Ultimate Number Theory Toolkit in Python

Cover Page

Project Title: Ultimate Number Theory Toolkit in Python

Subject: Python Essentials

Submitted By: Aniruddha Majumdar

Date: 23/11/25

Introduction

This project is a comprehensive collection of computational algorithms designed to investigate properties of integers, generating mathematical sequences, and performing cryptographic arithmetic. The suite is implemented in Python and places a strong emphasis on performance analysis.

Each module within the system is instrumented with `time` and `tracemalloc` libraries to provide real-time feedback on execution speed and memory consumption. This allows for an empirical analysis of algorithmic efficiency, making the toolset valuable not just for solving mathematical problems, but for understanding the computational cost of those solutions.

Problem Statement

The Core Problem: Mathematical computations involving large integers—such as primality testing, factorization, and modular arithmetic—can be computationally expensive. Students and developers often need a unified toolkit to verify number properties (e.g., "Is N a prime?", "Is N an Armstrong number?") without writing boilerplate code for performance tracking.

Goal: Develop a modular Python-based system that:

1. Solves specific number-theoretic problems.
2. Automatically profiles the solution for Time Complexity (Execution Time) and Space Complexity (Peak Memory Usage).

Functional Requirements

The system must provide the following distinct computational modules:

- **Basic Number Properties:** Check if a number is Palindrome, Harshad, Automorphic, Pronic, Abundant, Deficient, or a Perfect Power.
- **Factorization & Primes:** Compute Factorials, Prime Factors, Distinct Prime Factors, and count Divisors.
- **Primality Testing:** Deterministic checks and probabilistic tests (Miller-Rabin) for large numbers.
- **Sequences:** Generate Fibonacci numbers, Lucas sequences, and Polygonal numbers.
- **Modular Arithmetic:** Perform Modular Exponentiation (`mod_exp`), Modular Inverse (`mod_inverse`), and Chinese Remainder Theorem (`CRT`) calculations.
- **Cryptographic Primitives:** Implement Pollard's Rho algorithm for integer factorization and check for Quadratic Residues.
- **Analysis:** Calculate Multiplicative Persistence and Partition Functions.

Non-functional Requirements

- **Performance:** Algorithms must run efficiently. For example, primality tests should use `O(sqrt(N))` complexity or better (Miller-Rabin) rather than `O(N)` .

- **Resource Monitoring:** Every function execution must report:
    - Execution time in seconds.
    - Peak memory usage in bytes/KB.
- **Usability:** The interface is Command Line (CLI) based, prompting users for simple integer inputs.
- **Accuracy:** Mathematical results (e.g., `is_prime`) must be 100% accurate for deterministic algorithms and statistically significant for probabilistic ones.

System Architecture

The system follows a **Modular Scripting Architecture**. There is no central database; the state is transient.

- **Input Layer:** `input()` functions capture user data.
- **Processing Layer:** Distinct Python functions (e.g., `is_harshad` , `pollard_rho`) contain the logic.
- **Profiling Layer:** Wrappers using `time.time()` and `tracemalloc` surround the logic.
- **Output Layer:** `print()` statements display the boolean result/value and the performance metrics.

Design Diagrams

Use Case Diagram

The user interacts with the system by selecting specific mathematical queries.

```
usecaseDiagram
    actor User
    package "Number Theory System" {
        usecase "Check Number Property\n(Prime, Palindrome, etc.)" as UC1
        usecase "Generate Sequence\n(Fibonacci, Lucas)" as UC2
        usecase "Cryptographic Calculation\n(ModInverse, CRT)" as UC3
        usecase "View Performance Metrics\n(Time & Memory)" as UC4
    }
    User --> UC1
    User --> UC2
    User --> UC3
    UC1 ..> UC4 : <<include>>
    UC2 ..> UC4 : <<include>>
    UC3 ..> UC4 : <<include>>
```

Workflow Diagram

This illustrates the standard execution flow for any given module in the project.

```
flowchart LR
    A[Start] --> B[User Input n]
    B --> C[Start Timer & Tracemalloc]
    C --> D{Execute Algorithm}
    D --> E[Calculate Result]
    E --> F[Stop Timer & Tracemalloc]
    F --> G[Display Result]
    G --> H[Display Time & Peak Memory]
    H --> I[End]
```

Sequence Diagram

Example sequence for the `is_prime_miller_rabin` function.

```
sequenceDiagram
    participant User
    participant MainScript
    participant MathLogic
    participant Profiler

    User->>MainScript: Input Number (n), Rounds (k)
    MainScript->>Profiler: start_tracking()
    MainScript->>MathLogic: is_prime_miller_rabin(n, k)
    MathLogic->>MathLogic: Perform modular exponentiation
    MathLogic-->>MainScript: Return Boolean (True/False)
    MainScript->>Profiler: stop_tracking()
    Profiler-->>MainScript: Return (Time, Peak Memory)
    MainScript->>User: Print Result & Metrics
```

