**Guided Assignment 3**

Created: 06/03/21 by Tom Lever

Updated: 06/03/21 by Tom Lever

**Problem 1A: 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 2A: 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 3A: 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) |

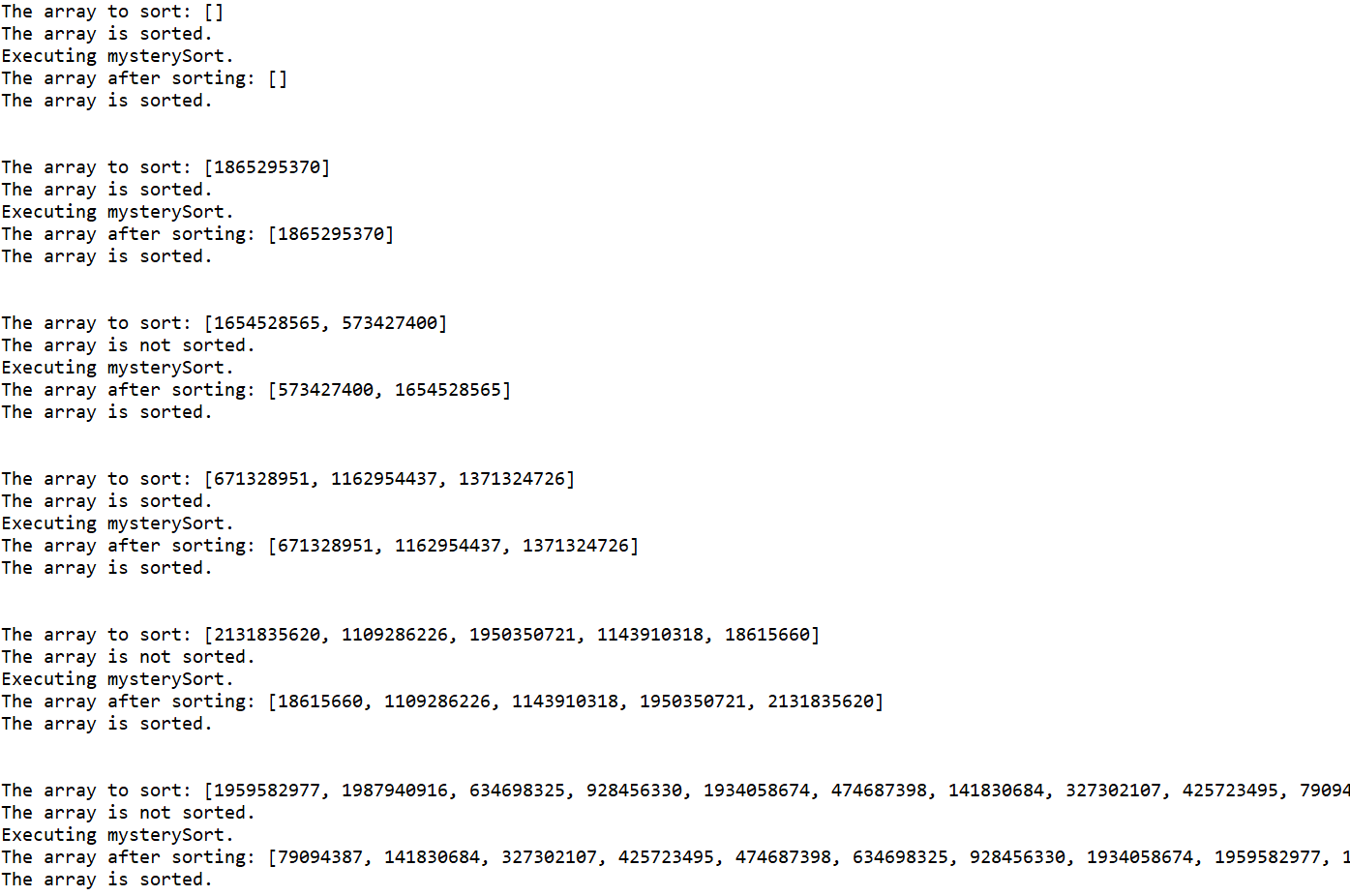
|  |  |
| --- | --- |
| Clone of factorial with n having a value of 0 | returns 1 |
| Clone of factorial with n having a value of 1 | returns 1 \* factorial(1 - 1)  returns 1 \* factorial(0) |
| Clone of factorial with n having a value of 2 | returns 2 \* factorial(2 - 1)  returns 2 \* factorial(1) |
| 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) |

**Problem 1B: Mystery Sort**

The mystery sort that forms the basis for this problem is a Bubble Sort of an array of integers. On the zeroth iteration, every pair of integers, starting with the “left”-most pair, is compared. The number of comparisons is . If the left-most integer in the pair is larger than its partner, the two integers are swapped. Thus, large elements migrate to the “right”. The largest element in the array migrates to the right-most cell in the array. On the first iteration, every pair of integers except the last is compared. The number of comparisons is . On the second iteration, every pair of integers except the last two are compared. The number of comparisons is . On the (array length – 2)-th iteration, one pair of integers is compared. The number of comparisons is .

This particular implementation of Bubble Sort has an order of growth based on comparisons of for large . The number of comparisons is the same for all executions of this implementation. Thus, the average order of growth is .

