Recursion in C++ and Python

Conor Sheehan

June 2024

1 Introduction

Recursion is a powerful and elegant technique in programming where a function calls itself to solve a problem. It is a natural way to handle problems that can be broken down into smaller, similar subproblems. Recursion is widely used in various algorithms, such as searching, sorting, and traversing data structures like trees and graphs.

When designing a recursive solution, two critical components must be defined: the base case and the recursive case. The base case provides a condition to terminate the recursion, ensuring that the function does not call itself indefinitely. The recursive case breaks the problem into smaller instances and calls the function with these smaller instances.

1.1 This Project

This project investigates several implementations of recursion in mathematical series and evaluates the efficacy of different approaches in terms of time taken for completion and activation records required on the stack. These investigations are conducted initially in C++ and subsequently in Python.

1.1.1 C++ Implementation

For the C++ implementation, a Recursion class is defined, with methods for each series under investigation. The time taken for the completion of each method is evaluated using the chrono library, and the number of activation records is counted simply using an integer attribute incremented per method call. This approach provides a quantitative measure of both performance and memory usage, allowing for a comprehensive analysis of recursive implementations in C++.

1.1.2 Python Implementation

Following the C++ analysis, the recursive algorithms are re-implemented in Python. Python's straightforward syntax and dynamic typing facilitate the

quick prototyping of recursive functions. The performance and stack usage are compared to the C++ implementations to assess the differences between the two languages in handling recursion.

1.2 Recursion Implementations

- Tree Recursion
- Sum of the Natural Numbers
- Power Series
- Taylor Series
- Fibonacci Series
- Binomial Coefficient
- Tower of Hanoi

2 Tree Recursion

Tree recursion is a form of recursion where a function makes multiple recursive calls within each invocation, creating a branching structure similar to a tree. This technique is useful for solving problems that can be divided into smaller, similar subproblems, such as combinatorial tasks, hierarchical data structures, and certain mathematical sequences.

In tree recursion, each call can generate multiple further calls, leading to a rapid increase in the number of function invocations. This exponential growth necessitates careful analysis of time and space complexity to ensure the algorithm's efficiency and feasibility for large inputs.

2.1 Time and Space Complexity Analysis

In this example, the `tree_recursion` method calls itself twice for each value of n greater than 0, forming a binary tree of recursive calls. We can derive the time and space complexity as follows:

2.1.1 Time Complexity

1. **Recurrence Relation**: Each call to `tree_recursion(n)` generates two additional calls: `tree_recursion(n-1)` and `tree_recursion(n-1)`. This gives us the recurrence relation:

   $$T(n) = 2T(n-1) + O(1)$$

   The $O(1)$ term accounts for the constant time operations (like printing and updating counters).

2. **Exponential Growth**: Solving this recurrence relation using the Master Theorem or by unrolling the recurrence, we find that the time complexity is:

$$T(n) \in O(2^{n+1} - 1)$$

2.1.2 Space Complexity

1. **Activation Records**: The space complexity is determined by the maximum depth of the recursion tree, which corresponds to the longest path from the root to a leaf.

2. **Depth of Recursion**: The maximum depth of the recursion tree is n, as each recursive call decreases the value of n by 1.

3. **Stack Space**: Therefore, the space complexity is:

$$S(n) \in O(n)$$

2.2 Code Output

```
Tree Recursion Object Instantiated
3, 2, 1, 0, 0, 1, 0, 0, 2, 1, 0, 0, 1, 0, 0,
The Number of Activation Records on the Stack :         15
The Maximum Depth of Activation Records on the Stack :  4
```

Figure 1: C++ Output for Tree_Recursion(3)

See figure 1 for the Tree Recursion C++ output. the total number of Activation records for Tree_Recursion(3) of 15 is in agreement with the time complexity of $2^4 - 1$, and the maximum stack depth of 4 agrees with the space complexity of order n.

