1. Cover Page

Project Title: Computational Analysis of Number Theoretic and Series Functions

Prepared for: [Recipient Name/Course Name]

Prepared by: [Your Name/Team Name]

Date: November 23, 2025

2. Introduction

This report documents the implementation and performance analysis of three fundamental mathematical functions: the Factorial function, the Riemann Zeta function approximation, and the Partition function. The primary goal of this project was to implement these algorithms in Python and evaluate their efficiency in terms of execution time and memory consumption using the `time` and `tracemalloc` libraries. The functions represent diverse computational challenges, ranging from simple iterative multiplication (Factorial) to infinite series approximation (Zeta) and dynamic programming (Partition).

3. Problem Statement

The core problem is to accurately implement algorithms for calculating $n!$, approximating $\zeta(s)$, and finding $p(n)$, and then quantitatively assess their runtime and memory usage to understand the computational trade-offs inherent in different algorithmic approaches.

4. Functional Requirements

The system (the Python implementation) must fulfill the following functional requirements:

1. **Factorial Calculation:** Accept a positive integer n and correctly calculate and output $n! = 1 \cdot 2 \cdot 3 \cdots n$.
2. **Riemann Zeta Approximation:** Accept a real number $s > 1$ and the number of terms N, and calculate the partial sum approximation of the Riemann Zeta function:

$$\zeta(s) \approx \sum_{k=1}^{N} \frac{1}{k^s}$$

3. **Partition Function Calculation:** Accept a positive integer n and correctly calculate the number of ways $p(n)$ that n can be written as a sum of positive integers (order does not matter).
4. **Performance Measurement:** Measure and report the execution time for each function call using the `time` module.

5. **Memory Profiling:** Measure and report the current and peak memory usage for each function call using the `tracemalloc` module.

5. Non-functional Requirements

1. **Performance:** The algorithms must execute efficiently, especially for moderately large inputs. Time and memory measurements must be consistently below acceptable thresholds for standard competitive programming limits.
2. **Accuracy:** The Zeta approximation must provide a result based on the specified number of terms N. The Factorial and Partition calculations must be mathematically exact for integer inputs within Python's integer limits.
3. **Maintainability:** The code should be clear, well-structured, and follow Python best practices.
4. **Usability:** The input mechanism (e.g., `input()`) should clearly prompt the user for the necessary variables.

6. System Architecture

The system employs a monolithic, single-script architecture.

- **Platform:** Jupyter Notebook environment (Python 3 interpreter).
- **Core Logic:** Defined within three distinct Python functions (`factorial` , `zeta_function` , `partition_function`).
- **Utility/Analysis Layer:** Standard Python libraries (`time` , `tracemalloc`) are used to wrap the core logic for performance profiling.
- **Data Flow:** User input $\rightarrow$ Utility wrapper (start timers/profiling) $\rightarrow$ Core function execution $\rightarrow$ Utility wrapper (stop timers/profiling) $\rightarrow$ Output results (function result, time, memory).

7. Design Diagrams

Workflow Diagram

This diagram illustrates the sequential execution and analysis workflow for any of the three functions.

Sequence Diagram

This diagram focuses on the interaction between the main script, the core function, and the utility libraries.

Class/Component Diagram

Since the system is procedural and uses built-in Python utilities, the components are defined as functional units.

ER Diagram (if storage used)

Since no persistent storage (database) is utilized, an ER Diagram is **not applicable** for this project. All data is transient (user input) or held in memory (DP arrays).

8. Design Decisions & Rationale

Feature	Design Decision	Rationale
Factorial	Iterative calculation using a `for` loop.	Simple, efficient for large integers (due to Python's arbitrary-precision integers), and avoids recursion overhead.
Zeta Approximation	Simple summation using `math.pow` or `k**s`.	The requirement was for an approximation using a finite number of terms (N). This is the most direct implementation of the series definition.
Partition Function	Dynamic Programming (DP) approach with a 1D array.	This method $p(n)$ has a pseudo-polynomial time complexity (approximately $O(n^2)$), which is significantly more efficient than a recursive approach or generating all combinations.
Performance Metrics	Use of `time` and `tracemalloc`.	These are standard, low-overhead built-in Python libraries providing accurate measurements of real-world execution time and process memory consumption.

9. Implementation Details

The implementation is contained within the uploaded Jupyter Notebook (`code.ipynb`).

