Discrete Mathematics
Department of Computer Science
Ashoka University

Modular arithmetic of long integers

Aman Antil
Aryan Sharma
Asheesh Kumar Singh
Saravana Chilamakuri

22-04-2020

Contents

1. Problem description

We are designing a software for doing Modular arithmetic on large numbers. The software aims to perform addition, subtraction, multiplication, exponentiation and division, i.e., perform the following operations on 1000 digit integers, i.e., numbers in the range 10^{1000}.

If the input is a, b and c, the following operations will be done:

$$(a + b) \% c$$
$$(a - b) \% c$$
$$(a \times b) \% c$$
$$a^b \% c$$
$$\frac{a}{b} \% c$$

where % is the the symbol for modulo.

The code is on the Github repository **https://github.com/aryansharma1323/DM_Project**

2. Design of solution

The code has been written in Python 3.8 and has uses Command-line interface for returning the output and another file: numbers.txt as input, which is created by the user in the same folder and will contain exactly four lines, first containing number and remaining three containing the three numbers.

We divided our tasks into several files then copy-pasted all the operations in a single file: **Project.py**. The other files in the repository contain the specific codes for the operations for the ease of reading the code. They use some functions such as isGreater or simplest_form which are stored in project.py

2.1 numbers.txt

This file is the input file of the program and contains exactly four lines.

The first line contains a symbol for the operation that is to be performed by the program. The supported operations are addition (+), subtraction (-), multiplication (*), division (/) and exponentiation ($\wedge$ or Shift + 6)

The first line will contain the symbol without the bracket.

The next three lines will contain the numbers a, b and c respectively. There will be exactly 4 lines in the file. Even if the fifth line is empty, the program will return an error.

Example: The input.txt file contains the following:

```
+
1000
2000
70
```

This means that the program will execute $(1000 + 2000)\%70$ and print the output, i.e, 60 on the command line.

2.2 How to Run

To begin with, please make sure to have Python 3 installed by writing the following line in Command Prompt.

```
python --version
```

Replace python with python3 on Mac or Linux. If it returns a version number, python is installed on the system and we can continue otherwise please download python 3.8 and run the above code.

Once python3 is installed, write your input in numbers.txt in the format explained above and save the file. The code can run now.

To run the code, open the folder containing project.py and numbers.txt in Terminal/ Command Prompt and execute the following line:

```
python project.py
```

python needs to be replaced with python3 on Linux or Mac.

The code might take some time depending upon the size of the input to compute the result and shows the result in the Terminal.

3. Solution

The project code can be found in https://github.com/aryansharma1323/DM_Project. We have mentioned the pseudocodes of some operations. We have used logic of some functions from the internet, which are cited in Bibliography Section.

3.1 Number to binary conversion

The code of this is present in the project.py file. The following code has been used to convert the entered number into binary code. We used the logic used in this code to write our own code for converting a decimal number to Binary.

```
def dec2bin(snum):
    n = int(snum)
    bin_s = []
    for b in range(n.bit_length()):
        cur_bit = (n >> b) & 1
        sbit = chr(ord('0') + cur_bit)
        bin_s.append(sbit)
    return ''.join(reversed(bin_s))
```

This idea of the code was taken from StackOverflow and since the requirement is to deal with large integers, certain modifications were made to match the requirements. The function toBinary() in project.py has used logic from this code. It takes a string containing a decimal number and returns a string containing binary equivalent of the number.

The right shift operator '»' right shifts m times and divides the number by 2^m.

The '&' operator does and operation on bit representation of number » n and 1.

This will return the remainder when number is divided by 2^m

Initially, the function takes the number, which is of the string form and converts it in to integer form. There is a function bit_length() which takes the number and returns the length of the bit representation of the number. Since the requirement is to find the binary representation of the number, the for loop is run as per the length of the bit representation of the integer. If the condition ((num » m) & 1) is satisfies, then 1 is added to the initially empty binary and if it does not, then 0 is added to the binary. This process is repeated bit_length() times and thus, the binary is formed.