3 Sum of Natural Numbers

3.0.1 Analytical Formula

The sum of the first n natural numbers can be calculated using a well-known analytical formula:

$$S_n = \frac{n(n+1)}{2}$$

This formula provides a direct method to compute the sum without the need for iterative or recursive procedures. For example, if $n = 10$:

$$S_{10} = \frac{10 \cdot (10+1)}{2} = \frac{10 \cdot 11}{2} = 55$$

3.0.2 Recursive Approach

In a recursive approach, the problem is broken down into smaller sub-problems. Specifically, to find the sum of the first n natural numbers, we can think of it as:

$$\text{sum}(n) = n + \text{sum}(n-1)$$

This continues until the base case is reached, where $n = 1$ and the sum is 1. This function repeatedly calls itself with decreasing values of n until it reaches the base case.

3.1 Time Complexity

The time complexity of a recursive function is determined by the number of times the function calls itself and the work done per call.

For the sum of natural numbers using recursion, the function calls itself n times, decrementing n by 1 with each call until it reaches the base case. Each call performs a constant amount of work (addition and the recursive call).

Therefore, the time complexity of this recursive solution is $O(n)$. This is because there are n recursive calls, and each call takes a constant amount of time.

3.2 Space Complexity

The space complexity of a recursive function is influenced by the call stack, which grows with each recursive call. In our case, each recursive call adds a new frame to the call stack until the base case is reached.

For the sum of natural numbers using recursion, there will be n frames on the call stack before the base case is reached. Thus, the space complexity of this recursive solution is $O(n)$.

Each frame on the stack requires a constant amount of space for the function's parameters and local variables, but since there are n recursive calls, the overall space complexity is $O(n)$.

3.3 Code Output

See in Figure 2 for the C++ output, the Recursive method returned quicker than the analytic calculation for N = 5. The total number of Activation Records aligns with O(N) also. The Analytic value returns quicker for large values of N though due to the large stack size involved recursively.

4 Power Series

4.1 General Power Series Formula

The general power series formula for raising a number m to the power n is:

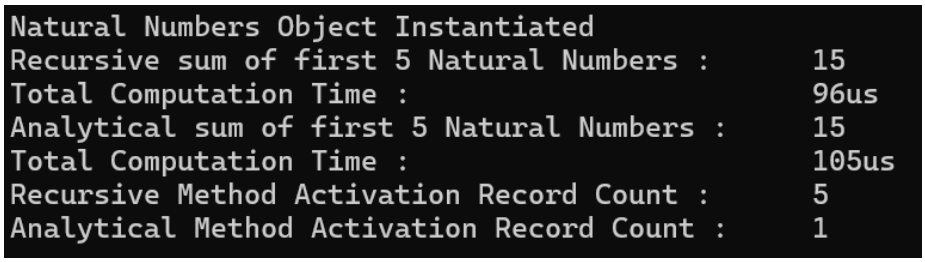

Figure 2: C++ Output for Sum of First 5 Natural Numbers Recursively & Analytically

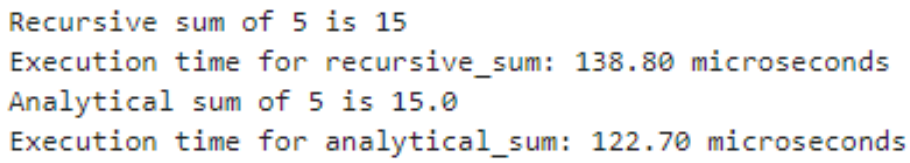

Figure 3: Python Output for Sum of First 5 Natural Numbers Recursively & Analytically

$$m^n = \underbrace{m \cdot m \cdot m \cdots m}_{n \text{ times}}$$

This operation multiplies m by itself n times.

4.1.1 Blunt Force Recursion

Blunt force recursion (also known as naive recursion) calculates m^n by directly multiplying m by itself n times through a recursive function.

