CmpE 150 Introduction to Computing, Fall 2015
Project 1 Report
CEMAL BURAK AYGÜN, 2014400072

PROBLEM DESCRIPTION

PROBLEM 1

As a teacher, I need to set up a multiplication table that has N rows and N columns, where N is a positive integer. If we think of the table as a rectangle, the diagonal of the rectangle that goes from top left corner to bottom right corner splits the rectangle into two triangular regions. I should leave the upper triangular region empty so that my students can fill in the blanks themselves.

1	2	3	. . .	N
2	4	6	. . .	2*N
3	6	9	. . .	3*N
. . .	. . .	. . .	. . .	. . .
N	N*2	N*3	. . .	N*N

Complete Multiplicaton Table

1	2	3	. . .	N
2	4	6	. . .	2*N
3	6	9	. . .	3*N
. . .	. . .	. . .	. . .	. . .
N	N*2	N*3	. . .	N*N

Diagonal, Upper Triangular Region and Lower Triangular Region

1				
2	4			
3	6	9		
. . .	. . .	. . .	. . .	
N	N*2	N*3	. . .	N*N

Multiplicaton Table with Upper Triangular Region Empty

PROBLEM 2

I need to write a sequence with the following properties:

- It should consist of 4 elements.
- Each element in the sequence should be equal to [the number of digits in the previous element] times [N (the integer in problem 1)].
- Since there is no 'previous element' for the first element of the sequence, I need to assign an integer constant, M, that is used to calculate the first element.

In mathematical form: if $[e_n]$ is the n^{th} term (or element) of the sequence, then

- $[e_1]$ = (the number of digits in M) * N
- $[e_2]$ = (the number of digits in $[e_1]$) * N
- $[e_3]$ = (the number of digits in $[e_2]$) * N
- $[e_4]$ = (the number of digits in $[e_3]$) * N

More generally,

$[e_n]$ = (the number of digits in $[e_{(n-1)}]$) * N , where $n \in Z^+$, $1 \le n \le 4$, and $[e_0]$ = M

PROBLEM SOLUTION

I define a method named **multiplicationTable** for problem 1 and a method named **sequence** for problem 2.

I declare 2 constant integers at the top of my program: **N** is a positive integer used in both of the methods and **M** is an integer used in the sequence() method.

In my main method, I call both of the methods and before calling each of them, I use a System.out.println to print a title about the method, just to make the output look better. Also, it prints the values of N and M in parantheses in the title of sequence() method so that we can see in the output the constants that are used in the sequence.

SOLUTION 1

When we look at the multiplication table in problem 1, we see that there is 1 element in the first row, 2 elements in the second row, 3 elements in the third row, and so on. In other words, the number of columns (or the number of elements) in each row is equal to the row number. We also see that the number of empty cells in each row is equal to the total number of rows minus the row number. We can write such a table by using nested for loops, a total of 3 for loops. (I use brackets ([ ]) for the empty cells.)

- ➔ I define a for loop (let's call it FL1) which determines the total number of rows of the table. I declare an int named **row**, which is the row number, that goes from 1 to N increasing by 1 at every step.

- ➔ In **FL1**, I define a for loop (let's call it FL2) which determines the number of colums in the specific row. I declare an int named **column**, which is the column number, and since the number of columns in each row is equal to the row number, it goes from 1 to row increasing by 1 at every step.
- ➔ In **FL2**, I use a System.out.print which prints **the result of row * column** and adds a **tab space** at the right side of that result.

- ➔ In **FL1**, I define a for loop (let's call it FL3) which determines the number of brackets in the specific row. I declare an int named **bracket**, which is the bracket number, that goes from 1 to N – row increasing by 1 at every step.
- ➔ In **FL3**, I use a System.out.print which prints **[ ]** and adds a **tab space** at the right side of it.
- ➔ In **FL1**, I use a System.out.println in order to go to new row after each row.

SOLUTION 2

We can use a for loop to find the number of digits in a given **positive** integer, a foor loop for a **negative** integer and a for loop to state the size (the number of elements) of the sequence. So, I use nested for loops, a total of 3 for loops.

- ➔ I declare an int named **previousNumber**, which is used to calculate the elements of the sequence, and assign M to it.
- ➔ I declare an int named **number** that is printed as the element of the sequence.
- ➔ I declare an int named **digitNumber** that stores the number of digits in previousNumber.
- ➔ I define a for loop (let's call it FL1) which determines the size (the number of elements) of the sequence. Since the sequence has 4 elements, I declare an int named **size** that goes from 1 to 4.
- ➔ In **FL1**, I set the value of digitNumber to **1** before the **FL2** and **FL3** because we need to calculate the number of digits in previousNumber for every element of the sequence and **FL2** and **FL3** need an 1-valued digitNumber to do it.
- ➔ In **FL1**, I define a for loop (let's call it FL2) which calculates the number of digits in the previousNumber that is **positive**. **FL2** takes previousNumber and divides it by **10** until it

is less than **10**.

