Arbitrary Precision Arithmetic Library

Final Project for CS1023 Course

Submitted by

Asit Patel (CS24BTECH11046)

Team Members:

Member 2 (ID)

Member 3 (ID)

Indian Institute of Technology Hyderabad

May 1, 2025

Contents

Abstract

This project implements a comprehensive arbitrary precision arithmetic library that overcomes the limitations of standard data types in programming languages. The library provides:

- Unlimited precision integer arithmetic operations
- High-precision floating-point calculations
- Exact decimal representations without rounding errors
- Four fundamental operations: addition, subtraction, multiplication, and division
- Usage of command-line interface for command executions.

Designed for educational purposes, this implementation demonstrates how basic arithmetic operations can be extended beyond hardware limitations using clever algorithms and proper data representation.

Introduction

The Need for Arbitrary Precision

In standard programming languages like Java, numerical data types have fixed sizes that limit their range and precision. For example:

Type	Size	Range
int	32-bit	-2^{31} to $2^{31}-1$
long	64-bit	-2^{63} to $2^{63}-1$
float	32-bit	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
double	64-bit	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$

Table 1: Standard Java Data Type Limitations

These limitations become problematic in applications requiring very high precision like Financial computations, Scientific simulations, Mathematical research and Cryptography.

Our Solution

Our library solves these problems by:

- Representing numbers as strings (allowing unlimited length)
- Implementing digit-wise arithmetic operations
- Providing precise control over decimal places
- Handling edge cases and special scenarios

Project Structure

File Organization

The project follows standard Maven directory structure:

- **src/main/java/arbitraryarithmetic/**
  - `AInteger.java` - Core integer operations

  - `AFloat.java` - Floating-point implementation
  - `MyInfArith.java` - Command-line interface
- **src/main/python/**
  - `MyInfArith.py` - Python wrapper script
- **pom.xml** - Maven build configuration
- **README.md** - Usage instructions

Key Components

- **AInteger Class**:
  - Handles arbitrary precision integers by using string representation
  - Stores numbers as strings with sign flag for positive negative
  - Implements all basic arithmetic operations
  - Implements Karatsuba method for multiplication and recursive long division.
- **AFloat Class**:
  - Manages high-precision decimals
  - Tracks decimal positions
  - Automatically aligns decimals for operations
- **MyInfArith**:
  - Provides user-friendly command-line interface
  - Validates input and formats output

Design and Implementation

Core Algorithms

Karatsuba Multiplication

This efficient multiplication algorithm reduces the complexity from $O(n^2)$ to $O(n^{1.585})$*byrecursivelybreakingtheproblemin*

1. Split numbers into high and low parts: $x = x_1 \times 10^m + x_0$
2. Compute three partial products:
   - $z_0 = x_0 \times y_0$
   - $z_1 = (x_1 + x_0) \times (y_1 + y_0)$
   - $z_2 = x_1 \times y_1$
3. Combine results: $z_2 \times 10^{2m} + (z_1 - z_2 - z_0) \times 10^m + z_0$

Decimal Handling in AFloat

The `pre_process` method ensures proper decimal alignment:

```
public int pre_process(AFloat other) {
    if(no_decimal_1 > no_decimal_2) {
        // Pad other_number with trailing zeros
```

```
        other.value += "0".repeat(no_decimal_1 - no_decimal_2);
    } else {
        // Pad this_number with trailing zeros
        this.value += "0".repeat(no_decimal_2 - no_decimal_1);
    }
}
```

This ensures numbers like 12.34 and 5.678 become 12.340 and 5.678 before operations. It make the number of decimal digits same by padding zeroes to the end

Testing with sample Inputs

Representative Test Cases

- **Large Integer Addition**:
  - Input: $10^{100} + 1$
  - Output: $100\ldots001$ (101 digits)
  - Verifies carry propagation
- **Precise Subtraction**:
  - Input: $1 - 0.99999999999999999$
  - Output: 0.00000000000000001
  - Tests exact decimal handling
- **Multiplication**:
  - Input: $123456789 \times 987654321$
  - Output: 121932631112635269
  - Validates algorithm correctness
- **Division**:
  - Input: $1/3$
  - Output: $0.\overline{3}$ (1000 decimal places)
  - Demonstrates precision control
- **Float Addition**:
  - Input: $999999999999.999999999999 + 0.000000000001$
  - Output: 1000000000000.0
  - Tests decimal alignment

Build and Usage

Building the Project

```
# Clone repository
git clone https://github.com/your-repo/arbitrary-arithmetic.git
cd arbitrary-arithmetic

# Build with Maven
mvn clean package
```

```

# Output JAR will be at:
# target/Final_Project-1.0-SNAPSHOT.jar
```

Command-line Usage

Basic syntax:

```
java -jar target/Final_Project-1.0-SNAPSHOT.jar <int/float> <operation> <num1> <num2>
```

Examples:

- Integer multiplication:

```
java -jar target/Final_Project-1.0-SNAPSHOT.jar int multiply 123456789 987654321
```

- Floating-point division:

```
java -jar target/Final_Project-1.0-SNAPSHOT.jar float divide 22 7
```

Limitations and Future Work

Current Limitations

- **Performance**: Operations slow down significantly with numbers ¿10,000 digits
- **Memory Usage**: Large numbers consume substantial memory
- **Functionality**: Limited to basic arithmetic operations
- **Precision**: Floating-point division precision capped at 1000 places

Future Enhancements

- The code could have been made more cleaner and modular for easy understanding and expansion.
- Advanced mathematical functions like square-root, logarithmic and exponential functions.
- For some of the operations, more efficient methods could have been used. Using Lists for storing the numbers instead of String can result in faster computations.
- Support for rational and irrational numbers can also be added in form of new class.

Key Learnings

This project provided invaluable learning experiences in multiple areas:

Technical Skills

- Implementation of advanced algorithms like Karatsuba multiplication
- String manipulation at scale. Object-oriented design principles. Exception handling and edge cases.
- Build system in Java. Use of Maven for project compilation and Jar file generation.
- Usage of Python for executing the Java code using Jar.

- Using Git and Github for version control. Container Image implementation using Docker. Pushing it on Dockerhub and pulling and running it on other devices.

Problem Solving

- Debugging complex numerical issues
- Performance optimization techniques
- Trade-off analysis between different approaches
- Testing methodologies for numerical code

Conclusion

This project successfully implements arbitrary precision arithmetic using string-based algorithms, overcoming standard data type limitations. This library delivers working arbitrary precision arithmetic but has clear performance limits with large numbers. While the core operations function correctly, real-world use would require significant optimization. The project provided valuable experience with numerical algorithms and their implementation from scratch.