4.1.2 Pseudo Code for Blunt Force Recursion

```
function blunt_force_power(m, n):
    if n == 0:
        return 1
    else:
        return m * blunt_force_power(m, n - 1)
```

4.1.3 Space Complexity of Blunt Force Recursion

The space complexity of blunt force recursion is $O(n)$ because it makes n recursive calls, each adding a new frame to the call stack.

4.2 Improved Power Recursion

The improved power recursion method (also known as exponentiation by squaring) is more efficient. It reduces the number of multiplications by leveraging the properties of even and odd exponents:

- If n is even: $m^n = (m \cdot m)^{n/2}$
- If n is odd: $m^n = m \cdot (m \cdot m)^{(n-1)/2}$

This approach significantly reduces the time complexity from $O(n)$ to $O(\log n)$.

4.2.1 Pseudo Code for Improved Power Recursion

```
function improved_power_recursion(m, n):
    if n == 0:
        return 1
    else if n % 2 == 0:
        return improved_power_recursion(m * m, n / 2)
    else:
        return m * improved_power_recursion(m * m, (n - 1) / 2)
```

4.3 Space Complexity of Improved Power Recursion

The space complexity of the improved power recursion method is $O(\log n)$ because it makes at most $\log n$ recursive calls, each adding a new frame to the call stack.

Summary

- **Blunt Force Recursion**
  - **Time Complexity**: $O(n)$
  - **Space Complexity**: $O(n)$
- **Improved Power Recursion**
  - **Time Complexity**: $O(\log n)$
  - **Space Complexity**: $O(\log n)$

4.4 Code Output

It can be seen from figure 4 that the second method improves on the first in both time taken, and memory used. The AR for each makes sense, with the initial method having 8,7,6,...0 records, and the second having records for m = 8,6,4,2,0.

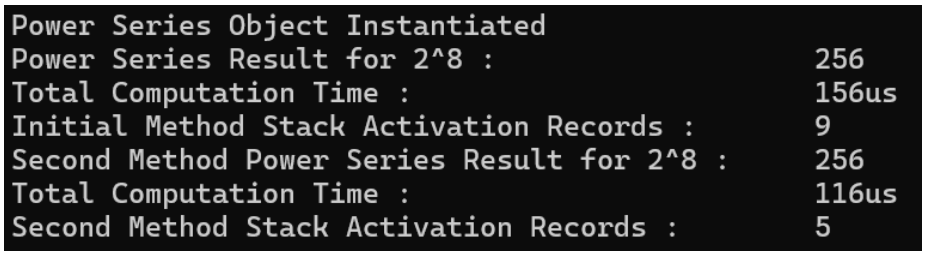

Figure 4: C++ Output for both Power Series Methods for 2^8

```
Power Series of 2 raised to 8 is 256
Execution time for MEthod 1 is : 142.30 microseconds
Power Series of 2 raised to 8 is 256
Execution time for Method 2 is : 141.30 microseconds
```

Figure 5: Python Output for both Power Series Methods for 2^8

5 Taylor Series

The Taylor series is a powerful mathematical tool used to approximate complex functions with polynomials. Given a function $f(x)$, the Taylor series expansion about a point a is expressed as:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$$

More formally, the Taylor series of a function $f(x)$ around the point a is given by:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Where:

- $f^{(n)}(a)$ denotes the n-th derivative of $f(x)$ evaluated at a.
- $n!$ is the factorial of n.

For simplicity, we often expand the series around $a = 0$, known as the Maclaurin series:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}x^n$$

5.1 Recursive Calculation of Taylor Series

The Taylor series can be computed using recursion. One method involves blunt force recursion where we recursively calculate the polynomial terms and sum them. The pseudo code below implements this approach:

```
function taylor_method_one(x, n)
    static p = 1, f = 1
    if n == 0
        return 1
    else
        r = taylor_method_one(x, n - 1)
        p = p * x
        f = f * n
        return r + (p / f)
```

In this method:

- p represents x^n and is updated by multiplying with x in each recursive call.
- f represents $n!$ and is updated by multiplying with n in each recursive call.
- The base case returns 1 when n reaches 0.
- The recursive call adds the current term $\frac{p}{f}$ to the result of the previous terms.

