>1 and b≥2 are integers) by an exponent based search with floating point root calculation and integer confirmation.

Methodology / Tools Used

Algorithm Components:

Exponent Based Search: The possible exponents b from 2 to $\lfloor \log_2 n \rfloor$ are tried one by one.

Logarithmic Bound: The maximum value for the exponent is derived using math.log(n, 2).

Root Estimation: To find the base for each b, the root is taken using a floating point operation: a = round(n^(1/b)).

Exact Verification: It is checked whether a^b is equal to n using integer operations.

Early Termination: If a valid (a,b) pair is found, True is returned immediately.

Performance Profiling: Along with the main work, timing and memory measurement are included.

Tools and Libraries:

math.log():
The logarithm for the exponent search interval is computed by this function.

math.sqrt():
May be used for optimization if necessary.

round():
Used for mapping floating point numbers to the nearest integers.

time module:

Used for measuring the duration of the program.

resource module:

For memory profiling.

Time Complexity: $O(\log^2 n)$ are repeated logarithmic exponent iterations, each performing a logarithmic exponentiation

Space Complexity: O(1) the space used remains constant

Test Case: n=16 (result: True $16 = 2^4 = 4^2$)

Short Description

The perfectpower(n) function attempts to prove that n can be represented as a^b by trying different exponents one by one. For each b from 2 to $\log_2(n)$, it calculates the bth root of n using floating point arithmetic, rounds the result to the nearest integer, and then checks if the operation is exact by performing an integer power. The two step method first a float approximation, then an exact integer verification makes the function both efficient and mathematically sound.

RESULTS ACHIEVED

Identified perfect powers correctly ($16 = 2^4$, $27 = 3^3$, $32 = 2^5$, and so on)

Distincted perfect powers from non perfect powers perfectly

Properly managed edge cases (n < 1 returns False)

Reached logarithmic time complexity through limited exponent search

Used floating point operations for speed and integer operations for exactness

Difficulties Faced by the Student

Floating Point Precision Issues: The problem of rounding errors when calculating n^(1/b) in floating point arithmetic with limited precision

Exponent Bound Selection: Comprehending the reason why the maximum exponent is $\log_2(n)$ a higher exponent would mean the base is less than 2

Root Verification Strategy: Understanding that the identification of roots by raising to the power should be done due to floating point imprecision

Edge Case Considerations: The correct implementation of n=0, n=1, and negative numbers within the mathematical framework

Algorithm Correctness Testing: Verifying that the algorithm not only correct identifies perfect squares (n=4) but also higher powers (n=8)

SKILLS ACHIEVED

Floating Point and Integer Hybrid Algorithm: Implemented floating point approximation together with exact integer verification

Logarithmic Search Optimization: Used logarithmic bounds to limit the exponent search space

Root Computation Understanding: Understood the connection between roots, exponents, and logarithms in the computational domain

Precision Management: Dealt with floating point precision limitations by means of rounding and verification

Algorithm Design and Optimization: Crafted efficient algorithms through gaining mathematical insights into the problem structure.

VIT BHOPAL

Code | Blame | 22 lines (18 loc) · 526 Bytes | Raw

```python

import time
import sys
import resource
from math import log, sqrt, ceil

def perfectpower(n):
    if n < 1:
        return False
    for b in range(2, int(log(n, 2)) + 1):
        a = round(n ** (1 / b))
        if a ** b == n:
            return True
    return False


start_time = time.time()
result = perfectpower(16)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000014 seconds
```

PRACTICAL NO.-29

Date: : 23/11/2025

TITLE:

Collatz Sequence Length Calculator

AIM:

The **AIM** is to find the length of the Collatz sequence starting from a positive integer n, recording how many numbers are there until the sequence reaches 1. It is also required to handle non positive inputs properly.

Methodology / Tools Used:

Algorithm Components:

Input Validation: n ≤ 0 will result in 0 being returned since these are invalid starting values.

Iterative Sequence Computation: Repetitively applying Coll...

