ABITRARY-PERCISION-INTEGER CALCULATOR

Nehemiah Yoseph Kumssa

13/12/2024

ABITRARY-PERCISION-INTEGER CALCULATOR

BY

NEHEMIAH YOSEPH KUMSSA

13/12/2024

# Introduction

The aim of this project is to build a manual implementation of a calculator that can perform Arbitrary Precision Arithmetic for integers of any size.

## Objectives

1. The calculator should be able to perform the following operations for integers of any size and handle signed integers as well:
   1. Addition
   2. Subtraction
   3. Multiplication
   4. Division and Modulo
   5. Exponential
   6. Factorial
2. Additional (Optional) Operations include:
   1. Support for non-decimal bases
      1. Calculations in different bases
      2. Implement base conversions for arbitrary-precision integers
   2. Fractions
      1. Handle arithmetic operations on fractions as well
   3. Logarithms
      1. Integer logarithms
      2. Compute square roots of arbitrary-precision integers
3. The calculator should be a wrapped in a REPL
   1. Enable the REPL to maintain history of calculations
   2. Allow recalling past results and use past results
   3. Enable input/output formatting, (e.g. Binary, decimal, mixed fractions, improper fractions, quotient and remainder)
   4. Custom Commands I.e. clear, help, exit, save export
4. Can handle Errors
   1. Invalid input
   2. Division by 0
   3. Overflow
   4. provide meaningful error messages.

Additional Enhancements

1. **Concurrency and Parallelism** (Optimization Objective):
   * Use parallel processing for computationally heavy operations like factorial or multiplication of very large integers.
   * Divide tasks into smaller chunks and compute in parallel.
2. **Performance Optimizations**:
   * Use efficient algorithms (e.g., Karatsuba multiplication for faster multiplication or binary exponentiation).
   * Implement caching/memoization for operations like factorial or repeated calculations.
3. **Extensibility**:
   * Ensure the architecture supports adding future operations easily.
   * Modularize the implementation for clarity and maintainability.
4. **Cross-Platform Compatibility**:
   * Test the calculator across different operating systems to ensure consistent behavior.
5. **Custom Commands for the REPL**:
   * Add commands like clear, help, or exit.
   * Provide options for saving or exporting calculation history.

## Project Constraints

1. No native support for arbitrary-precision arithmetic.
2. No libraries used for core functionalities

Therefore based of these constraints I have chosen to use Rust programming language as it does not support for arbitrary precision integers in its own standard library, I allows low level control and concurrency, and can implement the REPL application.

# Program Design

## General Guidelines

1. The calculator must be able to handle arbitrarily large integers (Scalability).
2. The algorithms should be optimized for peak performance. (Efficient).
3. The implementation should be modular and extensible as well to support addition of more features.
4. Error handling and safe input handling.

## Data structures

### Representing Arbitrary-Precision Integers

A structure to handle integers of arbitrary size.

The Pseudocode for this would be as follows:

LargeInt:

Digits: array of integers

Sign: integer’s sign with 1 for positive and -1 for negative integers

With the Digits field storing the digits in reverse order for simple arithmetic, since most mathematical calculations would begin form the lowest value all the way to the largest (for instance addition of two integers would start form the values in the one’s place value then the 10’s place value as it increases).

While the Sign field indicating whether the integer is a positive or a negative integer

### Representing Fractions

In fractions we would call the method LargeInt, for signs, but only have two fields of numerator and denominator.

Fractions:

Numerator: LargeInt

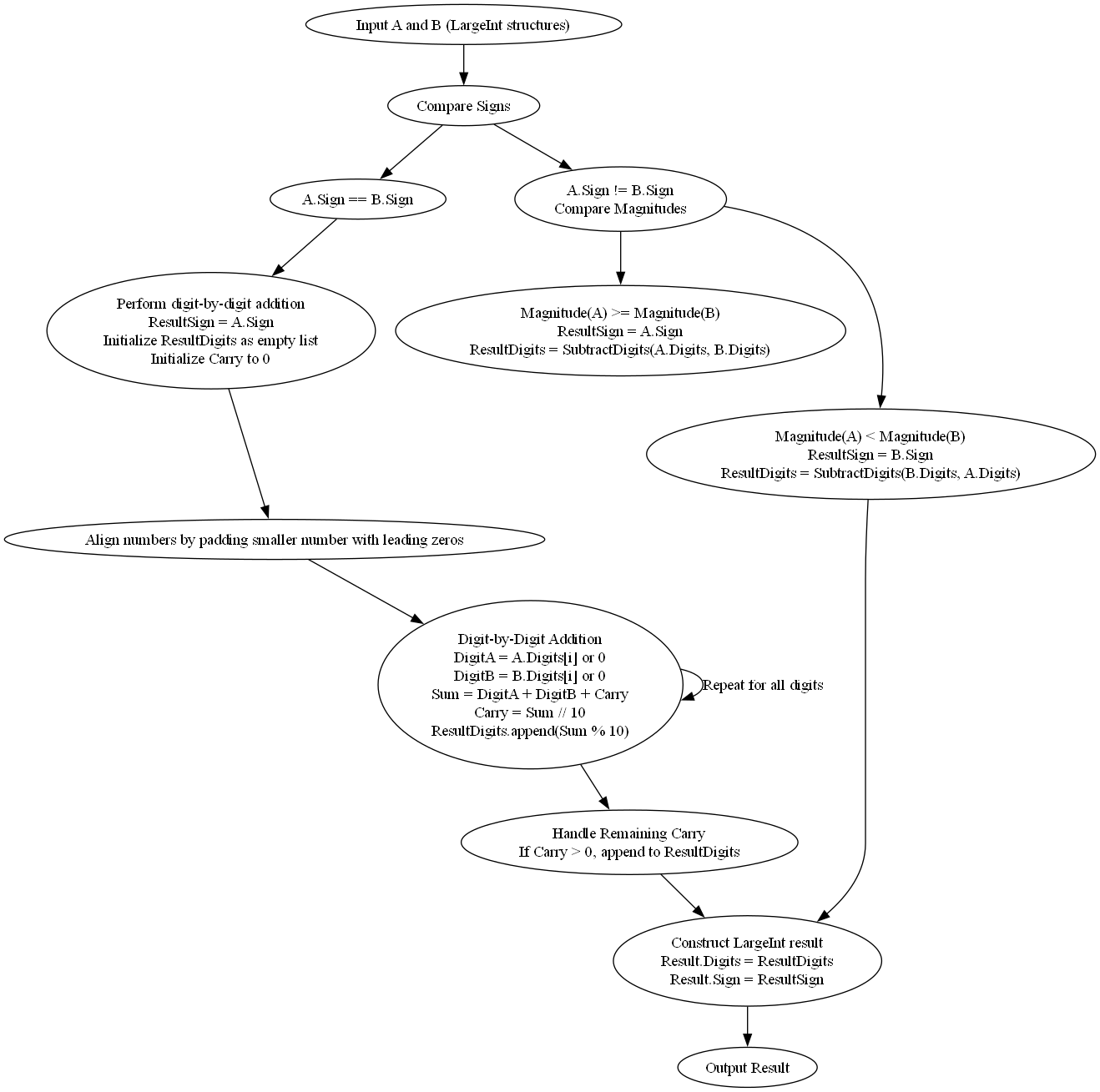
Denominator: LargeInt

For consistency in sign representation the signs will be represented in the numerator and the denominator always positive.

