2WF90 - Algebra for Security

Software Assignment 1

Group 36

Hoang Nam Mai
1959190

Vinh Nguyen
1234567

Khoi Nguyen
1979388

Abdifitaah Mohamed Hussein
1493086

September 23, 2024

Contents

1 Python Version

The submitted software is written in **Python 3.12.3**.

2 Purpose of the Software

The purpose of this software is to perform various integer and modular arithmetic operations on large integers. These include addition, subtraction, multiplication (using both primary school and Karatsuba methods), extended Euclidean algorithm, and modular operations like inversion and reduction. The software is designed to handle arbitrarily large integers, represented in various radices, and ensures the correct handling of different edge cases.

3 Approach and Methodology

The approach taken by the software is based on implementing multi-precision integer arithmetic and modular arithmetic. The operations are performed digit by digit. In case of addition or subtraction, if the sum exceeds the base, the extra is carried over to the next significant digit. Similarly, borrowing is applied if necessary for subtraction. For multiplication,

The following operations are supported:

3.1 Integer Arithmetic

Integer arithmetic refers to performing arithmetic operations on whole numbers (integers), which can be positive, negative, or zero. In the context of this software, integer arithmetic operations include addition, subtraction, and multiplication. These operations are performed directly on the integers, without any wrapping around or restriction by a modulus, allowing results to extend across the full range of integer values. The following sections explain how integer addition, subtraction, and multiplication are implemented in the submitted software, including both the primary school and Karatsuba methods for multiplication.

- **Addition**: Adding two integers digit by digit, carrying over if the sum exceeds the base.
- **Subtraction**: Subtracting two integers digit by digit, with borrowing applied when needed.
- **Multiplication (Primary School Method)**: We use simple multiplication algorithm, which is digit by digit. Both x and y are represented as strings, at a radix base from 2 to 16. At first, we try to convert the value of x and y into base 10. And then, we create a boolean variable $isNegative$ to check whether the product is negative or not based on the first characters of strings x and y. We then reverse both x and y to make the calculation easier. We create a list containing elements. We multiply each number in x to each number in y, and then locate the results in appropriate positions in the list. By primary school method, we add the values

based on the positions like doing the calculation on paper. During the multiplication process, each product of two digits may result in a multi-digit number. Any "carry" from one digit multiplication is propagated to the next higher position in the result array. After multiplying and placing each product in the list, a pass is made over the entire result to ensure that any carry values are correctly added to the adjacent higher place values. Finally, we remove all the extra 0 values at the end of the list, joining the list as a string and reversing it, adding '-' if necessary, and then we can get the correct result.

- **Multiplication (Karatsuba Method)**: Similar to primary school method, we also do some first steps (converting to base 10, checking negativity). For the base case of Karatsuba, if x and y have a single digit, the product can be calculated directly. Otherwise, we will get the longest length between two inputs, and then split two inputs into halves of that length. We denote those variables by $highX$, $lowX$, $highY$, $lowY$.

  For example, the chosen length is n, so we have:

  $$x = highX \cdot (10^n) + lowX$$

  $$y = highY \cdot (10^n) + lowY$$

  Therefore,

  $$\begin{aligned} x \cdot y &= (highX \cdot 10^n + lowX) \cdot (highY \cdot 10^n + lowY) && (1)\\ &= (highX \cdot highY) \cdot (10^n)^2 && (2)\\ &\quad + [(highX + lowX) \cdot (highY + lowY) && (3)\\ &\qquad - (highX \cdot highY) - (lowX \cdot lowY)] \cdot 10^n && (4)\\ &\quad + lowX \cdot lowY && (5) \end{aligned}$$

- **Multiplication Primary School Method vs Karatsuba Method**:

  - Primary School Method: The number of elementary operations is proportional to the square of the number of digits (O(n^2)). Although it is simple, it is not inefficient for very large numbers.
  - Karatsuba Method: Reduces the complexity to approximately $O(n^{1.585})$, making it more efficient for larger numbers. This method uses divide-and-conquer, splitting numbers into smaller components and combining results recursively.

