

2WF90 Algebra for security
Number Theory programming assignment

Bogdans Afonins



September 23, 2017

Contents

1	Preface	3
2	Setup	3
3	Input and output	3
4	Program	4
4.1	Number representation	4
4.2	Subroutines	4
4.3	Addition	6
4.4	Subtraction	7
4.5	Naive multiplication	8
4.6	Karatsuba multiplication	8
5	Contribution	10

1 Preface

The following document describes the aspects of the algorithms presented in the provided code. To be more precise, the principle behind each required operation, namely addition, subtraction, naive multiplication and Karatsuba's multiplication, is explained using references from the lecture notes and by giving example pseudocodes. We also present the auxiliary functions that are used in the algorithms. Moreover, in this document you can find not only contents about the programming part of the assignment such as limitations or installation manual but also information about work distribution among the group.

Despite the fact, that the actual program is written in the Java programming language, we do not stick to the syntax of the language within document. Instead, every code sample provided here should be considered as a pseudocode and not as Java or any other existing programming language. The pseudocode follows the basic rules of programming languages in general, it has well-known primitive operations, data structures, statements and control sequences, like arrays, lists, basic operations, **if-else** statements, **for**, **while** loops and so on. We will give comments on non-trivial parts of the provided samples in this document when necessary.

All the references indicated in the following text are collected in the end of the document and can be found in the last section of the document, see 5.

2 Setup

The program is written in the Java programming language, so we assume that the reviewer has JDK 1.8 installed on his computer, see [1]. We provide the sequence of steps that has to be executed in a terminal of your choice to compile the source code and run the program.

```
> cd working directory
> javac *.java
> java Main inputfile outputfile
```

Where **inputfile** is the file containing the test cases to be used and **outputfile** is the file where the results must be written.

3 Input and output

We assume that the input file has the structure as shown below. Also, we expect the input cases to be correct, namely both numbers must have radix **b** and **operation** can take values only within the set {**add**, **subtract**, **multiply**, **karatsuba**}. The program is not handling invalid inputs and in case of providing such corrupted input the outcome of the program is unpredictable. The last line with the answer is optional.

```
[radix] b
[operation]
[x] first number
[y] second number
# [answer] answer
```

The program outputs the result of each operation to a file that is specified before running the program, see 2. Output is formatted to give complete overview of the performed operation. First, the name of the operation is printed on the first line. A name can take any values from {**ADD**, **SUBTRACT**, **MULTIPLY**, **KARATSUBA**}. It is followed by two lines listing the numbers on which the operation is performed. Also, if the correct result is known and specified in the input file, then it is also present in the output file, otherwise this line is skipped. We also list the result calculated by the program right after the known result. The last 3 lines list the number of primitive operations performed during execution of the program, $n, m, k \in N$. The example of the output file is shown below.

[operation]	operation to be performed
[x]	first number
[y]	second number
[known result]	known correct result
[calculated result]	result calculated by the program
[nr. additions]	n
[nr. subtractions]	m
[nr. multiplications]	k

4 Program

This section is indented to describe the main details and features of the program.

4.1 Number representation

Numbers are of the main importance in the assignment, so we model them in a specific way which is explained in this section. First of all, as mentioned in the introduction, we do not stick to any specific programming language, but give a high level idea of the program, while still explaining critical and important parts. We define a number within this document as a custom, user defined type, that can have a certain number of publicly available fields, like `isPositive` which returns a boolean value indicating whether the number is positive and can be accessed with the following syntax `number.isPositive`. The number type can also encapsulate functions that can be called in the following way `number.some_function()`. We also emphasize the fact that the number type copies behaviour of a standard data structure, namely an array. This means that the type has a random access and the syntax `number[i]` is valid and returns a word that is stored at index i in the number `number`. Any number can be seen as an array of words, see [2] chapter 1.2, such that the least significant digit is at index 0 and the most significant digit is at index `number.length` - 1. For example, the number 12345 in the decimal representation can be seen as [5,4,3,2,1], `number[0]` = 5 and `number[4]` = 1. The number *dcba* in the hexadecimal representation can be seen as [10,11,12,13]. Later in the document we provide helper functions that allow us to keep words as decimal numbers, see 2 and 3. Once a new number is needed, it can be created using the following syntax `(number(n))`, where n is the length of the number.

