**Guided Assignment 3**

Created: 06/03/21 by Tom Lever

Updated: 06/03/21 by Tom Lever

**Problem 1: Multiplication**

*Algorithm and Implementation for Recursive Multiplication*

**package** Com.TSL.RecursiveMultiplicationUtilities;

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* RecursiveMultiplier encapsulates the entry point of this program, which displays an elementary multiplication table,

\* using a recursive multiplication method that does not use Java's \* operator.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/03/21

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

**public** **class** RecursiveMultiplier

{

/\*\* ------------------------------------------------------------------------------------------------------------

\* main is the entry point of this program, which displays an elementary multiplication table, using a recursive

\* multiplication method that does not use Java's '\*' operator. main throws ANotSufficientlyImplementedException

\* if multiplication with a non-positive factor is requested.

\*

\* **@param** args

\* **@throws** ANotSufficientlyImplementedException

------------------------------------------------------------------------------------------------------------ \*/

**public** **static** **void** main (String[] args) **throws** ANotSufficientlyImplementedException

{

// final was removed to avoid "Dead code" warning regarding

// if ((THE\_LOWEST\_FACTOR < 1) || (THE\_HIGHEST\_FACTOR < 1))

**int** THE\_LOWEST\_FACTOR = 1;

**int** THE\_HIGHEST\_FACTOR = 9;

**if** ((THE\_LOWEST\_FACTOR < 1) || (THE\_HIGHEST\_FACTOR < 1))

{

**throw** **new** ANotSufficientlyImplementedException("Exception: Factors must be positive.");

}

**int** product;

**for** (**int** i = THE\_LOWEST\_FACTOR; i <= THE\_HIGHEST\_FACTOR; i++)

{

**for** (**int** j = THE\_LOWEST\_FACTOR; j < THE\_HIGHEST\_FACTOR; j++)

{

product = (i < j) ?

ARecursiveMultiplicationMachine.*multipliesRecursively* (j, i) :

ARecursiveMultiplicationMachine.*multipliesRecursively* (i, j);

System.***out***.print (product + " ");

}

product = (i < THE\_HIGHEST\_FACTOR) ?

ARecursiveMultiplicationMachine.*multipliesRecursively* (THE\_HIGHEST\_FACTOR, i) :

ARecursiveMultiplicationMachine.*multipliesRecursively* (i, THE\_HIGHEST\_FACTOR);

System.***out***.println (ARecursiveMultiplicationMachine.*multipliesRecursively* (i, THE\_HIGHEST\_FACTOR));

}

}

}

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* ARecursiveMultiplicationMachine encapsulates a method that multiplies recursively two integers without using Java's

\* '\*' operator.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/03/21

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

**class** ARecursiveMultiplicationMachine

{

/\*\* -------------------------------------------------------------------------------------------

\* multipliesRecursively multiplies recursively two integers without using Java's '\*' operator.

\*

\* **@param** theFirstInteger

\* **@param** theSecondInteger

\* **@return**

\* **@throws** ANotSufficientlyImplementedException

------------------------------------------------------------------------------------------- \*/

**static** **int** multipliesRecursively(**int** theFirstInteger, **int** theSecondInteger)

{

**if** (theSecondInteger == 1)

{

**return** theFirstInteger;

}

**return** theFirstInteger + *multipliesRecursively*(theFirstInteger, theSecondInteger - 1);

}

}

*Verifying Algorithm for Recursive Multiplication*

1. The Base-Case Question: Is there a non-recursive way out of the algorithm, and does the algorithm work correctly for this base case?

Yes. When theFirstInteger has a value of any positive integer and theSecondInteger has a value of 1, multipliesRecursively returns theFirstInteger.

1. The Smaller-Caller Question: Does each execution of a recursive method in the algorithm involve either the base case or a smaller case of the original problem?

Yes. When theFirstInteger has a value of any positive integer *i* and theSecondInteger has a value of 2, multipliesRecursively returns the value provided by multipliesRecursively in a base case where theFirstInteger has a value of *i* and theSecondInteger has a value of 2 minus 1, or 1.

When theFirstInteger has a value of any positive integer *i* and theSecondInteger has a value of *m*, multipliesRecursively returns the value provided by multipliesRecursively when theFirstInteger has a value of *i* and theSecondInteger has a value of *m* minus 1. Each execution of multipliesRecursively with a value for theSecondInteger involves either the base case or an execution of multipliesRecursively with a smaller value for theSecondInteger.

1. The General-Case Question: Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

When theFirstInteger has a value of any positive integer *i* and theSecondInteger has a value of *n*, multipliesRecursively returns the value provided by multipliesRecursively when theFirstInteger has a value of *i* and theSecondInteger has a value of *n* minus 1. Each execution of multipliesRecursively with a value for theSecondInteger involves either the base case or an execution of multipliesRecursively with a smaller value for theSecondInteger.

**Problem 2: Letter Permutation**

*Labeling LetterPermutation as Recursive*

LetterPermutation is recursive; method main in class LetterPermutation calls method permutation in the same class, and method permutation calls itself.

*Verifying Algorithm for Letter Permutation*

1. The Base-Case Question: Is there a non-recursive way out of the algorithm, and does the algorithm work correctly for this base case?

Yes. When str has a value of any string and remaining has a value of 1, permutation outputs “L\nR\n” and ends.

1. The Smaller-Caller Question: Does each execution of a recursive method in the algorithm involve either the base case or a smaller case of the original problem?

Yes. Within permutation, when str has a value of any string *s* and remaining has a value of 2, permutation is executed for a base case of str having a value of the result of concatenating *s* and “L”, and remaining having a value of 2 minus 1, or 1.