3.2 Addition

The actual code is present in addition.py file and the function is also present in the main file project.py file. The following pseudo code has been used to take two large integers *a* and *b* and return the sum of the two integers which were converted into strings. The time complexity is Time Complexity = O(max(log a, log b)) where log is taken in base 10 or O(n) where n is maximum number of digits in a and b (assuming both have nearly same digits). Here, log a and log b denote the number of digits in a and b respectively.

```
procedure addition(a, b):
    a = reverse(a)
    b = reverse(b)

    iterations = max(sizeOf(a), sizeOf(b))
    output = ""  # Empty String, stores output
    carry = 0

    for i in range(0, iterations - 1):
        s = 0

        if(i >= sizeOf(a) - 1):
            s = b[i] + carry
        else if(i >= sizeOf(b) - 1):
            s = a[i] + carry
        else:
            s = a[i] + b[i] + carry

        if(s >= 10):
            carry = 1
        else:
            carry = 0

        s = s % 10
        output = String(s) + output

    return output
```

The code initially takes the integers which were earlier converted into string and reverses it. The reverse has been done to make the calculations easier.

The technique used to write the code is the ordinary pen and paper method wherein the addition begins from the right hand side. The addition is done and if the sum of the numbers of the i-th place is less than ten, then the concept of carry has been used where the carry is then added to the (i - 1)th place and the remainder using % will remain in the first place. The same process is repeated 'iterations' times. The integer 'iterations' variable as been determined as the maximum of size of a and b since one number can have a different size when compared to the other number. The output has been stored in the output variable which was initially an empty string. The function returns this output variable as the sum of the two integers which were converted in to strings. This function considers almost all cases and works in almost all the corner cases.

The final addition (a + b) % m is done using this method and another method: mod(x, y), which returns x mod y. Addition is simplified by using (a + b) mod c = (a mod c + b mod c) mod c

3.3 Subtraction

The actual code is present in subtraction.py file and the function is also present in the mail file project.py. The following pseudo code has been used to take two large integers *a* and *b* and return the difference of the two integers which were converted into strings. The time complexity is Time Complexity = O(n) where n is the number of digits in a and b assuming a and b have same order of number of digits. It can also be written as: Time complexity = O(max(log a, log b)) where log is taken in base 10, n is the number of digits in *a* and *b* (assuming both have nearly same digits). Also, log a and log b denote the number of digits in *a* and *b* respectively.

```
procedure subtraction(a, b):
    a = reverse(a)
    b = reverse(b)

    iterations = max(sizeOf(a), sizeOf(b))
    output = ""  # Empty String, stores output
    borrow = 0

    if(a >= b):
        for i in range(0, iterations - 1):
            x = (i < sizeOf(a))? a[i] : 0
            y = (i < sizeOf(b))? b[i] : 0

            diff = x - y
            if(diff < 0):
                diff += 10
                borrow = 1
            else:
                borrow = 0
            output = diff + output

        output = trim_extra_zero(output)

        return output

        else:
            for i in range(0, iterations - 1):
                x = (i < sizeOf(a))? a[i] : 0
                y = (i < sizeOf(b))? b[i] : 0

                diff = y - x
                if(diff < 0):
                    diff += 10
                    borrow = 1
                else:
                    borrow = 0
                output = diff + output

            output = trim_extra_zero(output)
            return "-" + output
```

The code initially takes the integers which were earlier converted into string and reverses it. The reverse has been done to make the calculations easier.

The technique used to write the code is the ordinary pen and paper method wherein the subtraction begins from the right hand side. The subtraction is done and if the difference of the numbers of the 1st place is greater or equal to zero, then that is the first number of the output and if the it is less than zero, then the concept of borrow has been used where the borrow is then added to the 1st place and one is subtracted from the tens place. The same process is repeated 'iterations' times. The integer 'iterations' variable as been determined as the maximum of size of a and b since one number can have a different size when compared to the other number. The output has been stored in the output variable which was initially an empty string. The function returns this output variable as the difference of the two integers which were converted in to strings.

The modular subtraction is done using (a - b) mod c = (a mod c - b mod c) mod c equation.