➔ In **FL2**, I increase `digitNumber` by **1** at every step. When the **FL2** is completely done, the final value of `digitNumber` becomes equal to the number of digits in the `previousNumber`.

➔ In **FL1**, I define a for loop (let's call it FL3) which calculates the number of digits in the `previousNumber` that is **negative**. **FL3** takes `previousNumber` and divides it by **10** until it is greater than **-10**.

More Explanation for Last 4 Parts (Every black right arrow is a part.)

*Unfortunately, neither **FL2** nor **FL3** can calculate the number of digits in **zero**. To solve this problem, I set the value of `digitNumber` to **1**, which is the number of digits in zero, at the beginning and make the for loops run the number of digits in previousNumber – 1 times. There are 3 cases:*

CASE 1: *When `previousNumber` is zero, neither **FL2** nor **FL3** is effective. In this case, the value of `digitNumber` remains as **1** which is the number of digits in **zero**.*

CASE 2: *When `previousNumber` is a positive integer, only **FL2** is effective. The loop increases the value of `digitNumber` the number of digits in previousNumber – 1 times and the final value of `digitNumber` becomes equal to the number of digits in previousNumber.*

CASE 3: *When `previousNumber` is a negative integer, only **FL3** is effective and it does the same things as **FL2** does in case 2.*

*In **FL2** and **FL3**, we take advantage of a feature of **integer division**. In JAVA, when you divide an integer by **10**, you simply get rid of the most-right digit in the integer.*

➔ In **FL1**, after **FL2** and **FL3**, `number` is updated and its new value becomes the result of `digitNumber` * `N`.

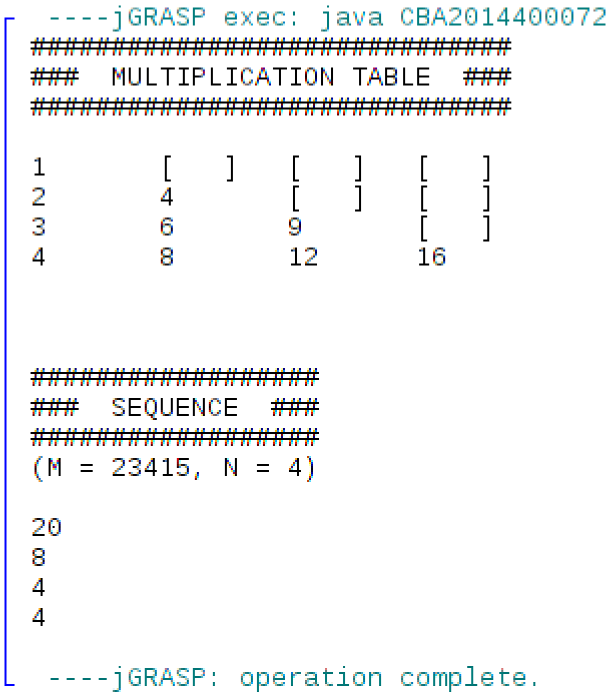
➔ In **FL1**, I use `System.out.println(number)` to print `number` as an element of the sequence and to go to new line.

➔ In **FL1**, I assign `number` to `previousNumber` so that the element of the sequence becomes 'previous element' for the next element.