5.2 Horner's Method for Taylor Series

Horner's method optimizes the computation of polynomial terms, making it more efficient by reducing the number of multiplications required. The Taylor series using Horner's method can be expressed as:

$$f(x) = 1 + x\left(1 + x\left(\frac{1}{2} + x\left(\frac{1}{3} + \cdots\right)\right)\right)$$

The recursive implementation of Horner's method in pseudo code is as follows:

```
function taylor_series_horner_method(x, n)
    static s = 1
    if n == 0
        return s
    s = 1 + (x / n) * s
    return taylor_series_horner_method(x, n - 1)
```

In this method:

- s accumulates the series result.
- Each recursive step updates s by multiplying the current value of s by $\frac{x}{n}$ and adding 1.
- The base case returns the accumulated result when n reaches 0.

5.2.1 Computational Improvement of Horner's Method

Horner's method significantly improves computational efficiency over the blunt force recursive method. The key improvements are:

- **Reduced Multiplications**: In the blunt force method, each term requires recalculating powers and factorials, resulting in multiple multiplications per term. Horner's method reduces this to a single multiplication and addition per term.
- **Lower Complexity**: The blunt force method has a time complexity of $O(n^2)$ due to repeated calculations of powers and factorials. Horner's method reduces this to $O(n)$ because it processes each term in a single step.
- **Improved Numerical Stability**: Horner's method accumulates results in a way that reduces round-off errors, making it more numerically stable for large values of n.

Overall, Horner's method provides a more efficient and stable way to compute Taylor series, making it preferable for practical implementations.

5.3 Code Output

```
Taylor Series Object Instantiated
Initial Static Variable Method Result :         20.0855
Total Computation Time :                        364us
Horner Method Result :                          20.0855
Total Computation Time :                        275us
```

Figure 6: C++ Output for the MacLaurin Expansion of e^3 for 25 terms for both the Horner Method and Blunt Force Method

See the above comparison for the time taken with both methods. We can see the Horner method approach is considerably quicker than the the initial approach, due to the reasons outlined in section 5.2.1.

```
Taylor Series of e^3 with 25 terms is  20.085536923187654
Execution time for Method 1 is : 140.60 microseconds
Taylor Series of e^3 with 25 terms is 20.08553692318766
Execution time for Horner method is : 107.70 microseconds
```

Figure 7: Python Output for the MacLaurin Expansion of e^3 for 25 terms for both the Horner Method and Blunt Force Method

6 Fibonacci Series

6.0.1 Introduction to the Fibonacci Series

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. This sequence appears in many different areas of mathematics and computer science, as well as in nature. The first few numbers in the Fibonacci series are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The n-th Fibonacci number is commonly denoted as $F(n)$, and it can be defined by the following recurrence relation:

$$F(n) = F(n-1) + F(n-2)$$

with initial conditions:

$$F(0) = 0$$

$$F(1) = 1$$

6.1 Blunt Force Recursion

Blunt force recursion involves directly translating the mathematical definition of the Fibonacci series into a recursive algorithm. Although this approach is straightforward, it is highly inefficient due to the repeated calculations of the same values.