Factorial Implementation (`factorial(n)`)

This function uses a simple iterative approach.

```
def factorial(n):
    # Initialize result
    factorial = 1
    # Loop from 1 to n (inclusive)
    for x in range(1, n + 1):
        factorial *= x
    print(factorial)
```

Riemann Zeta Approximation (`zeta_function(s, N)`)

Uses the built-in `math.pow` for efficient exponentiation within the summation.

```
import math
def zeta_function(s, N):
    result = 0
    # Sum the first N terms
    for k in range(1, N + 1):
        result += 1 / math.pow(k, s)
        return result
```

Partition Function Implementation (`partition_function(n)` **)**

A classic dynamic programming solution.

```
def partition_function(n):
    # dp[i] stores the number of partitions of i
    dp = [0] * (n + 1)
    dp[0] = 1 # Base case: 1 way to partition 0 (empty sum)

    # Iterate through all possible part sizes (i)
    for i in range(1, n + 1):
        # Update partitions for all numbers j >= i
        for j in range(i, n + 1):
            dp[j] += dp[j - i]
    return dp[n]
```

10. Screenshots / Results

The uploaded notebook includes execution results for sample inputs, demonstrating both correctness and performance metrics.

Function	Input	Result	Execution Time (s)	Memory Used (bytes)
Factorial	$n = 10$	$3,628,800$	0.000003	$1,749$
Zeta Approx	$s = 6, N = 7$	1.01733	0.000307	$19,523$
Partition	$n = 100$	$190,569,292$	0.000494	$3,048$

Note: Time and memory usage are highly dependent on the execution environment and input size. The values provided are sample runs from the attached notebook.

11. Testing Approach

The testing approach was a simple unit-level validation:

1. **Correctness Testing:** Each function was tested against known, small-scale results.
   - *Factorial:* $5! = 120$.
   - *Zeta:* $\zeta(2) \approx 1.6449$. The approximation was visually checked for convergence.
   - *Partition:* $p(5) = 7$ (5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1).
2. **Performance Testing:** The functions were executed with increasing values of n (or N) to observe how execution time and memory scale, ensuring the measurements correctly capture the overhead of the algorithm itself.

12. Challenges Faced

The main challenge was accurately measuring the performance metrics, particularly memory usage.

- **Memory Overhead:** `tracemalloc` reports the memory allocated by the Python process *during* the function execution. For very fast functions (like Factorial), the measurement can be dominated by the initial allocation of the environment, leading to high variance and difficulty in isolating the function's true memory footprint.
- **Zeta Function Convergence:** Determining a "sufficient" number of terms N for a good approximation required external knowledge and testing, though the requirement was strictly to sum N terms.

13. Learnings & Key Takeaways

1. **Dynamic Programming Efficiency:** The partition function, despite involving nested loops, is highly efficient (pseudo-polynomial $O(n^2)$) for the problem size, highlighting the power of DP in avoiding redundant calculations.
2. **Algorithm Complexity Dominates:** The time complexity of the algorithm is the single biggest factor in performance, far outweighing the minimal constant overhead from the `time` and `tracemalloc` instrumentation.
3. **Python's Strengths:** Python's handling of arbitrarily large integers makes the Factorial implementation simple and robust for large n, a feature not readily available in fixed-size integer languages.

14. Future Enhancements

1. **Benchmarking against Optimized Libraries:** Compare the custom implementations (e.g., Factorial) against highly optimized libraries (e.g., NumPy, SciPy) for larger inputs.
2. **Advanced Zeta Calculation:** Implement methods for the analytic continuation of the Zeta function or use the Euler-Maclaurin formula for faster convergence.
3. **Visualization:** Plot the execution time and memory usage as a function of the input size (n) to visually confirm the expected complexity curves ($O(n)$, $O(n^2)$, etc.).

15. References for all the functions

1. **Factorial Function ($n!$):** Standard definition from discrete mathematics and combinatorics.
2. **Riemann Zeta Function ($\zeta(s)$):** The Dirichlet series definition for $\mathrm{Re}(s) > 1$. Used the Python `math` library for the power operation.
3. **Partition Function ($p(n)$):** Dynamic programming algorithm based on the recurrence relation for the number of ways to partition an integer n.
4. **Performance Profiling:** Python's official documentation for the `time` and `tracemalloc` standard libraries.