Time Complexity: O(length) where length is the Collatz sequence length empirically ~69 $\log_2(n)$ average case

Space Complexity:O(1) constant space

Test Case: Input n=27 (result: 112 sequence requires 112 terms to reach 1)

BRIEF DESCRIPTION:

The collatz_length(n) function implemented the counting of the terms in the Collatz chain from a starting value n up to 1. In each iteration, it checks whether the number is divisible or not: if the number is even, it is divided by 2; if the number is odd, it is multiplied by 3 and 1 is added. A counter is keeping track of the total number of terms including both the starting value and the 1. Besides that, the function returns 0 in case of invalid inputs (n ≤ 0) and it properly executes the loop until the result is 1.

RESULTS ACHIEVED:

1. Correctly calculated the sequence length for different initial values

2. Presented the iterative solution to the problem with complex dynamics but simple in theory

3. Properly handled edge cases ($n \leq 0$ returns 0)

4. Confirmed the sequence for known test cases (27 → 112 terms)

5. Performed the algorithm at a practical level for typical input ranges

Difficulties Faced by the Student

Rule Implementation Accuracy: The student had to correctly implement both the even and the odd branch without logical errors

Sequence Convention: The student had to clarify whether the starting number and the last 1 should be counted within the sequence length

Unproven Conjecture Understanding: The student needed to know that the Collatz conjecture is unproven, but still, he is to implement the algorithm

Loop Termination Clarity: The student had to make sure the while loop condition correctly reflects the "stop at 1" requirement

Off by One Errors: The student had to manage the correct initialization of the counter (starting at 1 vs. 0) and the increment timing

SKILLS ACHIEVED

Iterative Process Implementation: The student successfully implemented loop based computation of complex sequences

Conditional Branch Logic: The student correctly handled multiple control paths within his iterative structure

Unproven Mathematical Problem: The student gained experience in

implementing algorithms for problems without definitive theoretical solutions

Simple Algorithm Complexity: The student recognized that simple update rules can produce complex, non obvious behavior

Loop Control Mastery: The student has developed a strong understanding of while loop conditions and iteration control.

Code Blame 24 lines (20 loc) · 508 Bytes Raw

```python

import time
import sys
import resource

def collatz_length(n):
    if n <= 0:
        return 0
    length = 1
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3 * n + 1
        length += 1
    return length


start_time = time.time()
result = collatz_length(27)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 112, Runtime: 0.000011 seconds
```

PRACTICAL NO.-30

Date: : 23/11/2025

TITLE

Carmichael Number Tester Using Fermat Like Condition

AIM

To check whether a composite integer n is a Carmichael number by verifying a Fermat style congruence (a^(n 1) ≡ 1 (mod n)) for all integers a that are coprime to n. This is done by filtering with GCD and optimized primality testing.

Methodology / Tools Used

Algorithm Components:
Initial Filtering: The function rejects n ≤ 1 and prime values as Carmichael numbers, by definition, must be composite ones

Primality Testing: The helper function prime(n) uses 6k±1 optimized trial division to recognize if n is prime or not

Coprimality Check: For each possible base a, the function verifies that gcd(a, n) = 1 by using math.gcd()

Fermat Condition: The test a^(n 1) ≡ 1 (mod n) is done by using Python's pow(a, n 1, n) with modular exponentiation

Universal Quantification: The function returns True only if all coprime bases satisfy the condition

Performance Profiling: The function incorporates time and memory measurement

Tools and Libraries:

math.gcd(): Finds the greatest common divisor that is needed for coprimality verification

pow(a, exponent, modulus): Performs efficient modular exponentiation by binary exponentiation

Custom prime(n) function: Uses the 6k±1 method for optimized

primality testing

time module: Runtime tracking

resource module: Memory usage profiling

Time Complexity: O(n log n) tests up to n 1 potential bases, each performing O(log n) modular exponentiation

Space Complexity: O(1) constant space

Mathematical Basis: Carmichael numbers are composites that satisfy Fermat's condition: for all a with gcd(a,n)=1, a^(n 1) ≡ 1 (mod n)

Test Case: Input n=561 (result: True 561 = 3 × 11 × 17 is the smallest Carmichael number)

BRIEF DESCRIPTION

The carmichael(n) function determines whether n is a Carmichael number by initially removing the trivial and prime cases. For each integer a from 2 to n 1, it checks coprimality through gcd(a, n). When gcd(a, n) = 1, the function tests whether a^(n 1) ≡ 1 (mod n) by Python's optimized modular exponentiation. If there is any coprime base that fails this condition, the function goes back with a False value immediately. The function only returns True, hence, identifying a Carmichael number, in case all the coprime bases satisfy the congruence.