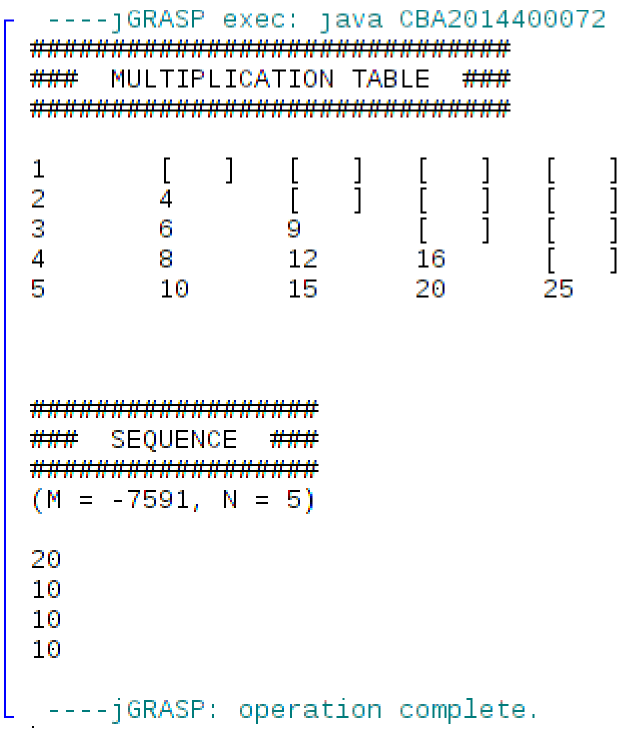
IMPLEMENTATION

```
/*
CmpE 150 Introduction to Computing, Fall 2015
Project 1
CEMAL BURAK AYGÜN, 2014400072
*/

public class CBA2014400072 {

   public static final int N = 4;   // A positive integer constant that is used in multiplicationTable() and sequence() methods
   public static final int M = 23415;   // An integer constant that is used in sequence() method

   public static void main(String[] args) {

      /* Prints a title for the multiplicationTable() method for a better looking output :) */
      System.out.println("################################\n###  MULTIPLICATION TABLE
###\n################################\n");
      multiplicationTable();

      /* Prints a title for the sequence() method and the values of M and N in parantheses for a better looking output :) */
      System.out.println("\n\n\n##################\n###  SEQUENCE  ###\n##################\n(M = "+M+",
N = "+N+")\n");
      sequence();

   }

   /*
   The method below prints a N * N dimensional multiplication table, N is declared at the top of the program.
   It leaves the elements that stay at upper triangular region of the table blank for kids to fill in.
   */
   public static void multiplicationTable() {

      /* The loop below determines the number of rows of the table */
      for(int row = 1; row <= N; row++) {   // row is the row number

         /* The loop below prints the elements in the specific row except the ones that stay at upper triangular side of the table
*/
         for(int column = 1; column <= row; column++) {   // column is the column number
            System.out.print(row * column + "\t");
         }

         /* The loop below prints brackets in the specific row instead of the elements that stay at upper triangular side of the
table */
         for(int bracket = 1; bracket <= N - row; bracket++) {   // bracket is the bracket number
            System.out.print("[   ]\t");
         }

         /* Goes to new row */
         System.out.println();
      }

   }

   /*
   The method below prints a sequence that has following properties:
   # It has 4 elements
   # Each element is equal to [the number of digits in the previous element] times [N, which is declared at the top of the
program]
   # First element uses M, which is declared at the top of the program, as the 'previous element'
   */
   public static void sequence() {

      int previousNumber = M;   // A variable that is used to calculate the elements of the sequence
      int number;   // number is the element of the sequence
      int digitNumber;   // A variable that stores the number of digits in previousNumber

      /* Limits the sequence with 4 elements */
      for(int size = 1; size <= 4; size++) {    // size is the number of elements of the sequence

         digitNumber = 1;   // Sets digitNumber to 1 which is the number of digits in zero

         /* Calculates the number of digits in previousNumber for positive numbers */
         for(; previousNumber >= 10; previousNumber /= 10) {
            digitNumber++;   // After the loop, the value of digitNumber becomes the number of digits in positive
previousNumber
         }

         /* Calculates the number of digits in previousNumber for negative numbers */
         for(; previousNumber <= -10; previousNumber /= 10) {
            digitNumber++;   // After the loop, the value of digitNumber becomes the number of digits in negative
previousNumber
         }

         number = digitNumber * N;   // number gets updated, it becomes [the number of digits in previousNumber] times [N]

         /* Prints number as the element of the sequence and goes to new line */
         System.out.println(number);

         previousNumber = number;   // previousNumber gets updated, number becomes the 'previous element' for the next
element of the sequence

      }

   }

}
```

OUTPUT OF THE PROGRAM



```
 ----jGRASP exec: java CBA2014400072
##############################
###  MULTIPLICATION TABLE  ###
##############################

1       [   ]   [   ]   [   ]
2       4       [   ]   [   ]
3       6       9       [   ]
4       8       12      16



##################
###  SEQUENCE  ###
##################
(M = 23415, N = 4)

20
8
4
4

 ----jGRASP: operation complete.
```



```
 ----jGRASP exec: java CBA2014400072
##############################
###  MULTIPLICATION TABLE  ###
##############################

1       [   ]   [   ]   [   ]   [   ]
2       4       [   ]   [   ]   [   ]
3       6       9       [   ]   [   ]
4       8       12      16      [   ]
5       10      15      20      25



##################
###  SEQUENCE  ###
##################
(M = -7591, N = 5)

20
10
10
10

 ----jGRASP: operation complete.
```

CONCLUSION

I have solved both of the problems correctly. I have used 6 for loops in my program. Solution to problem 2 could have been improved by writing a single for loop (*if possible*) that works for both positive and negative integers, and also for zero.

In addition, I have made some improvements to the program to make the output look better. These are:

- Writing a title before the output of each problem.
- In multiplication table, adding a **tab space** between the elements on the same row rather than a one character space.
- In multiplication table, writing a **[ ]** for missing elements to show students which parts exactly they are supposed to fill in.