Oddonacci Sequence Calculations: A Study of Recursive Methods

Compiled in 2024

1 Introduction

This document provides an in-depth analysis of the Oddonacci sequence calculations using two distinct recursive methods implemented in Java. The Oddonacci sequence is a variation of the Fibonacci sequence where each number is the sum of the preceding three numbers, as opposed to the traditional two.

2 Project Overview

The project implements two approaches to compute Oddonacci numbers: a linear tail-recursive method and a multiple recursive method. Both methods are explored to evaluate their performance in terms of computational efficiency and execution time.

3 Implementation Details

3.1 Package and Imports

The Java implementation resides within the package `L_Oddonacci` and employs the following imports:

- `java.math.BigInteger` for handling very large numbers.
- `java.io.*` for input and output operations, crucial for performance tracking.

3.2 Main Class Structure

The main class `Linear_Oddonacci` contains methods to calculate Oddonacci numbers and measure their performance. Key methods include:

- `MultipleOddonacci_Run(int)`: Executes the multiple recursive calculation multiple times.
- `LinearTailRecursiveOddonacci_Run(int)`: Handles the linear tail-recursive calculation.
- `Full_Test(int)`: Compares outputs from both methods across a range of inputs.

3.3 Methods

3.3.1 Linear Tail-Recursive Method

This method is defined as follows:

```
public static BigInteger LinearTailRecursiveOddonacci(int n, BigInteger a,
    if (n < 0)
        throw new IllegalArgumentException("Negative␣index␣not␣allowed");
    else if (n <= 3)
        return c;
    else
        return LinearTailRecursiveOddonacci(n - 1, b, c, a.add(b).add(c));
}
```

3.3.2 Multiple Recursive Method

The multiple recursive method is more computationally intensive:

```
private static BigInteger MultipleOddonacci(int n, BigInteger first, BigIn
    if (n < 0)
        throw new IllegalArgumentException("Negative␣index␣not␣allowed");
    else if (n <= 3)
        return BigInteger.ONE;
    else
        return MultipleOddonacci(n - 1, second, third, first.add(second).a
                .add(MultipleOddonacci(n - 2, second, third, first.add(secon
                .add(MultipleOddonacci(n - 3, second, third, first.add(secon
}
```

4 Performance Analysis

The performance of both methods is analyzed by measuring the time taken to compute Oddonacci numbers at various positions in the sequence. Results are saved to text files, facilitating a detailed comparison of the computational efficiency between the linear tail-recursive and the multiple recursive approaches.

5 Results and Discussion

Initial tests indicate that the linear tail-recursive approach is significantly more efficient, particularly as the index of the Oddonacci number increases. This efficiency stems from the reduced complexity and lower memory usage associated with tail recursion.

6 Conclusion

The analysis highlights the benefits of using tail recursion for recursive calculations in terms of both speed and memory efficiency. Future work may involve exploring further optimizations or hybrid approaches that could potentially enhance performance even more.