Practical Index:
A collection of 34 function implementations, each addressing a core mathematical or number-theoretic problem.

Title:
Comprehensive Implementation and Analysis of Key Number Theory and Arithmetic Functions in Python

Aim/Objectives:

- To develop Python functions that implement fundamental and advanced mathematical algorithms
- To demonstrate concepts like prime factorization, modular arithmetic, special sequences, and combinatorial functions
- To profile execution time and memory usage for assessing computational efficiency
- To build a modular library usable for educational and research purposes

Introduction:
This project involves designing and implementing a wide array of mathematical functions primarily focused on number theory, arithmetic properties, and computational algorithms. These include classical problems such as primality checking, factorization, divisor calculations, special sequences, modular arithmetic operations, and advanced functions like the partition function and Riemann zeta function approximations. Each function is developed aiming for clear, maintainable code with emphasis on correctness and efficiency.

Tools and Methodology:

- Programming Language: Python 3.x
- Standard libraries: math, random, functools, time, tracemalloc
- Each function isolates a distinct concept, tested with representative inputs
- Wrapper added to measure execution runtime and peak memory allocation using `time.perf_counter()` and `tracemalloc`
- Emphasis on clarity and readability to facilitate learning and adaptation

Brief Description of Key Functions:

- `factorialn`, `ispalindromen`, `meanofdigitsn`: Basic arithmetic and digit operations
- `primefactorsn`, `countdistinctprimefactorsn`, `isprimepowern`: Prime factorization and power checks
- `ismersenneprimep`, `isfibonacciprimen`: Special prime tests
- `modexp`, `modinverse`, `crtremaindersmoduli`: Modular arithmetic and solving congruences
- `multiplicativepersistencen`, `partitionfunctionn`: Repetitive digit manipulation and combinatorial counting
- `pollardrhon`, `isprimemillerrabinn`: Efficient factorization and primality testing algorithms
- Additional numeric properties like abundant, deficient, automorphic, pronic numbers

Results Achieved:

- Correct computation of all problem statements
- Performance metrics provide insights into efficiency and resource consumption
- Modular design allows easy extension and integration

Difficulties Faced:

- Balancing readability with performance in computational-heavy functions
- Implementing efficient prime checks and large number handling

Skills Developed:

- Proficiency in Python coding paradigms for mathematics
- Understanding of foundational algorithms in number theory
- Techniques to profile and optimize code performance
- Experience in modular software development and documentation

Implementation Approach:

- Functions are written modularly, with isolation of specific tasks (e.g., one function for factorials, another for detecting automorphic numbers).
- Leveraged Python's standard libraries (`math`, `random`, `functools`) for foundational operations to increase robustness and reduce complexity.

- Used iterative and recursive techniques where appropriate, e.g., for partitions and persistence calculations.
- Incorporated standard algorithms for primality and factorization like Miller-Rabin test and Pollard's Rho method to handle large inputs efficiently.
- Developed helper utilities, such as greatest common divisor calculations, to support advanced modular arithmetic functions and Chinese Remainder Theorem solvers.

Performance Measurement:

- Integrated precise timing using `time.perf_counter()` to measure execution duration per function call.
- Employed `tracemalloc` to monitor and record peak memory usage during function execution, enabling detailed resource profiling.

Results:

- All functions produce correct outputs on test cases representative of their mathematical domain.
- Performance and memory profiling highlight the efficiency of algorithms and identify computationally intensive operations like cryptographic primality tests and complex partition calculations.
- The modular structure allows easy testing, benchmarking, and extension.

Analysis and Learnings:

- Striking a balance between algorithmic efficiency and code clarity is critical, especially for educational applications.
- Python's native support for arbitrary large integers and built-in math functions simplifies implementation and improves accuracy.
- Profiling enriches understanding of computational costs, guiding optimization efforts.
- The project reinforces the importance of classical number theory concepts translated into practical computational tools.

This report documents the successful implementation and profiling of diverse mathematical functions, suitable for academic assignments, research prototypes, or learning aids.