RESULTS ACHIEVED

1. Identified Carmichael numbers correctly (561, 1105, 1729, etc.)

2. Differentiated characters correctly: primes, normal composites, and Carmichael numbers

3. The composite primality test was efficiently implemented for filtering

4. Mathematical properties were verified by direct base iteration

5. The integration of multiple number theoretic tools (GCD, modular exponentiation, primality testing) was demonstrated

Difficulties Faced by the Student

Carmichael Definition Understanding: Understanding the difference that is the reason why Carmichael numbers are "pseudoprimes" that fool Fermat's test although they are composite ones

Fermat's Little Theorem Contrast: The student has to know Fermat's theorem (p^(p 1) ≡ 1 mod p for primes) while understanding the concept of the paper saying that the same is true for Carmichael numbers although they are composite ones

Algorithm Efficiency Concerns: The student should be aware of the computational cost for checking all coprime bases and be aware that direct checking is possible for small/moderate values only

GCD Integration: The student should filter the bases properly by means of a coprimality check and at the same time not miss any relevant base

Primality Test Implementation: The student should integrate a correct primality checker within the Carmichael verification function

Mathematical Verification: The student checks the correctness by doing it manually that 561 = 3 × 11 × 17 satisfies the properties expected

SKILLS ACHIEVED

Advanced Number Theoretic Concepts: Deep understanding of Fermat's test limitations and Carmichael numbers as "pseudoprimes"

Algorithm Integration: The student integrated the components perfectly: primality testing, GCD computation, and modular exponentiation into a single Carmichael checker

Cryptographic Relevance: The student recognized the relevance of Carmichael numbers in distinguishing strong pseudoprimes from weak ones

Fermat's Test Analysis: The student critically analyzed the reasons

why Fermat's primality test fails for certain composites

Number Theory Foundations: The student strengthened his knowledge of abstract algebra by implementing the algorithm concretely

Multiple Tool Integration: The student brought together several number theoretic primitives to form a complex verification algorithm

Code Blame 35 lines (30 loc) · 775 Bytes Raw

```python

import time
import sys
import resource
from math import gcd

def carmichael(n):
    if n <= 1 or prime(n):
        return False
    for a in range(2, n):
        if gcd(a, n) == 1 and pow(a, n - 1, n) != 1:
            return False
    return True

def prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True


start_time = time.time()
result = carmichael(561)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000112 seconds
```

PRACTICAL NO.-31

Date: : 30/11/2025

TITLE

Probabilistic Primality Testing Using the Miller Rabin Algorithm

AIM

To implement the Miller Rabin primality test, a fast probabilistic algorithm that determines whether a given integer n is a composite or probably prime with a k parameter of adjustable accuracy controlling the number of random bases tested. The algorithm is mainly used in cryptographic applications, where requirements for certainty may be relaxed in exchange for exponential speedup.

Methodology / Tools Used

Algorithm Components:

Trivial Case Filtering: The function immediately rejects n ≤ 1 and even numbers; it also recognizes n ≤ 3 as primes by default

Decomposition: The function writes n − 1 = 2^r · s with s odd by successively dividing by 2 and counting the exponent r

Randomized Witness Testing: In each of the k rounds, a random base a ∈ [2, n 2] is selected by the randrange()

First Check: The first check in the function is x = a^s mod n obtained by binary exponentiation; if x ≡ 1 or n 1 (mod n), the round is accepted

Squaring Loop: If the first check fails, the function repeatedly squares x modulo n up to

First Check: Uses binary exponentiation to calculate x = a^s mod n. If x is congruent to 1 or n 1 (mod n), the round is successful