## Algorithms

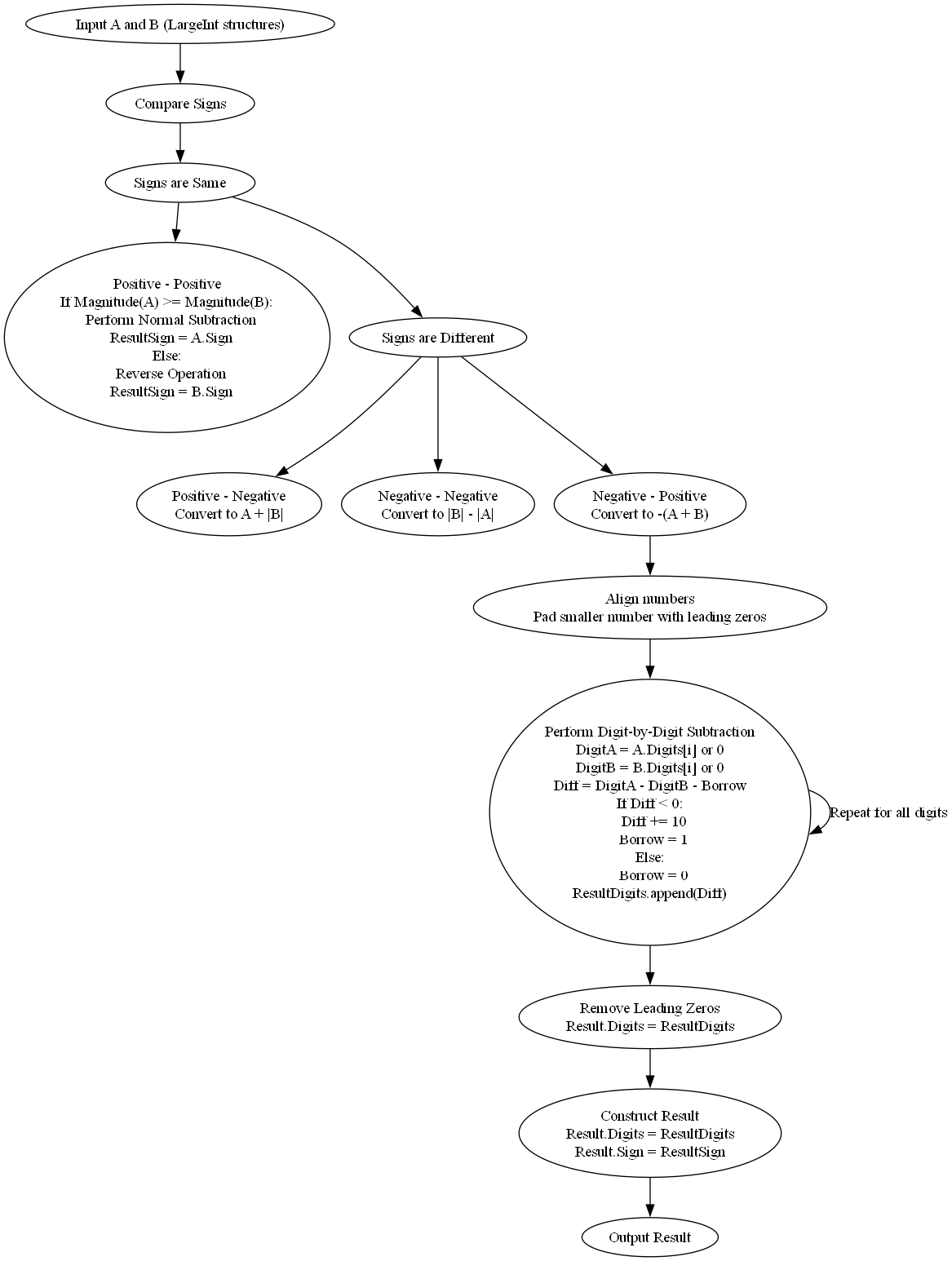
### Addition

1. Input two LargeInt structures: Addends A and B
2. Compare for Signs:
   1. If both integers have the same sign:
      1. Perform digit by digit addition. (Step 3)
      2. The result’s sign will be the same as that of the input integer’s
   2. If the integers have opposite signs:
      1. Treat the operation as subtraction:
         1. Check the magnitude of both integers.
         2. Subtract the smaller magnitude form the larger (using a subtraction algorithm).
         3. The result’s sign is the same as the integer with the larger magnitude.
3. Ensure both numbers are aligned for addition by padding the smaller number with leading zeros if needed
4. Initialize Variables:
   1. Create an empty list ResultDig to store the results in
   2. Initialize carry to 0
5. Perform Digit-by-Digit addition for integers A and B:
   1. Let DigitA be the current digit of A or 0 if it has no digit at the current place value.
   2. And DigitB be the current digit of B or 0 if it has no digit at the current place value.
   3. Compute the sum: Sum = DigitA + DigitB + Carry
6. Handle the remaining carry, Carry = Sum // 10 in long division and then Store Sum modulo 10 in the ResultDig.
7. Construct the result with the sign as well in a LargeInt Object.



### Subtraction

1. Input two LargeInt structures: A (minuend) and B (subtrahend).
2. Check signs:
   1. If positive – positive, for A – B, perform normal subtraction, if A > = B, else for A < B, then reverse the operation and negate the result
   2. If positive – negative, for A – (-B), then convert to addition and it becomes A + |B|
   3. If negative – negative, for (-A) –(-B), then it becomes |B|-|A|
   4. If negative – positive, for (-A) – B, convert to addition with a negative result, -(|A| + |B|)
3. Ensure both numbers are aligned for subtraction by padding the smaller number with leading zeros if needed.
4. Initialize the Borrow to 0 and ResultDig
5. Perform Digit-by-Digit subtraction of the digits in the minued (A) and subtrahend (B).
   1. Let DigitA be the current digit of A or 0 if it has no digit at the current place value.
   2. And DigitB be the current digit of B or 0 if it has no digit at the current place value.
   3. Compute the difference: Diff = DigitA – DigitB – Borrow
      1. If Diff is negative then Diff = Diff +10, and set borrow to 1, or else Borrow to 0
6. Store Diff in ResultDigits with the sign as well in a LargeInt Object



### Multiplication

1. Input two LargeInt structures: multiplicand (X) and multiplicator (Y).
2. Check Signs:
   1. If both Integers entered have the same sign, then the result will be a positive.
   2. If both integers entered have different signs, then the result will be negative.
3. In parallel perform Digit-by-Digit multiplication of all numbers for faster computation, through grid multiplication then add it and construct the result.

### Division and Modulo

1. Input two LargeInt Structures: (X) and (Y).
2. Check Signs:
   1. If both Integers entered have the same sign, then the result will be a positive.
   2. If both integers entered have different signs, then the result will be negative.