Detailed Report: Number Theory and Utility Function Suite

1. Overview

This Python codebase presents 34 modular mathematical and number theory functions, each solving a distinct classical problem, with applications spanning education, research, and algorithmic problem-solving. Functions include arithmetic utilities (factorials, divisor/prime counting, palindromes), advanced number properties (abundant, deficient, amicable, harshad numbers), modular arithmetic (modexp, modular inverse, CRT), special sequences (Fibonacci/Lucas), combinatorial and analytical tools (partition function, zeta function), primality and factorization methods (Miller-Rabin, Pollard's rho), and more.

A benchmarking wrapper, `measure_performance`, is applied universally to all functional evaluations, capturing run time and peak memory allocation for diagnostic reporting.

2. Design Principles

- **Clarity and Modularity:** Each function is fully independent, accepting minimal input and returning results in their native mathematical format.
- **Performance Profiling:** The use of `time.perf_counter()` for high-resolution elapsed time and `tracemalloc` for peak memory snapshots gives fine-grained feedback on computational efficiency.
- **Readability and Reusability:** Core logic in each implementation prioritizes clarity, using Python's standard libraries where possible, and purposely avoids unnecessary complexity, aiding pedagogical use and extension.
- **Algorithmic Scope:** The suite covers a wide array of problem types—from determination of simple arithmetic properties (e.g., palindromes, automorphic, pronic) to complex iterative and recursive problems (e.g., Pollard's rho, partition numbers, Chinese Remainder Theorem solvers).

3. Execution Protocol

Each mathematical function can be called directly, but to profile its operation, the code uses:

```
python measure_performance(function, *args)
```

This returns a dictionary:

- `result`: The native answer from the function.
- `exec_time_sec`: Time to compute (seconds).
- `peak_mem_bytes`: Peak memory used (bytes).

The `run_all()` orchestrator runs a sample input for every function, making the suite instantly ready for benchmarking or demonstration, and yielding a dictionary of results for analysis.

4. Performance and Use

- **Efficiency:** Most computations, due to their mathematical nature, complete within milliseconds—and use negligible memory for reasonable inputs. Exceptionally, combinatorial and advanced factorization functions (such as partitionfunctionn and pollardrhon) may see longer execution times and more memory, dependent on input size.
- **Test Coverage:** The included `test_cases` dictionary provides sample arguments, offering immediate transparency for how to use each function and what sort of results to expect.

5. Recommendations and Extensibility

Users can:

- Extend test coverage with custom arguments,
- Integrate any function into broader analytical workflows,

- Analyze performance envelopes for larger input sizes,
- Use this as instructional material or a foundation for more advanced numerical libraries.

This codebase stands as a robust and educational resource for anyone exploring number theory or computational mathematics, combining trusted algorithms, extensible structure, and inbuilt diagnostic reporting for real-world computational tasks.