- **Extended Euclidean Algorithm**: The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm that not only computes the greatest common divisor (GCD) of two integers a and b, but also finds coefficients x and y such that:

  $$a \cdot x + b \cdot y = \gcd(a, b)$$

  This equation is known as Bézout's identity, where x and y are integers, and the values of x and y are the coefficients found by the algorithm. The Extended Euclidean Algorithm is particularly useful in solving linear Diophantine equations and in modular arithmetic (e.g., finding modular inverses).

The Python function `gcdExtended` implements the Extended Euclidean Algorithm recursively. The base case occurs when $a = 0$, at which point the GCD is b, and the coefficients are $x = 0$ and $y = 1$.

For the recursive case, the algorithm computes the GCD of the modulus of b and a, and then updates the coefficients x and y using the results of the recursive call. The quotient of the division of b by a is used to update the coefficients:

$$x = y_1 - \left(\left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right)$$

$$y = x_1$$

where x_1 and y_1 are the coefficients from the recursive call, and $\left\lfloor \frac{b}{a} \right\rfloor$ is the integer division of b by a. The final result is the GCD along with the coefficients x and y.

3.1.1 Arbitrary-Base Addition and Subtraction Algorithms

The implemented algorithms facilitate addition and subtraction of integers in arbitrary bases ranging from 2 to 16. They adeptly handle both positive and negative numbers represented as strings. A pivotal aspect of these algorithms is the avoidance of explicit arithmetic operations; instead, they rely on index manipulation and logical comparisons to ensure accuracy across various bases.

3.1.1.1 Key Components

- **Digit Mapping**: For bases exceeding 10, digits beyond '9' are represented using uppercase letters ('A' for 10 through 'F' for 15). A function, `create(base)`, generates a list of valid digits for the specified base, enabling the translation between character representations and their numerical indices.
- **Padding**: To streamline computations, input strings are padded with leading zeros using the `padding(x, y)` function. This ensures both numbers have equal lengths, facilitating digit-wise operations without index mismatches.
- **Sign Handling**: The algorithms first determine the sign of each operand by checking for a leading '-' sign. Based on the combination of signs (both positive, both negative, one positive and one negative), the algorithms decide whether to perform addition or subtraction of magnitudes, and appropriately assign the sign to the result.

3.1.1.2 Addition Algorithm

When both operands share the same sign, their magnitudes are added:

1. Initialize a carry variable to zero.
2. Iterate through each digit from right to left (least to most significant):
   - Retrieve the numerical indices of the current digits from the digit list.
   - Compute the sum of these indices along with any existing carry.

    - If the sum equals or exceeds the base, subtract the base to obtain the new digit index and set carry to one; otherwise, set carry to zero.
    - Append the corresponding digit to the result.

3. After processing all digits, if a carry remains, prepend it to the result.

4. Remove any leading zeros and assign the appropriate sign based on the operands.

3.1.1.3 Subtraction Algorithm

For operands with differing signs or when performing magnitude-based subtraction:

1. Determine which operand has the larger magnitude using the `compare_numbers` function.

2. Initialize a borrow variable to zero.

3. Iterate through each digit from right to left:

    - Retrieve the numerical indices of the current digits.
    - Adjust the subtrahend digit by adding any existing borrow.
    - If the minuend digit is less than the adjusted subtrahend digit, add the base to the minuend digit and set borrow to one; otherwise, set borrow to zero.
    - Compute the difference and append the corresponding digit to the result.

4. After processing all digits, remove leading zeros and assign the appropriate sign based on the comparison of magnitudes.

3.1.1.4 Important Observations

These algorithms rely on index manipulation and logical comparisons instead of direct arithmetic operations. By processing digits based on their positions in the digit list (`basenum`), the methods ensure accurate addition and subtraction across different bases. This technique maintains precision without the need for built-in arithmetic functions and effectively handles carries, borrows, and sign management.

3.2 Modular Arithmetic

Modular arithmetic is a system of arithmetic for integers where numbers "wrap around" upon reaching a certain value—the modulus. The following sections explain how modulus reduction, modulus subtraction, and modulus multiplication are performed in the submitted software.