4.2 Subroutines

In this section we describe the helper functions of the program that are referenced throughout the document and will have an important role further in the text. These functions are vital and provide convenient approaches to deal with our representation of numbers that is explained in the previous section.

We first define the `rebaseLeft` function that is responsible of extending the number with additional zeros in the part of the number where the least significant words reside. In other words we are increasing the number by $base^{shift}$ times. This procedure is used to acquire a copy of the existing number with the new size, to extend the number or fill the least significant bits with 0's. Note, that further in the text we also reference the `rebaseRight` function, which is very similar to `rebaseLeft`, but inserts 0's to the right side of the number.

Data: $x, n, shift$

Result: a new number of length n and shifted by $shift$ zeros from the left.

```
1  $new\_number \leftarrow number(n + shift);$ 
2 for  $i \leftarrow 0$  to  $n$  do
3   |  $new\_number[i + shift] \leftarrow x[i];$ 
4 end
5 return  $new\_number;$ 
```

Algorithm 1: `rebaseLeft`

In the program we store each word as a decimal number, e.g. 'a' is mapped to 10 and so on, we have to define a way to transform input numbers into their analogs in base 10 and vice versa to give meaningful output.

For the '`string to number`' transformation we use the ASCII codes of each character in the string and simply map to a number within the range.

Data: *str_number*
Result: *number* out of a string

```

1  $n \leftarrow \text{str\_number.length}$ ;
2  $\text{number} \leftarrow \text{number}(n)$ ;
3 for  $i \leftarrow n - 1$  down to 0 do
4    $\text{char} \leftarrow \text{str\_number}[i]$ ;
5    $\text{range} \leftarrow \text{char} - 48$ ;
6   if  $0 \leq \text{char} - 48 \leq 9$  then
7      $\text{number}[n - i - 1] = \text{char} - 48$ ;
8   end
9   else if  $10 \leq \text{char} - 87 \leq 15$  then
10     $\text{number}[n - i - 1] = \text{char} - 87$ ;
11  end
12 end
13 return number;

```

Algorithm 2: fromRawString

For the 'number to string' transformation we define an array **values** that contains all legal characters, namely from 0 to f, and map each word from the number with the corresponding character from **values**.

Data: *x*
Result: string representation of *x*

```

1  $\text{values} \leftarrow ['0', '1', \dots, 'f']$ ;
2  $\text{str} \leftarrow \text{nil}$ ;
3  $n \leftarrow x.\text{length}$ ;
4 if  $x.\text{isPositive}$  then
5    $\text{str} \leftarrow \text{string}(n)$ ;
6 else
7    $\text{str} \leftarrow \text{string}(n + 1)$ ;
8    $\text{str.append}('-')$ ;
9 end
10 for  $i \leftarrow n - 1$  down to 0 do
11    $\text{value} \leftarrow \text{values}[x[i]]$ ;
12    $\text{str.append}(\text{value})$ ;
13 end
14 return str;

```

Algorithm 3: toRawString

In order to be consistent, we also define the **removeLeadingZeros** function that is used in almost every function to remove 0's that do not change the value of the number.

Data: *x*
Result: *x* without any leading zeros

```

1  $i \leftarrow x.\text{length}$ ;
2 while  $i \geq 0$  and  $x[i] == 0$  do
3    $i = i - 1$ ;
4 end
5 if  $i == -1$  then
6    $\text{number}_2 \leftarrow \text{number}(1)$ ;
7    $\text{number}_2[0] \leftarrow 0$ ;
8 else
9    $\text{number}_2 \leftarrow \text{number}(i + 1)$ ;
10  for  $j \leftarrow i$  down to 0 do
11     $\text{number}_2[j] \leftarrow x[j]$ ;
12  end
13 end
14 return  $\text{number}_2$ ;

```

Algorithm 4: removeLeadingZeros

4.3 Addition

Addition is one of the basic operations we had to implement in terms of this assignment. We present the basic primary school method that is presented and analyzed in the lecture notes, see [2]. In lines 1 – 5 we perform initialization of the required types. We set `carry` to 0, since no addition has been performed. Make both numbers to be the same size in order to simplify the index arithmetic. We do that by adding leading zeros where necessary, see 1. Also, create the resulting number that will hold the result of summing `number1` and `number2`. Note, that the resulting number is at most larger by 1 than the longest number that is being added. This is simply because the summation can result in having the carry equal to 1 after summing the most significant digits within the number. The for loop on lines 6 – 14 adds two words residing at the i^{th} position in both numbers, as well as keeping track of the carry. So, each iteration of the loop results in two additions and a single subtraction only if we have to update the carry restore the word in the resulting number to less than the base of the numbers. Before returning the result we trim the leading zeros that might appear in the resulting number by calling the `removeLeadingZeros` function, see 4.