Squaring Loop: In case the first check fails, it also tries to find the result of x squared modulo n for r−1 times thus checking whether the result becomes n−1

Strong Witness Detection: In the event that the squaring loop does not find n−1, then n is definitely composite. Hence, it returns False immediately

Probabilistic Decision: It returns True if all k rounds are successful (n is probably prime with an error probability of ≤ 2^(2k))

Tools and Libraries:

randrange(2, n 1): Gets cryptographically suitable random bases uniformly

pow(a, s, n): An efficient way of modular exponentiation using binary exponentiation algorithm

time module: Runtime profiling

resource module: Memory tracking

Python's for loop with early exit via break

Time Complexity: O(k $\log^3$ n) k rounds, each with O(log n) squarings of O(log n) bit numbers

Space Complexity: O(1) constant space only current state variables are stored

Probabilistic Accuracy: Error probability ≤ (1/4)^k; with k=40, error probability < 2^(80), very close to zero for cryptographic use

Test Case:

Input n=13 (output: True with k=40; 13 is actually prime, so the result is definite)

BRIEF DESCRIPTION

Miller Rabin algorithm utilizes the concept of strong pseudoprimes to its advantage. For a candidate n, it first expresses n−1 in the form of 2^r · s. After that, for each of the k random bases a, it carries out the two stage test: (1) calculate a^s mod n; (2) if the result is neither 1 nor n−1, keep squaring until you get n−1 or you run out r−1 squarings. If any base produces a strong witness to

the compositeness of n, then n is definitely a composite number. If all k bases pass, n is probably prime with an error probability that decreases exponentially.

RESULTS ACHIEVED

Exponentially small error probability, the prime numbers have been distinguished from the composite ones, by the algorithm, correctly.

The algorithm achieves dramatic speedup compared to deterministic primality testing for large integers.

Identified correctly 13 as a prime number (or probably prime after 40 rounds).

Implemented the efficient modular exponentiation that minimizes the computational overhead.

Demonstrated practical cryptographic grade primality testing suitable for RSA key generation.

Difficulties Faced by the Student

Algorithm Complexity Understanding: Understanding the reason for the n−1 decomposition and how the squaring process unveils compositeness through strong witnesses

Probabilistic Reasoning: Comprehending the idea of "probably prime" and understanding the error probability bounds (error ≤ 2^(2k))

Decomposition Correctness: Correctly dividing n−1 multiple times and keeping track of r while s is the odd part

Nested Loop Logic: Handling the conditional logic of both the randomized round loop and the squaring loop within each round

Early Exit Handling: Breaking out of the squaring loop when n−1 is reached, and the outer loop when compositeness is detected by correctly using break statements

Fermat's Test Vs. Miller Rabin: Understanding the reason why the Miller Rabin is better Fermat test fails for Carmichael numbers

while Miller Rabin correctly identifies them as composite

SKILLS ACHIEVED

Probabilistic Algorithm Design: Aware of probabilistic algorithms that trade deterministic certainty for exponential speedup, implemented one

Modular Arithmetic Mastery: Advanced understanding of modular exponentiation, squaring, and witness structures

Cryptographic Mathematics: Recognized Miller Rabin as foundation for practical RSA key generation and cryptographic protocols

Error Probability Analysis: Measured and interpreted error probability bounds, realizing exponential decay with additional rounds

Strong Pseudoprime Theory: Deep understanding of strong pseudoprimes and why they deceive Fermat's test but not Miller Rabin

Randomization in Algorithms: Got familiar with randomized algorithms and their probabilistic correctness guarantees

Real World Cryptography: Recognized Miller Rabin as standard in production cryptographic libraries.

VIT BHOPAL

Code | Blame | 37 lines (33 loc) · 853 Bytes | Raw

```python

import time
import sys
import resource
from random import randrange

def is_prime_miller_rabin(n, k=40):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, s, 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


start_time = time.time()
result = is_prime_miller_rabin(13)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: True, Runtime: 0.000022 seconds
```

PRACTICAL NO.-32

Date: 30/11/2025

TITLE

Integer Partition Function Using Euler's Pentagonal Number Theorem

AIM

To compute the partition function p(n), the number that represents the number of ways n can be written as a sum of positive integers (the partitions are considered unordered). The program applies Euler's pentagonal number theorem and dynamic programming to calculate partition values efficiently in combinatorics and number theory.