6.1.1 Pseudo Code for Blunt Force Recursion

```
function fibonacci_recursive(n)
    if n <= 1
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Time Complexity: $O(2^n)$
Space Complexity: $O(n)$

6.2 Memoization

Memoization is an optimization technique that involves storing the results of expensive function calls and reusing them when the same inputs occur again. This technique significantly improves the efficiency of the Fibonacci calculation by avoiding redundant computations.

6.2.1 Pseudo Code for Memoization

```
function fibonacci_memoization(n, memo)
    if memo[n] is not -1
        return memo[n]
    if n <= 1
        memo[n] = n
    else
        a = fibonacci_memoization(n - 1, memo)
        b = fibonacci_memoization(n - 2, memo)
        memo[n] = a + b
    return memo[n]
```

Time Complexity: $O(n)$
Space Complexity: $O(n)$

6.3 Complexity Analysis for Memoization Method

Time Complexity: $O(n)$

Each Fibonacci number up to $F(n)$ is calculated exactly once and stored in the vector.

Space Complexity: $O(n)$

The space complexity is due to the storage required for the memoization table (in the C++ implementation, I used a member vector `F`) and the call stack used by the recursion. Since we store the Fibonacci numbers in a vector of size $n + 1$, the space complexity is $O(n)$.

By using memoization, the efficiency of calculating the Fibonacci series is greatly improved compared to the blunt force recursion method.

6.4 Code Output

It can be seen from the above snippet that the Memoization method is both drastically quicker, and drastically more space efficient than the brute force method by orders of magnitude. The Space consumption from the initial method is due to the Tree Recursion behaviour of the function, where each call results in 2 more calls. The time and space requirements using memoization are linear with respect to N here, while the brute force method is not.

```
Fibonacci Object Instantiated
Using Initial Blunt Force Method Fifteenth Fibonacci Element:   9227465
Total Computation Time :                                        306456us
Total Number of Activation Records :                            29860703
Using Memoization Method Fifteenth Fibonacci Element:           9227465
Total Computation Time :                                        342us
Total Number of Activation Records :                            69
```

Figure 8: C++ Output for both Fibonacci Series Methods for the 35^{th} Fibonacci element

```
The 35th Fibonacci element is  9227465
Execution time for recursive method is : 4873508.40 microseconds
The 35th Fibonacci element is 9227465
Execution time for Memoization method is : 211.80 microseconds
```

Figure 9: Python Output for both Fibonacci Series Methods for the 35^{th} Fibonacci element

7 Binomial Coefficient

The binomial coefficient, denoted as $\binom{n}{r}$, represents the number of ways to choose r elements from a set of n elements without regard to the order of selection. It is a fundamental concept in combinatorics and has applications in probability, statistics, and various fields of mathematics.

7.1 Calculation Using Factorials

The binomial coefficient can be calculated using factorials. The formula is given by:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

where $n!$ (n factorial) is the product of all positive integers up to n:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

Pseudo code for calculating $\binom{n}{r}$ using factorials:

```
function factorial(n):
    if n == 0 or n == 1:
        return 1
    result = 1
    for i from 2 to n:
        result = result * i
    return result
```

```
function binomialCoefficient(n, r):
    if r > n:
        return 0
    if r == 0 or r == n:
        return 1
    numerator = factorial(n)
    denominator = factorial(r) * factorial(n - r)
    return numerator / denominator
```

Time Complexity: $O(n)$ for the factorial calculation.
Space Complexity: $O(1)$ since we only use a constant amount of extra space.

7.2 Calculation Using Pascal's Triangle

Pascal's Triangle is a triangular array of numbers where each number is the sum of the two numbers directly above it. The binomial coefficient $\binom{n}{r}$ is located at the n-th row and r-th column of Pascal's Triangle.

To compute $\binom{n}{r}$ using Pascal's Triangle, we use the recursive relationship:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

with base cases:

$$\binom{n}{0} = \binom{n}{n} = 1$$

Pseudo code for calculating $\binom{n}{r}$ using Pascal's Triangle:

```
function binomialCoefficientPascal(n, r):
    if r > n:
        return 0
    if r == 0 or r == n:
        return 1
    dp = array of size (n+1) x (r+1) initialized to 0
    for i from 0 to n:
        for j from 0 to min(i, r):
            if j == 0 or j == i:
                dp[i][j] = 1
            else:
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
    return dp[n][r]
```

Time Complexity: $O(n \times r)$ due to the nested loops iterating through the triangle.
Space Complexity: $O(n \times r)$ for the storage of the DP table.

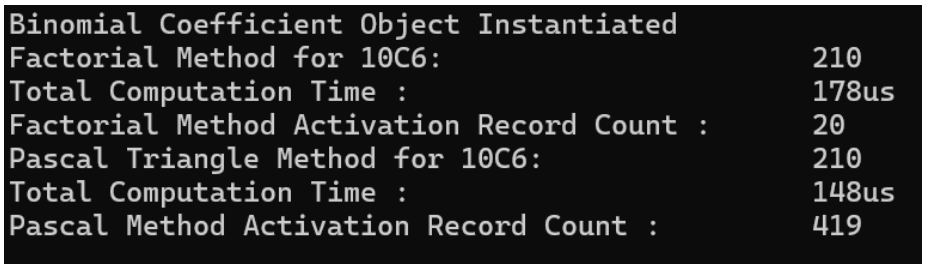

Figure 10: C++ Output for both Factorial and Pascal Method for 10C6

7.3 Code Output

We can see from the above snippet that the Pascal Triangle Dynamic Programming method is quicker, however it does require much more calculations. This is as expected. The Factorial method only needs 20 AR (6!+ 4! + 10!), whereas the Pascal Method required two 'Parent' values to be calculated for each value in the triangle until we reach 10C6 from the top. These AR's must be held on the stack, until we start returning 1's from the top of the Triangle down. They can then be popped off the stack.

An alternative approach here could be to use an array, or if you want to be fancy use a dynamic approach with vectors and store your values on the heap rather than the stack, keeping our AR values down.

```
10C6 is  210.0
Execution time for recursive method is : 149.30 microseconds
10C6 is 210
Execution time for Pascal Triangle method is : 325.00 microseconds
```

Figure 11: Python Output for both Factorial and Pascal Method for 10C6

8 Tower of Hanoi

The Tower of Hanoi is a classic mathematical puzzle that consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, with the smallest disk at the top. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the top disk from one rod and placing it on top of another rod.

3. No disk may be placed on top of a smaller disk.

The minimum number of moves required to solve the Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

8.1 Trivia

- The Tower of Hanoi puzzle was invented by the French mathematician Édouard Lucas in 1883.
- It is based on a legend about a temple in India where there are three diamond needles and sixty-four disks of pure gold. Monks are tasked with moving the disks from one needle to another, following the rules above. Legend has it that when the last move is completed, the world will end.

8.2 Recursive Solution

The Tower of Hanoi problem can be elegantly solved using recursion. Here's a simple C++ implementation:

```
void TOH(int n, char A, char B, char C) {
    if (n > 0) {
        TOH(n - 1, A, C, B);
        std::cout << "\nMove disk from tower " << A <<
        " to tower " << C;
        TOH(n - 1, B, A, C);
    }
}
```

This recursive function, `TOH`, takes four parameters: `n` (the number of disks), and characters `A`, `B`, and `C`, representing the three rods. Here's how the recursive algorithm works:

1. Move $n - 1$ disks from rod `A` to rod `B`, using rod `C` as an auxiliary.
2. Move the remaining disk from rod `A` to rod `C`.
3. Move the $n - 1$ disks from rod `B` to rod `C`, using rod `A` as an auxiliary.

This process continues recursively until all disks are moved to the target rod, following the rules of the Tower of Hanoi puzzle.

8.3 Code Output

Figure 7 shows the printed output for the Tower of Hanoi solution for 3 disks. If you grab a pen and paper, or have 3 disks handy, I'm sure you'll be as pleased as I was with it! The exact same steps were produced in the Python code.

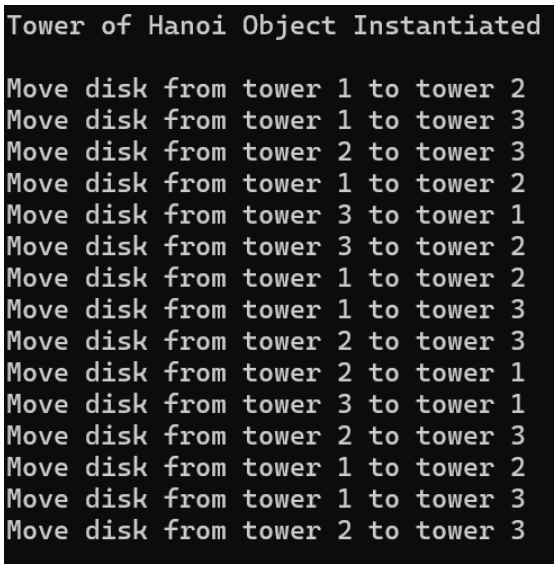

Figure 12: Tower of Hanoi Solution for 3 disks using C++

9 C++ & Python Code

9.1 Tree Recursion

9.1.1 C++ Code

Consider the following implementation of tree recursion in C++ within a class method. This function prints the value of n and then recursively calls itself twice, reducing n by 1 each time.