3.4 Multiplication

The actual code is present in multiplication.py file and is also present in the email file project.py. The following pseudo code has been used to take two large integers a and b and return the difference of the two integers which were converted into strings. The time complexity is Time Complexity $= O(n^(log3)$ where log is in base 2 which is nearly equal to $O(n^1.59)$, n is the number of digits in a and b (assuming both have nearly same digits).

```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
    return num1*num2

  calculates the size of the numbers
  m = max(size_base10(num1), size_base10(num2))
  m2 = m/2

  split the digit sequences about the middle
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)

  3 calls made to numbers approximately half the size
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
  return (z2 * 10 ^ (2 * m2)) + ((z1 - z2 - z0) * 10 ^ (m2)) + (z0)
```

The code takes the integers which were earlier converted into string and multiplies them.

The technique used to write the code is the Karatsuba method wherein the numbers a and b are The function returns this output as the product of the two integers which were converted in to strings. This function considers almost all cases and works in almost all the corner cases.

The modular multiplication is done using (a * b) mod c = (a mod c * b mod c) mod c

3.5 Divison

The code for division uses Long division method for calculation of remainder and quotient. We have created modulo_division(a, b) function which takes a and b and returns a tuple containing the quotient and remainder when these two numbers are divided.

In the operation (a / b) mod c, we failed to find the actual result using the modular inverse, as we had been reading about. Instead, we have found the quotient of a / b and then returned quotient mod c as output. This method uses repeated subtraction

3.6 Modular exponentiation

The actual code is present in exponentiation.py file and is also present in the file project.py. The logic of exponentiation has been used from the fast-exponentiation method explained in Khan Academy

The code is divided into multiple functions- power_two(a, b, c) which calculates a^b mod c when b is a power of 2. The other function, power(a, b, c) breaks b into powers of 2 and calls power_two for calculation.

Example:

```
Computation of a^b mod c where a = 19, b = 5 and c = 7 is done in the following way:

a^b mod c = 19^(4 + 1) mod c
= (19^4 * 19) mod 7
= (19^4 mod 7 * 19 mod 7) mod 7   ---(1)

19^4 mod 7 is calculated in power_two as follows:
19^4 mod 7 = (19^2 mod 7 * 19^2 mod 7) mod 7
           = ((19 mod 7 * 19 mod 7) mod 7 * (19 mod 7 * 19 mod 7) mod 7) mod 7
           = ((5 * 5) mod 7 * (5 * 5) mod 7) mod 7
           = (4 * 4) mod 7
           = 2

Now, we can substitute this value in equation 1:
= (2 * 5) mod 7
= 10 mod 7
= 3

The program returns 3 as the output
```

4. Bibliography

1.“Large Integer Arithmetic.” Analysis of Algorithms: Lecture 20,

faculty.cse.tamu.edu/djimenez/ut/utsa/cs3343/lecture20.html

2. “Fast Modular Exponentiation (Article).” Khan Academy, Khan Academy, www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation

3. Leigin, Daan. "Divison and Modulus for Computer Scientists." University of Utrecht, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/divmodnote-letter.pdf

4. Dosunmu, Olawale. “Karatsuba Algorithm (for Fast Integer Multiplication).” OpenGenus IQ: Learn Computer Science, OpenGenus IQ: Learn Computer Science, 31 Jan. 2020, iq.opengenus.org/karatsuba-algorithm/

5. Humayun, Zaid HumayunZaid. “Karatsuba Algorithm in Python.” Code Review Stack Exchange, 1 Mar. 1967,

codereview.stackexchange.com/questions/165215/karatsuba-algorithm-in-python/165220?newreg=9dfce4873d3948c69a2

a691e71b7c2fe

6. SteffSteff 3155 bronze badges, et al. “Function That Takes a String Representing a Decimal Number and Returns It in Binary Format.” Stack Overflow, 1 Aug. 1965, stackoverflow.com/a/33860928/13225678

7. “Modular Arithmetic.” Number Theory - Modular Arithmetic, crypto.stanford.edu/pbc/notes/numbertheory/arith.html

5. List of individual contributions

1. Aman Antil - Wrote the code for addition of long integers and helped in writing the report

2. Aryan Sharma - Wrote the code for multiplication, exponentiation and division of long integers

3. Asheesh Kumar Singh - Helped in formation of logic for Modulo and division operations and also in optimization and debugging of code done by Aman and Saravana

4. Saravana Chilamakuri - Wrote the code for subtraction of long integers and the report