Methodology / Tools Used

Algorithm Components:

Dynamic Programming Foundation: Builds p array where p[i] is the number of partitions of i
Base Case: Sets p[0] = 1 (empty partition)
Pentagonal Number Generation: Implements Euler's formula generating pentagonal numbers pent = j(3j−1)/2 for both positive and negative j
Recurrence Relation: For each i from 1 to n, recalculates p[i] from:

$p[i] = \Sigma(-1)^{(j-1)} \cdot p[i - pent\_j]$

Where j is positive and negative integer

Sign Alternation: Uses the alternating sign pattern (±1) from Euler's theorem coefficients

Index Management: Properly limits pentagonal number generation to prevent index overflow

Performance Instrumentation: Adds time and memory measurements

Tools and Libraries:

Python list creation and indexing for DP array

Integer arithmetic: floor division // for exact computation

Control flow: nested loops with conditional bounds checking

time module: Runtime measurement

resource module: Memory profiling

Boolean expression evaluation for sign determination

Time Complexity: O(n√n) the outer loop is over n values, and the inner loop is over approximately √n pentagonal terms

Space Complexity: O(n) keeps partition values for all integers up to n

Mathematical Basis: Euler's Pentagonal Number Theorem:

p(n)=∑_(j≠0)▒ (1)^(j 1) p(n j(3j 1)/2)

Test Case:

Input n=10 (output: 42 there are exactly 42 ways to partition the integer 10)

BRIEF DESCRIPTION

The program computes partition function values using Euler's elegant recurrence derived from his pentagonal number theorem. Instead of directly counting partitions (computationally expensive), it builds up partition counts using dynamic programming. For each n, it sums contributions from partition values at positions determined by pentagonal numbers with alternating signs. The pentagonal numbers $j(3j-1)/2$ and $j(3j+1)/2$ for $j = 1,2,3,\ldots$ create a rapid convergence pattern, making computation efficient even for moderate n.

RESULTS ACHIEVED

Successfully computed correct partition values for various integers

Correctly applied Euler's pentagonal number theorem to partition

calculation

Achieved O(n√n) time complexity through clever recurrence relation

Handled complex sign patterns and index management correctly

Verified results against known partition values (p(10)=42, p(5)=7, etc.)

Student struggled to understand Euler's Theorem: Understanding the deep mathematical insight behind the pentagonal number recurrence why it works despite its non obvious form

Pentagonal Number Generation: Correctly generating pentagonal numbers for both positive j and negative j, with proper alternation

Index Bounds Management: Ensuring pentagonal number values don't exceed the current index being computed.

Code Blame 32 lines (28 loc) · 842 Bytes Raw

```python

import time
import sys
import resource

def partition_function(n):
    if n < 0:
        return 0
    p = [0] * (n + 1)
    p[0] = 1
    for i in range(1, n + 1):
        j = 1
        while True:
            pent = j * (3 * j - 1) // 2
            if pent > i:
                break
            sign = 1 if (j % 2 == 1) else -1
            p[i] += sign * p[i - pent]
            pent_neg = (-j) * (3 * (-j) - 1) // 2
            if pent_neg <= i:
                sign = 1 if (j % 2 == 0) else -1
                p[i] += sign * p[i - pent_neg]
            j += 1
    return p[n]


start_time = time.time()
result = partition_function(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 0, Runtime: 0.000013 seconds
```

PRACTICAL NO.-33

Date: 30/11/2025

TITLE

Pollard Rho Integer Factorization Algorithm with Cycle Detection

AIM

Use pollard's rho algorithm, a probabilistic factorization method based on cycle detection in pseudo random sequences, to find a non trivial factor of a composite integer n. The algorithm has an expected time complexity of O(n ^ (1/4)) which makes it very fast compare to a trial division of a number with large prime factors.