```
void Recursion::tree_recursion(int n)
{
    // Tree Recursion Implementation Method
    std::cout << n << ",";

    call_count++;
    current_depth++;
    max_depth = std::max(current_depth, max_depth);

    if (n > 0) {
        tree_recursion(n - 1);
        tree_recursion(n - 1);
    }

    current_depth--;
}
```

9.2 Natural Numbers

9.2.1 C++ Code

See below methods for a recursive and analytic calculation respectively:

```
int Recursion::recursive_sum(int N)
{
        recursive_stack_count++;
        if (N == 1)
        {
                return 1;
        }
        else
        {
                return (recursive_sum(N - 1) + N);
        }
}
int Recursion::analytical_sum(int N)
{
        analytical_stack_count++;
        return ((N * (N + 1)) / 2);
}
```

9.2.2 Python Code

```
        def recursive_sum(self, n):
        self.recursive_stack_count += 1
        if (n == 1):
            return 1
        return n + self.recursive_sum(n-1)

    def analytic_sum(self, n):
        return (n*(n + 1)/2)
```

9.3 Power Series

9.3.1 C++ Code

```
    int Recursion::power_recursion(int m, int n)
{
        power_stack_count++;
        if (n == 0)
        {
                return 1;
        }
```

```
        else
        {
                return (m*power_recursion(m,n-1));
        }
}
int Recursion::improved_power_recursion(int m, int n)
{
        improved_power_stack_count++;
        if (n == 0)
        {
                return 1;
        }
        else if (n % 2 == 0)
        {
                return (improved_power_recursion(m*m, n/2));
        }
        else
        {
                return (m*improved_power_recursion(m*m,(n-1)/2));
        }
}
```

9.3.2 Python Code

```
        def recursive_method_one(self, m, n):
        if (n == 0):
            return 1
        return (m)*self.recursive_method_one(m,n-1)

    def recursive_method_two(self, m, n):
        if (n ==0):
            return 1
        if (n % 2 == 0):
            return self.recursive_method_two(m*m,n/2)
        else:
            return m*self.recursive_method_two(m*m, (n-1)/2)
```

9.4 Taylor Series

9.4.1 C++ Code

```
double Recursion::taylor_method_one(int x, int n)
{
        static double p=1, f = 1;
        if (n == 0)
```

```
        {
                return 1;
        }
        else
        {
                double r = taylor_method_one(x, (n - 1));
                p = p * x;
                f = f * n;
                return (r + (p / f));
        }
}
double Recursion::taylor_series_horner_method(int x, int n)
{
        static double s=1;
        if (n == 0)
        {
                return s;
        }
        s = 1 + (x / static_cast<double>(n))*s;
        return taylor_series_horner_method(x, n - 1);
}
```

9.4.2 Python Code