Within permutation, when str has a value of any string *s* and remaining has a value of *m*, permutation is executed with str having a value of the result of concatenating *s* and “L”, and remaining having a value of *m* minus 1. permutation is executed again with str having a value of the result of concatenating *s* and “R”, and remaining having a value of *m* minus 1. Each execution of permutation with a value for remaining involves either the base case or an execution of permutation with a smaller value for remaining.

1. The General-Case Question: Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

Yes. Within permutation, when str has a value of any string *s* and remaining has a value of *n*, permutation is executed with str having a value of the result of concatenating *s* and “L”, and remaining having a value of *n* minus 1. permutation is executed again with str having a value of the result of concatenating *s* and “R”, and remaining having a value of *m* minus 1. Each execution of permutation with a value for remaining involves either the base case or an execution of permutation with a smaller value for remaining.

**Problem 3: Call Stack**

*The Result of Calling* example(3)

**int** example(**int** n) {

**if** (n == 0) {

**return** 0;

}

**else** {

**return** *example*(n - 1) + n \* n \* n;

}

}

Executing method example after passing example the value 3 for parameter n results in example returning the value 36.

*Using a Call Stack to Explain the Result of Calling* example(3)

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
|  |  |
| Clone of example with n having a value of 2 | returns example(2 - 1) + 2 \* 2 \* 2  returns example(1) + 2 \* 2 \* 2 |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
| Clone of example with n having a value of 1 | returns example(1 - 1) + 1 \* 1 \* 1  returns example(0) + 1 \* 1 \* 1 |
| Clone of example with n having a value of 2 | returns example(2 - 1) + 2 \* 2 \* 2  returns example(1) + 2 \* 2 \* 2 |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
| Clone of example with n having a value of 0 | returns 0 |
| Clone of example with n having a value of 1 | returns example(1 - 1) + 1 \* 1 \* 1  returns example(0) + 1 \* 1 \* 1 |
| Clone of example with n having a value of 2 | returns example(2 - 1) + 2 \* 2 \* 2  returns example(1) + 2 \* 2 \* 2 |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
| Clone of example with n having a value of 1 | returns example(1 - 1) + 1 \* 1 \* 1  returns example(0) + 1 \* 1 \* 1  returns 0 + 1 \* 1 \* 1  returns 0 + 1 \* 1  returns 0 + 1  returns 1 |
| Clone of example with n having a value of 2 | returns example(2 - 1) + 2 \* 2 \* 2  returns example(1) + 2 \* 2 \* 2 |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
|  |  |
| Clone of example with n having a value of 2 | returns example(2 - 1) + 2 \* 2 \* 2  returns example(1) + 2 \* 2 \* 2  returns 1 + 2 \* 2 \* 2  returns 1 + 4 \* 2  returns 1 + 8  returns 9 |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns example(2) + 3 \* 3 \* 3 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| Clone of example with n having a value of 3 | returns example(3 - 1) + 3 \* 3 \* 3  returns 9 + 3 \* 3 \* 3  returns 9 + 9 \* 3  returns 9 + 27  returns 36 |
| Method that calls example(3) for the first time | example(3) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
| Method that calls example(3) for the first time | 36 |

*The Number of Times* factorial(5) *Invokes Itself*

**int** factorial(**int** n) {

**if** (n == 0) {

**return** 1;

}

**else** {

**return** (n \* factorial(n – 1));

}

}

During execution of method factorial with parameter n having value 5, factorial invokes itself 5 times.

*Using a Call Stack to Explain the Result of Calling* factorial(5)

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Method that calls factorial(5) for the first time | factorial(5) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |

Instance of factorial invoking itself

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
| Clone of factorial with n having a value of 4 | returns 4 \* factorial(4 - 1)  returns 4 \* factorial(3) |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| Clone of factorial with n having a value of 3 | returns 3 \* factorial(3 - 1)  returns 3 \* factorial(2) |
| Clone of factorial with n having a value of 4 | returns 4 \* factorial(4 - 1)  returns 4 \* factorial(3) |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |

|  |  |
| --- | --- |
|  |  |
|  |  |
| Clone of factorial with n having a value of 2 | returns 2 \* factorial(2 - 1)  returns 2 \* factorial(1) |
| Clone of factorial with n having a value of 3 | returns 3 \* factorial(3 - 1)  returns 3 \* factorial(2) |
| Clone of factorial with n having a value of 4 | returns 4 \* factorial(4 - 1)  returns 4 \* factorial(3) |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |

|  |  |
| --- | --- |
|  |  |
| Clone of factorial with n having a value of 1 | returns 1 \* factorial(1 - 1)  returns 1 \* factorial(0) |
| Clone of factorial with n having a value of 2 | returns 2 \* factorial(2 - 1)  returns 2 \* factorial(1) |
| Clone of factorial with n having a value of 3 | returns 3 \* factorial(3 - 1)  returns 3 \* factorial(2) |
| Clone of factorial with n having a value of 4 | returns 4 \* factorial(4 - 1)  returns 4 \* factorial(3) |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |

|  |  |
| --- | --- |
| Clone of factorial with n having a value of 0 | returns 1 |
| Clone of factorial with n having a value of 1 | returns 1 \* factorial(1 - 1)  returns 1 \* factorial(0) |
| Clone of factorial with n having a value of 2 | returns 2 \* factorial(2 - 1)  returns 2 \* factorial(1) |
| Clone of factorial with n having a value of 3 | returns 3 \* factorial(3 - 1)  returns 3 \* factorial(2) |
| Clone of factorial with n having a value of 4 | returns 4 \* factorial(4 - 1)  returns 4 \* factorial(3) |
| Clone of factorial with n having a value of 5 | returns 5 \* factorial(5 - 1)  returns 5 \* factorial(4) |
| Method that calls example(3) for the first time | factorial(5) |