Class/Component Diagram

Since the code is functional, we visualize the logical grouping of functions as components.

```
classDiagram
    class NumberProperties {
        +factorial(n)
        +is_palindrome(n)
        +is_harshad(n)
        +is_armstrong(n)
    }
    class PrimeAnalysis {
        +is_prime(n)
        +miller_rabin(n, k)
        +pollard_rho(n)
        +prime_factors(n)
    }
    class ModularArithmetic {
        +mod_exp(base, exp, mod)
        +mod_inverse(a, m)
        +chinese_remainder_theorem()
    }
    class Sequences {
        +fibonacci(n)
        +lucas_sequence(n)
        +collatz_length(n)
    }

    NumberProperties --|> PrimeAnalysis : Uses
```

ER Diagram

*Note: As this is a computational tool with no persistent storage (Database), an Entity-Relationship Diagram is **Not Applicable**.*

Design Decisions & Rationale

1. **Library Selection (`tracemalloc` vs `sys.getsizeof`):**
    - *Decision:* Used `tracemalloc` .
    - *Rationale:* `sys.getsizeof` only measures the size of a specific object wrapper. `tracemalloc` tracks the actual memory blocks allocated by the interpreter during the execution of the function, providing a more realistic "peak memory" footprint.
2. **Algorithm Choice (Miller-Rabin):**
    - *Decision:* Included Miller-Rabin for primality testing.
    - *Rationale:* For very large integers (common in cryptography), trial division is too slow ($O(\sqrt{N})$). Miller-Rabin offers a probabilistic approach that is significantly faster ($O(k \log^3 n)$).
3. **Iterative vs Recursive:**
    - *Decision:* Mixed approach, but favored iterative for `factorial` and `fibonacci` checks.

- *Rationale:* Python has a recursion limit (typically 1000). Iterative solutions avoid `RecursionError` on deeper inputs.

Implementation Details

The project is implemented in **Python 3**. Below are highlights of key algorithms used:

- **Digital Root:** Implemented using a loop to repeatedly sum digits until a single digit remains.
- **Abundant/Deficient Numbers:** Calculates the sum of proper divisors using a loop up to $N/2$.
- **Miller-Rabin Test:** Uses Modular Exponentiation (`pow(a, d, n)`) to verify primality with high probability.
- **Pollard's Rho:** Implements a Floyd's cycle-finding algorithm to find non-trivial factors of composite numbers.
- **Zeta Approximation:** Approximates the Riemann Zeta function $\zeta(s)$ by summing the series $\sum(1/n^s)$.

Screenshots / Results (Sample Outputs)

Test Case 1: Deficient Number Check

```
Enter a number: 7
7 is a Deficient Number.
memory utilized is 1940
execution time is 0.000945 seconds
```

Test Case 2: Mersenne Prime Check

```
Enter a number p: 7
Is Mersenne prime? True
```

```
memory utilized is 2486
execution time is 0.00119 seconds
```

Test Case 3: Lucas Sequence Generation

```
Enter n for Lucas Sequence: 9
Lucas Sequence: [2, 1, 3, 4, 7, 11, 18, 29, 47]
Execution Time: 0.00055 seconds
Memory Used: 19.16 KB
```

Testing Approach

- **Unit Testing:** Each function was tested in isolation using known mathematical constants (e.g., verifying `factorial(5) = 120`).
- **Boundary Value Analysis:** Tested with:
  - Smallest inputs (0, 1).
  - Prime numbers vs Composite numbers.
  - Negative numbers (where applicable/handled).
- **Performance Testing:** Ran algorithms against increasing inputs (e.g., calculating partition functions for small vs large N) to observe the growth in execution time.

Challenges Faced

1. **Memory Tracking Overhead:** The `tracemalloc` library itself introduces slight overhead. Calibrating the code to ensure it measured the *algorithm* and not the overhead was tricky.
2. **Large Number Precision:** Calculating partition functions `p(n)` or factorials for large N results in massive integers. Python handles this automatically, but printing them slows down the "Execution Time" metric if the print statement is inside the timer block.
3. **Algorithmic Complexity:** Implementing the Partition Function using dynamic programming ($O(N^2)$) becomes slow for large N.

Learnings & Key Takeaways

- **Complexity Matters:** There is a tangible difference in execution time between $O(N)$ (linear scan) and $O(\log N)$ (binary/modular) algorithms, especially observed in the `is_prime` vs `miller_rabin` comparisons.
- **Profiling is Essential:** "Fast" code is subjective. Adding `time` and `tracemalloc` provided objective data to optimize loops and data structures.
- **Mathematical Depth:** Implementing these algorithms reinforced concepts of modular arithmetic, congruences (CRT), and number theoretic functions.

Future Enhancements

- **GUI Implementation:** Wrap the scripts in a `Tkinter` or `Flask` web interface for a better user experience.
- **Graphing Results:** Automatically plot Execution Time vs Input Size (N) using `matplotlib` to visually demonstrate Big-O complexity.