```
    p = 1
    f = 1
    @staticmethod
    def taylor_method_one(x, n):
        if n == 0:
            return 1
        r = Taylor_Series_Class.taylor_method_one(x, n - 1)
        Taylor_Series_Class.p = Taylor_Series_Class.p*x
        Taylor_Series_Class.f = Taylor_Series_Class.f*n
        return r + (Taylor_Series_Class.p / Taylor_Series_Class.f)

    s = 1
    @staticmethod
    def taylor_series_horner_method(x, n):
        if n == 0:
            return Taylor_Series_Class.s
        Taylor_Series_Class.s = 1 + (x / n) * Taylor_Series_Class.s
        return Taylor_Series_Class.taylor_series_horner_method(x, n - 1)
```

9.5 Fibonacci Series

9.5.1 C++ Code

```
int Recursion::fibonacci_method_one(int n)
{
        fib_m1_stack++;
        if (n <= 1)
        {
                return n;
        }
        else
        {
                return (fibonacci_method_one(n - 2)
                + fibonacci_method_one(n - 1));
        }
}
    int Recursion::fibonacci_method_two(int n)
{
        fib_m2_stack++;
        if (n <= 1) {
        return n;
    }
        if (F[n] != -1) {
                return F[n];
        }
        F[n] = fibonacci_method_two(n - 1)
                + fibonacci_method_two(n - 2);
        return F[n];
}
```

9.5.2 Python Code

```
        def fibonacci_method_one(self, n):
        if (n <=1):
            return n
        else:
            return (self.fibonacci_method_one(n-2)
            + self.fibonacci_method_one(n-1))

    F = []

    def fibonacci_method_two_list(self,n):
        Fibonacci_Series_Class.F = [-1] * n

    def fibonacci_method_two(self, n):
```

```
        if n <= 1:
            return n
        if Fibonacci_Series_Class.F[n-1] != -1:
            return Fibonacci_Series_Class.F[n-1]
        Fibonacci_Series_Class.F[n-1] =
        self.fibonacci_method_two(n - 1) +
        self.fibonacci_method_two(n - 2)
        return Fibonacci_Series_Class.F[n-1]
```

9.6 Binomial Coefficicent

9.6.1 C++ Code

```
    int Recursion::factorial_method(int n, int r)
{
        int num = factorial(n);
        int denom_a = factorial(r);
        int denom_b = factorial(n - r);
        return (num / (denom_a * denom_b));
}
int Recursion::pascal_triangle_method(int n, int r)
{
        pascal_stack++;
        if (n == r || r == 0)
        {
                return 1;
        }
        return (pascal_triangle_method(n-1,r-1) +
 pascal_triangle_method(n-1,r));
}
```

9.6.2 Python Code

```
        def factorial(self, n):
        if (n <=1):
            return n
        else:
            return n*self.factorial(n-1)
    def recursive_method(self, n, r):
        num = self.factorial(n)
        denom_a = self.factorial(r)
        denom_b = self.factorial(n-r)
        return (num/(denom_a*denom_b))
    def pascal_method(self, n,r):
        if (n == r or r == 0):
```

```
            return 1
        else:
            return (self.pascal_method(n - 1, r - 1) + self.pascal_method(n - 1,
```

9.7 Tower of Hanoi

9.7.1 C++ Code

```
    void Recursion::TOH(int n, int A, int B, int C)
{
        if (n > 0)
        {
                TOH(n - 1, A, C, B);
                std::cout << "\nMove disk from tower "
  << A << " to tower " << C;
                TOH(n - 1, B, A, C);
        }
}
```

9.7.2 Python Code

```
        def TOH(self, a,b,c,n):
        if (n>0):
            self.TOH( a, c, b, n-1)
            print("Move disk from tower ", a, " to tower ", c)
            self.TOH( b, a, c, n-1)
```