Data: x, y , base

Result: $x + y$

```
1 carry ← 0;
2 n ← max(x.length, y.length) + 1;
3 number11 ← rebaseRight(x, n, 0);
4 number22 ← rebaseRight(y, n, 0);
5 result ← number(n);
6 for i ← 0 to n do
7   | result[i] ← number11[i] + number22[i] + carry;
8   | if result[i] > base then
9     |   | result[i] ← result[i] - base;
10    |   | carry ← 1;
11    | else
12    |   | carry ← 0;
13    | end
14 end
15 return removeLeadingZeros(result);
```

Algorithm 5: add

According the requirements of the assignment, the program must handle addition of both, positive and negative numbers, but the pseudocode provided above can deal only with positive numbers. In order to fulfill the requirements we distinguish between different input cases and explain how the program handles them. See 6 for the extension.

- $x, y > 0$. In this case we can safely execute `add` on x and y without violating any invariants of the algorithm.
- $x < 0$ and $y > 0$. In other words, we are dealing the following expression $(-x) + y$. As it is pointed out in the lecture notes of the course, see [2] chapter 1.3.1, it is a simple subtraction of the form $y - x$. Because of this, we can delegate execution to the subtraction algorithm with the initial call `subtract(y, x, base)`, see 7.
- $x > 0$ and $y < 0$. The logic is exactly the same as in the previous case. We delegate execution to the subtraction algorithm, `subtract(x, y, base)`.
- $x, y < 0$. In this case we dealing with expression of the form $(-x) + (-y)$. Since both number are negative, we can simply add, treating both x and y as positive numbers, and changing the resulting sign, see the lecture notes [2] chapter 1.3.1. So, we make an additional recursive call to the `add` function, `add(x.negate(), y.negate()).negate()`. We make both numbers positive by calling `negate()` on them and then we change the sign of the result.

```
1 if x < 0 and y > 0 then
2   | return subtract(y, x, base);
3 end
4 if x > 0 and y < 0 then
5   | return subtract(x, y, base);
6 end
7 if x < 0 and y < 0 then
8   | return add(x.negate(), y.negate()).negate();
9 end
```

Algorithm 6: extension add

4.4 Subtraction

The very same method used for addition is used for subtraction. We step by step process both numbers starting from the least significant parts (from the right side) and keep track of the carry. The pseudocode for the **subtract** algorithm is presented below, see 7. Lines 1 – 5 are again performing certain initialization operations and the loop on lines 6 – 14 goes through the numbers considering i^{th} word at i^{th} iteration. If the result after subtracting two words is negative, then the resulted digit is being replaced by this digit plus radix and the value 1 is being copied into the carry, otherwise 0. After the result is obtained the leading zeros are erased.

Data: x, y , base

Result: $x - y$

```

1 carry  $\leftarrow 0$ ;
2  $n \leftarrow \max(x.length, y.length)$ ;
3  $number_{11} \leftarrow rebaseRight(x, n, 0)$ ;
4  $number_{22} \leftarrow rebaseRight(y, n, 0)$ ;
5  $result \leftarrow number(n)$ ;
6 for  $i \leftarrow 0$  to  $n$  do
7    $result[i] \leftarrow number_{11}[i] - number_{22}[i] - carry$ ;
8   if  $result[i] < 0$  then
9      $result[i] \leftarrow result[i] + base$ ;
10     $carry \leftarrow 1$ ;
11  else
12     $carry \leftarrow 0$ ;
13  end
14 end
15 return  $removeLeadingZeros(result)$ ;

```

Algorithm 7: subtract

As well as addition, subtraction has several specific cases to properly handle positive and negative numbers. We mention them here and discuss how the program handles such different inputs, as well as present the extension code for the **subtract** function, see 8.