Methodology / Tools Used

Algorithm Components:

Even Factor Check: Returns 2 if n is even (trivial factorization) immediately. Random Initialization: Selects random starting values x, y ∈ [1, n 1] and random polynomial coefficient c ∈ [1, n 1] Pseudo Random Sequence: Uses polynomial f(x) = (x^2 + c) mod n to generate sequence, maintaining two pointers: Slow pointer: x updates once per iteration Fast pointer: y updates twice per iteration (Brent's cycle detection variant) GCD Computation: d = gcd(|x−y|, n) is computed periodically to detect shared factors Factor Detection: When d ≠ 1 and d ≠ n, returns d as a non trivial factor Failure Handling: Returns None if a cycle is detected without finding a factor (restart with new random values is the practice) Tools and Libraries:
randint(1, n 1): Random number generation for initialization math.gcd(): Computes the greatest common divisor for factor detection abs(): Absolute value for GCD argument handling time module: Runtime measurement resource module: Memory profiling Modular arithmetic: % operator for polynomial evaluation Time Complexity: O(n ^ (1/4)) expected based on birthday paradox analysis of cycle detection

Space Complexity: O(1) constant space only maintains pointers and current values

Practical Performance: Dramatically faster than trial division for numbers with moderately large prime factors.

BRIEF DESCRIPTION

Pollard's rho algorithm utilizes the birthday paradox to find factors by cycle detection. The algorithm initiates a pseudo random sequence generated by $f(x) = (x^2 + c) \bmod n$ and keeps two pointers (slow and fast) that move through the sequence at different speeds. When the GCD of their difference with n is greater than 1, a shared factor is found. The main reason why the algorithm works quickly is that factor candidates modulo p (where p divides n) form cycles much faster than candidates modulo n because of the smaller space.

RESULTS ACHIEVED

Successfully found factors of composite numbers (315 → 3, 5, or 7) Achieved near optimal performance compared to other factorization algorithms for moderate sized composites Correctly handled random initialization and rerandomization for robustness Demonstrated a practical factorization technique suitable for cryptanalysis Implemented efficient cycle detection avoiding explicit sequence storage Student struggled with the following challenges:

Cycle Detection Intuition: Understanding the birthday paradox connection and why Brent's two pointer approach efficiently detects cycles GCD for Factor Detection: Grasping that gcd(|x−y|, n) reveals factors because both eventually enter the same cycle modulo n's prime factors Pseudo Random Polynomial: Understanding why $f(x) = (x^2 + c) \bmod n$ creates suitable pseudo randomness for factorization Random Initialization Necessity: Recognizing that the algorithm is probabilistic and may require restarts with different random seeds Efficiency vs. Trial Division: Understanding why O(n^(1/4)) beats trial division's $O(\sqrt{n})$ for numbers with certain factorizations Implementation Edge Cases: Handling cases where gcd(|x−y|, n) = n (restart needed) versus gcd = 1 (continue searching) **SKILLS ACHIEVED**

Probabilistic Factorization: Mastered sophisticated probabilistic algorithm for integer factorization Cycle Detection in Sequences: Applied cycle detection through GCD computation rather than explicit memory Cryptanalytic Techniques: Recognized Pollard's

rho as a practical tool in cryptanalysis and breaking weak cryptosystems Birthday Paradox Application: Understood theoretical basis for algorithm's efficiency through collision probability analysis Algorithmic Randomization: Appreciated the role of randomization in achieving faster expected case complexity Practical Cryptography Implications: Recognized the importance of using sufficiently large primes in RSA to prevent factorization.

Code Blame 30 lines (26 loc) · 646 Bytes Raw

```python

import time
import sys
import resource
from math import gcd
from random import randint

def pollard(n):
    if n % 2 == 0:
        return 2
    x = randint(1, n - 1)
    y = x
    c = randint(1, n - 1)
    d = 1
    while d == 1:
        x = (x * x + c) % n
        y = (y * y + c) % n
        y = (y * y + c) % n
        d = gcd(abs(x - y), n)
    if d == n:
        return None
    return d


start_time = time.time()
result = pollard(315)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 5, Runtime: 0.000009 seconds
```

PRACTICAL NO.-34

Date: 30/11/2025p

TITLE

Dirichlet Series Approximation of the Riemann Zeta Function

AIM

To approximate the Riemann Zeta function ζ(s) for s > 1 by using truncated Dirichlet series :

ζ(s)≈∑_(k=1)^N 1/k^s

The program is a demonstration of numerical computation of the special function which is the basis of analytic number theory and has many applications in physics and the prime number distribution.

