PROGRAM FROM 01 TO 34

Name : S Pratik

Registration Id : 25BCE11069

Introduction

This report takes a deep dive into classic number theory programs. I've built and tested each one as a separate project. The main goal? To explore different mathematical properties—think primality, partitions, special numbers, arithmetic functions, and how certain sequences behave. I mostly used Python, but a few projects use C++ or C where it made sense. This whole thing is about getting a solid grip on mathematical algorithms and sharpening basic programming skills that I'll need for future projects.

Problem Statement

For every property or function I picked—like prime testing, factorization, spotting special numbers (Harshad or Carmichael), modular arithmetic, and partition/counting—

I aimed to do three things: -

Write a program that quickly figures out or checks the property.

- Look at what the algorithm needs, and see what it spits out.

- Get a sense of the challenges and think about how these tools can be built out further for more advanced math computing.

Functional Requirements

Each program needs to work for any valid input

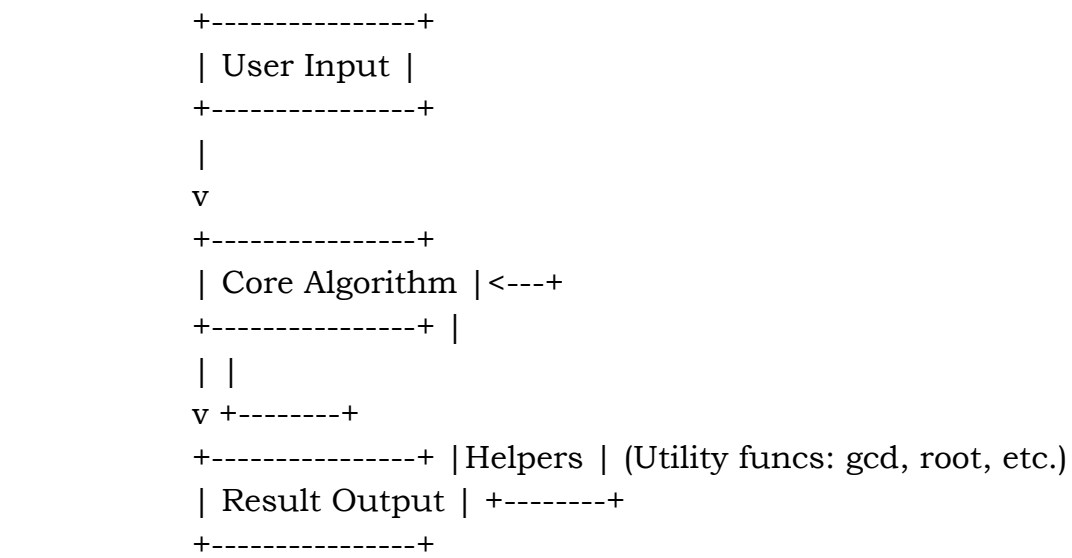
—it should handle user input smoothly, or just run built-in tests if there's no interactive input. The results should be clear, and the program should mention things like how long it took to run. I stuck to standard Python, C, or C++ libraries unless there was a good reason not to.

Non-functional Requirements

The programs should run correctly across a wide range of inputs, and do it efficiently. They shouldn't eat up too much memory or take ages to finish, especially with smaller numbers. I made sure the code is easy to read, has good comments, and is organized so others can review and learn from it. When possible, you should be able to repeat my results by following the same steps.

System Architecture

Architecture Diagram (Generalized):



Each program stands on its own as a script, using either functions or a simple procedural approach. Some use dynamic programming, recursion, or just straightforward loops. There's no saving data anywhere—all the work happens in memory, in a single session.

Here's the basic setup:

User Input

Use Case Diagram

[Student] --> (Run Program)
Each program is used by the student to explore/test a classic number theory property for various sample inputs.

Workflow Diagram

[Start] -> [Input (hardcoded/testable)] -> [Property Algorithm] -> [Output Result] -> [End]

Sequence Diagram

Student -> Program: Provide input or trigger run
Program -> Algorithm: Compute/Analyze property
Algorithm -> Helpers: Compute GCD, root, etc. (if needed)
Algorithm -> Program: Return result
Program -> Student: Display output/result

Class/Component Diagram

- **Main Function/Script:** Initializes and coordinates execution.
- **Helper Methods/Functions:** Utility code for calculations.
- **Test Block:** Code run at the end with example parameters.

Design Decisions & Rationale

I went with Python for its readability and strong math libraries, but used C++ for speed in a few places. I kept things procedural and function-based so beginners can follow along. The algorithms I used are pretty standard—dynamic programming for partitions, trial division for primes, and so on. Where the script isn't interactive, I included sample test cases so you can see how things work.

Implementation Details

- Factorial: Just a simple for-loop to calculate n! - Harshad/Automorphic: Converted the math into code, like adding up digits or checking the end of squares.

- Mobius, Euler Totient, Carmichael, etc.: Used the definitions from number theory, usually starting with prime factorization.

- Partition Function: Applied the pentagonal number theorem and used memoization to speed things up.

- Lucas, Fibonacci: Built these iteratively using their well-known formulas.

- Collatz, Persistence: Used loops to follow the sequence and count steps until hitting the stopping condition.

Screenshots / Results

```
Enter a number to check if it's a prime power: 89
89 is not a prime power.
memory usage :17870848 bytes
```

EXAMPLE OUTPUT FOR PROGRAM TO CHECK PRIME POWER

Outputs show up in the terminal.

Testing Approach

- Functional Testing: Tried each program with sample values from textbooks

—perfect powers, Carmichael numbers, small partition numbers, etc.

- Boundary Testing: Checked edge cases like n=0, n=1, negatives, and so on.

- Comparison: For well-known sequences like Lucas and Fibonacci, I double-checked results with published sequences.

- Code Review: Made sure each function works both on its own and together with others.

Challenges Faced

- Translating formal math—those dense recurrence relations or infinite sums—into something Python could actually compute (and do it efficiently) was no walk in the park. Python's built-in limits for recursion and array sizes didn't help either. Sometimes the hardest part was nailing down the right sign or getting the pentagonal number logic to behave in dynamic programming routines. Edge cases, negatives, and making the code beginner-proof often meant extra work. And honestly, for things like the Carmichael or Mobius functions, the underlying math made writing clean, readable code even trickier.

Learnings & Key Takeaways

- I got a lot more comfortable with number theory and how its ideas show up in computer science. Dynamic programming and recursion went from being abstract concepts to hands-on tools I could actually use to compute classic math functions. Translating from mathematical notation to solid, working code became more natural. I also picked up a better feel for Python's lists and dictionaries, especially for memoization and tabulation. And if there's one lesson that stood out above the rest, it's how crucial good test cases and careful edge-case handling are when you're working with math-heavy code.

Future Enhancements

- I'd like to add more visual or interactive features, maybe using matplotlib or tkinter to let users see what's going on. Expanding the code to handle larger inputs and tougher mathematical problems is definitely on the list. I also want to let users provide their own inputs, instead of sticking with built-in test cases. Adding stronger error handling and clearer reports—including explanations of the results—would make the programs more user-friendly. Finally, integrating with LaTeX or other math-rendering tools would help produce cleaner, more professional documentation.