- $x, y > 0$ and $x > y$. We safely execute the **subtract** algorithm.
- $x, y > 0$ and $x < y$. It is preferable that the subtrahend is smaller than the minuend, because it simplifies the algorithm in general. We deal with the following expression $x - y = -(y - x)$. In this case we flip the arguments and change the sign of the resulting number. So, we recall the function again with the initial call **subtract**(y , x).**negate**() .
- $x > 0$ and $y < 0$. This case is equivalent to adding two positive numbers, simply because $x - (-y) = x + y$. We delegate execution to the **add** function with the initial call **add**(x , y).**negate**() .
- $x < 0$ and $y > 0$. The same as in the last case for the **add** algorithm extension, this operation is equivalent to adding x and y and changing the resulting sign. Hence, we make an additional call **add**(x).**negate**() , y).**negate**() .
- $x, y < 0$. $(-x) - (-y) = y - x$, simply change the signs and flip the arguments **subtract**(y).**negate**() , x).**negate**() .

```

1 if  $x > 0$  and  $y < 0$  then
2   return  $add(x, y.negate())$ ;
3 end
4 if  $x < 0$  and  $y > 0$  then
5   return  $add(x.negate(), y).negate()$ ;
6 end
7 if  $x < 0$  and  $y < 0$  then
8   return  $subtract(y.negate(), x.negate())$ ;
9 end
10 if  $x < y$  then
11   return  $subtract(y, x)$ ;
12 end

```

Algorithm 8: extension subtract

4.5 Naive multiplication

In comparison to addition and subtraction, the naive multiplication method is more sophisticated. What is more, the `multiply` algorithm uses `add` as a subroutine. So, lines 1 – 8 are left for performing the basic initialization, that is very similar to what was described before in other algorithms. It is worth noting, that every intermediate result of multiplication is stored in the `intermediate` data structure, which is defined as an array of numbers. In other words, a number at `intermediate[i]` is the result of multiplying i^{th} word of `y` with the whole number `x`. We make `intermediate` to be of size n , because for each word in `y` we get a new entry in `intermediate` and since `y` is of size n we result in having at most n entries. Note, that each intermediate number is of length $n + 1$. This is because multiplication might result in a number that will not fit in one word anymore, hence we have to extend the number to avoid overflow, see [2] chapter 1.3.2. Both loops on lines 9 – 17 are responsible for doing correct bookkeeping of the intermediate results. Calculations there are trivial and explained well in the lecture notes [2]. Lines 18 – 21 sum all the elements in `intermediate` by calling the `add` function. Each number at this stage is shifted by i zeros from the left. This increases the numbers by $base^i$ times and guarantees that the resulting number is of correct length. The rest of the algorithm is left for determining the resulting sign and can be implemented as a simple `xor` of two booleans. If two numbers are both positive or both negative the result of the `xor` operation is `false` and we return a positive number, in any other case the condition will evaluate to `true` and the sign of the number is changed.

Data: $x, y, base$

Result: $x * y$

```

1 carry ← 0;
2  $n \leftarrow \max(x.length, y.length)$ ;
3  $number_{11} \leftarrow \text{rebaseRight}(x, n, 0)$ ;
4  $number_{22} \leftarrow \text{rebaseRight}(y, n, 0)$ ;
5  $intermediate \leftarrow [n]$ ;
6 for  $i \leftarrow 0$  to  $n$  do
7    $intermediate[i] \leftarrow \text{number}(n + 1)$ ;
8 end
9 for  $i \leftarrow 0$  to  $n$  do
10   carry ← 0;
11   for  $j \leftarrow 0$  to  $n$  do
12      $t \leftarrow number_{11}[j] \cdot number_{22}[i] + \text{carry}$ ;
13      $\text{carry} \leftarrow \text{floor}(\frac{t}{base})$ ;
14      $intermediate[i][j] \leftarrow t - \text{carry} * base$ ;
15   end
16    $intermediate[i][n] \leftarrow \text{carry}$ ;
17 end
18  $result \leftarrow intermediate[0]$ ;
19 for  $i \leftarrow 1$  to  $n$  do
20    $result \leftarrow \text{add}(result, \text{rebaseLeft}(intermediate[i], n + 1, i))$ ;
21 end
22 if  $x.isPositive \oplus y.isPositive$  then
23   return  $\text{removeLeadingZeros}(result.negate())$ ;
24 else
25   return  $\text{removeLeadingZeros}(result)$ ;
26 end

```

Algorithm 9: multiply

4.6 Karatsuba multiplication

Karatsuba multiplication is a combination of both multiplication and addition. However, Karatsuba is still completely different from it and way harder to understand. Nevertheless, using this approach results in a smaller number of multiplications that are considered to be slow.