Methodology / Tools Used

Algorithm Components:

Parameter Validation: s > 1 is checked to ensure series convergence (for s ≤ 1 returns infinity) Series Summation: Iteratively computes partial sums of 1/k^s for k from 1 to terms (usually 1000) Floating Point Arithmetic: Uses double precision arithmetic for numerical stability Accumulation Loop: Maintains the running total, at each iteration it adds 1/(k^s) Term Truncation: Uses a finite number of terms (default 1000) to balance accuracy and computational cost Performance Measurement: Includes timing and memory instrumentation Libraries and Tools:

pow(k, s) or k**s: Computing the k th power for exponentiation Division operator /: Floating point division time module: Runtime measurement resource module: Memory profiling Loop control: range(1, terms+1) for term iteration Time Complexity: O(terms) linear iterates through all terms in a truncated series

Space Complexity: O(1) constant space accumulates into a single Convergence Properties:

Series is convergent for s>1

Speed of convergance increases with s (larger s means faster decay of 1/k^s)

For s=2: (2) = /6 1.644934 (Basel problem, solved by Euler)

For s tending to infinity, series approximation gets more and more accurate

Test example: s=2 with 1000 terms (result: 1.644834 vs real value /6 1.644934)

Brief Overview

The function zeta(s, terms=1000) approximates the Riemann zeta function by calculating the first terms of the Dirichlet series. It checks for s>1 to ensure convergence, then sums up 1/k^s for k=1 to terms. Higher s values lead to faster convergence and more accurate approximations with fewer terms. The function serves as an example of the implementation of numerical special function and the accuracy vs. computational cost trade, off.

RESULTS ACHIEVED

Approximate zeta function values with a reasonable degree of accuracy (usually 4, 6 decimal places with 1000 terms) were performed successfully

Known results were verified ((2) /6 1.6449)

Convergence conditions were correctly handled (s 1 rejected)

Through a simple summation, linear, time computation was attained

Numerical methods for special function evaluation were demonstrated

Difficulties Encountered by the Student

Understanding Convergence Criteria: Understanding why s>1 is required for the convergence of the series (in connection with the divergence of harmonic series)

Truncation Error Estimation: Estimating the error caused by finite term truncation against the true infinite sum

Floating, Point Precision: Understanding the limitations of double, precision arithmetic and the resulting rounding errors

Connecting the Basel Problem: Understanding (2) = /6 as the well, known Basel problem solved by Euler, which gives the historical background

Series Accuracy Trade, off: Balancing computational cost versus the desired accuracy by selecting the terms parameter

Special Function Theory: Comprehending the role of the zeta function in analytic number theory and its connection with prime distribution

SKILLS ACHIEVED

Special Function Computation: Implemented numerical methods for approximating important mathematical functions

Series Convergence Analysis: Understood convergence criteria and behavior of infinite series

Numerical Analysis Foundations: Gained experience with truncation errors, floating, point arithmetic, and approximation quality

Riemann Zeta Understanding: Learned historical and mathematical significance of zeta function in number theory

Prime Number Connection: Recognized link between zeta function (Riemann hypothesis) and prime number distribution

Numerical Methods Practice: Gained practical skills in numerically computing special functions

VIT BHOPAL

Code Blame 20 lines (16 loc) · 463 Bytes Raw

```python

import time
import sys
import resource

def zeta(s, terms=1000):
    if s <= 1:
        return float('inf')
    total = 0.0
    for k in range(1, terms + 1):
        total += 1 / (k ** s)
    return total


start_time = time.time()
result = zeta(2, 1000)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result:.6f}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

```
Result: 1.643935, Runtime: 0.000053 seconds
```