- **Reduction**: The modulus reduction operation finds the smallest non-negative integer congruent to a given integer x modulo m. In other words, given an integer x and a modulus m, the result of the modulus reduction is r, where:

  $$r = x \mod m$$

  The Python implementation checks if x is negative or positive. For non-negative x, the code repeatedly subtracts the modulus from x until $x < m$. For negative x, it repeatedly adds m until $x \geq 0$.

- **Addition**: Modulus addition is an arithmetic operation that computes the sum of two integers x and y within a modular system, ensuring the result remains within the range defined by the modulus m. The result of modulus addition is the smallest non-negative integer congruent to $x + y$ modulo m:

  $$z = (x + y) \mod m$$

  In the software implementation, the function first adds the two integers x and y. If the sum exceeds the modulus, it repeatedly subtracts the modulus from the result until the value is within the valid range. The final result is the last valid integer before the sum becomes negative, ensuring a non-negative value within the bounds of the modulus.

- **Subtraction**: Modulus subtraction calculates $z = (x - y) \mod m$, where x, y, and m are integers. The subtraction process is performed by first calculating $z = x - y$, then applying modulus reduction to ensure the result is within the valid range:

  $$z = (x - y) \mod m$$

  If the result becomes negative, the function performs additional steps to ensure the smallest non-negative integer is returned.

- **Multiplication**: Modulus multiplication computes the product of two integers x and y modulo m:

  $$z = (x \times y) \mod m$$

  In the Python implementation, both integers are first reduced modulo m, and then their product is computed. Finally, the result of the multiplication is reduced modulo m to ensure the correct result.

- **Inversion**: Modular inversion refers to finding the modular inverse of a number x modulo m, which is an integer x^{-1} such that:

  $$x \cdot x^{-1} \equiv 1 \pmod{m}$$

  This is particularly useful in cryptography and number theory, where modular inverses are needed in operations like modular division. The modular inverse exists if and only if x and m are coprime, i.e., $\gcd(x, m) = 1$.

  The Python function `inversion` uses the **Extended Euclidean Algorithm** to compute the modular inverse. If the GCD of x and m is not 1, the inverse does not exist, and the function returns an error.

  The algorithm proceeds by initializing variables and performing integer division and modulus operations iteratively. At each step, it updates the coefficients that correspond to the Bézout's identity:

  $$x \cdot x_0 + m \cdot y_0 = 1$$

  Once the GCD is found to be 1, the coefficient x_0 is the modular inverse of x modulo m. If the result x_0 is negative, it is adjusted by adding m to ensure the result is a positive integer.

3.3 Mathematical Description

Let $x = \sum_{i=0}^{n} x_i b^i$ and $y = \sum_{i=0}^{n} y_i b^i$ represent two integers in base b. The addition or subtraction of two numbers is performed by calculating:

$$z_i = x_i \pm y_i + c_i$$

where c_i is the carry from the previous digit. If $z_i \geq b$, the excess is carried to c_{i+1}.

For multiplication, the primary school method involves:

$$z = x \times y = \sum_{i=0}^{n} x_i \sum_{j=0}^{m} y_j b^{i+j}$$

Karatsuba's method is a more efficient approach:

$$x \times y = (x_1 \times y_1) \times B^{2m} + ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) \times B^m + x_0 y_0$$

4 Limitations

The software is designed to operate within the restrictions of 32-bit arithmetic. Operations on integers larger than this bit size are split into smaller parts. The execution time is limited to 5 seconds per exercise, which may pose challenges for very large inputs.

5 Illustrative Examples

5.1 Addition Example

For example, adding two large integers in base 3:

$$x = 1020_3, \quad y = 1102_3$$

The result will be:

$$z = 2122_3$$

5.2 Edge Case: Modular Inversion

An edge case involving modular inversion with modulus 0 will correctly output `null`.

6 Contributions

- Hoang Nam Mai: Do Multiplication (Primary School Method) and Multiplication (Karatsuba Method)
- Vinh Nguyen: Do Modular Reduction and Modular Addition
- Khoi Nguyen: Do Modular Subtraction and Modular Multiplication
- Abdifitaah Mohamed Hussein: Do Addition and Substraction
- We continuously check others' commits while working on our own part, and do Extended Euclidean Algorithm and Modular Inversion together.q

7 References

- Lecture notes from 2WF90 Algebra for Security, Eindhoven University of Technology.