As it can be seen on the very first line of the pseudocode, in provided implementation of this method the sign of the final result is clarified in advance and stored in a boolean variable `negative`. That is because the function makes several additional calls to `add` that expects the numbers to be positive, as well recursively calls itself. The signs of both integers are being set to positive before the start, see 11. The essence of recursive algorithms is a base case. In our case we stop going deeper into the recursive tree once both number contain only a single digit. Lines 2 – 4 check for that case and if the condition evaluates to true, we simply delegate the multiplication operation to the naive multiplication algorithm. In further steps, both numbers will be parsed into equal parts of length $n/2$, where n is the maximum length of the number present or $max + 1$ if it is odd. This is exactly what is being done on lines 5 – 11. On lines 12 – 15, new words, namely `a`, `b`, `c`, and `d`, of length

$n/2$ are created. Then on lines 16 – 21 the **for** loop is used to split the numbers to:

- **a** - high importance half of the first number, **b** - least importance half of the first number
- **c** - high importance half of the second number, **d** - least importance half of the second number

After that the following arithmetical operations are being carried out (the meaning behind these is described in the chapter 1.3.2 of the lecture notes [2]: $(a + b) \cdot (c + d) - a \cdot c - b \cdot d$ (under '.' the karatsuba multiplication is meant, thus resulting in recursive call). When recursion completes, the result is obtained and the sign calculated in advance is adjusted to it with deleting the leading zeros in parallel, see lines 27 – 31.

```

Data:  $x, y, base$ 
Result:  $x * y$  using the karatsuba method
1  $negative \leftarrow x.isPositive \oplus y.isPositive$ ;
2 if  $x.length == 1$  and  $y.length == 1$  then
3   | return  $multiply(x, y, base)$ ;
4 end
5  $n \leftarrow \max(x.length, y.length)$ ;
6 if  $n.isOdd$  then
7   |  $n \leftarrow n + 1$ ;
8 end
9  $half \leftarrow \frac{n}{2}$ ;
10  $number_{11} \leftarrow rebaseRight(x, n, 0)$ ;
11  $number_{22} \leftarrow rebaseRight(y, n, 0)$ ;
12  $a \leftarrow number(half)$ ;
13  $b \leftarrow number(half)$ ;
14  $c \leftarrow number(half)$ ;
15  $d \leftarrow number(half)$ ;
16 for  $i \leftarrow 0$  to  $half$  do
17   |  $b[i] \leftarrow number_{11}[i]$ ;
18   |  $d[i] \leftarrow number_{22}[i]$ ;
19   |  $a[i] \leftarrow number_{11}[i + half]$ ;
20   |  $c[i] \leftarrow number_{22}[i + half]$ ;
21 end
22  $ac \leftarrow karatsuba(a, c, base)$ ;
23  $db \leftarrow karatsuba(b, d, base)$ ;
24  $abcd \leftarrow karatsuba(add(a, b, base), add(c, d, base), base)$ ;
25  $e \leftarrow subtract(subtract(abcd, ac), bd)$ ;
26  $result \leftarrow add(add(rebaseLeft(ac, ac.length, n), add(rebaseLeft(e, e.length, half))), bd)$ ;
27 if  $negative$  then
28   | return  $removeLeadingZeros(result.negate())$ ;
29 else
30   | return  $removeLeadingZeros(result)$ ;
31 end

```

Algorithm 10: karatsuba


```

1 if  $x.isNegative$  then
2   |  $x \leftarrow x.negate()$ ;
3 end
4 if  $y.isNegative$  then
5   |  $y \leftarrow y.negate()$ ;
6 end

```

Algorithm 11: extension karatsuba

5 Contribution

Bogdan Afonins	
----------------	---

References

- [1] Oracle. <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>, Java Development Kit 1.8
- [2] Benne de Weger. *Part 1 - Algorithmic Number Theory*, 2WF90 - Algebra for Security, page 3, version 0.65, 2017, TU/e
- [3] <http://www.asciitable.com>, ASCII table