Test Output



**package** Com.TSL.MysterySortUtilities;

**import** java.util.Arrays;

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

\* MysterySort encapsulates the entry point of this program, which creates an array of random integers, displays the

\* created array, indicates whether or not the created array is already sorted, executes a mystery sort, displays the

\* sorted array, and indicates whether or not the sorted array is actually sorted.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/05/21

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

**public** **class** MysterySort

{

// **TODO**: Change to private before deployment.

**static** **final** **int** ***THE\_MAXIMUM\_INTEGER*** = 2147483647;

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

\* main is the entry point of this program, which creates an array of random integers, displays the created array,

\* indicates whether or not the created array is already sorted, executes a mystery sort, displays the sorted array,

\* and indicates whether or not the sorted array is actually sorted.

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

**public** **static** **void** main( String[] args ) **throws** AnInvalidArraySizeException, AnIntegerOverflowException

{

**int** theArraySize = TheInputAndOutputManager.*providesTheArraySizeAsAnIntegerBasedOn*(args[0]);

**int**[] theArrayToSort = **new** **int**[theArraySize];

**for** (**int** i = 0; i < theArraySize; i++)

{

theArrayToSort[i] =

TheRandomNumberGenerator.*getARandomIntegerInclusivelyBetween*(0, ***THE\_MAXIMUM\_INTEGER*** - 1);

//theArrayToSort[i] = TheRandomNumberGenerator.getARandomIntegerInclusivelyBetween(0, 99);

}

System.***out***.println("The array to sort: " + Arrays.*toString*(theArrayToSort));

TheInputAndOutputManager.*printsWhetherOrNotIsSorted*(theArrayToSort);

System.***out***.println("Executing mysterySort.");

*mysterySort*(theArrayToSort);

System.***out***.println("The array after sorting: " + Arrays.*toString*(theArrayToSort));

TheInputAndOutputManager.*printsWhetherOrNotIsSorted*(theArrayToSort);

}

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

\* isSorted indicates whether or not an array is sorted.

\*

\* **@param** arr

\* **@return**

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

**public** **static** Boolean isSorted(**int**[] arr)

{

**for**(**int** i=1; i<arr.length; i++)

{

**if**(arr[i] < arr[i-1]) {

**return** **false**;

}

}

**return** **true**;

}

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

\* mysterySort performs a mystery sort of an array.

\*

\* **@param** arr

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

**public** **static** **void** mysterySort(**int**[] arr)

{

**for**(**int** i=0; i<arr.length; i++)

{

**for**(**int** k=0; k<arr.length-i-1; k++)

{

**if**(arr[k]>arr[k+1])

{

**int** hold=arr[k+1];

arr[k+1]=arr[k];

arr[k]=hold;

}

}

}

}

}

**package** Com.TSL.MysterySortUtilities;

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

\* TheInputManager encapsulates functionality to provide a desired array size based on a command-line argument.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/05/21

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

**class** TheInputAndOutputManager

{

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

\* providesTheArraySizeAsAnIntegerBasedOn provides a desired array size as an integer based on a command-line

\* argument.

\*

\* **@param** theArraySizeAsAString

\* **@return**

\* **@throws** AnInvalidArraySizeException

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

**static** **int** providesTheArraySizeAsAnIntegerBasedOn(String theArraySizeAsAString) **throws** AnInvalidArraySizeException

{

**int** theArraySizeAsAnInteger = Integer.*parseInt*(theArraySizeAsAString);

**if** (theArraySizeAsAnInteger < 0)

{

**throw** **new** AnInvalidArraySizeException("Exception: The array size is negative.");

}

**return** theArraySizeAsAnInteger;

}

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

\* printWhetherOrNotIsSorted prints whether or not an array is sorted.

\*

\* **@param** theArray

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

**static** **void** printsWhetherOrNotIsSorted(**int**[] theArray)

{

System.***out***.println("The array " + ((MysterySort.*isSorted*(theArray)) ? "is " : "is not ") + "sorted.");

}

}

**package** Com.TSL.MysterySortUtilities;

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

\* AnInvalidArraySizeException represents the structure for an exception that occurs if a row or column ArraySize for

\* a matrix is negative.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/18/21

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

**class** AnInvalidArraySizeException **extends** Exception {

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

\* AnInvalidArraySizeException() is a conventional zero-argument constructor for AnInvalidArraySizeException, which

\* calls Exception's zero-argument constructor.

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

AnInvalidArraySizeException() {

**super**();

}

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

\* AnInvalidArraySizeException(String message) is a one-argument constructor for AnInvalidArraySizeException, which

\* passes an error message to Exception's one-argument constructor with a message argument.

\*

\* **@param** message

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

AnInvalidArraySizeException(String message) {

**super**(message);

}

}

**package** Com.TSL.MysterySortUtilities;

**import** java.util.Random;

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

\* TheRandomNumberGenerator represents a random number generator.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

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

**class** TheRandomNumberGenerator {

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

\* THE\_MAXIMUM\_INTEGER is an attribute of ARandomNumberGenerator.

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

**private** **static** **final** **int** ***THE\_STATIC\_MAXIMUM\_INTEGER*** = 2147483647;

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

\* thePopularRandomNumberGenerator is a component of ARandomNumberGenerator.

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

**private** **static** Random *thePopularRandomNumberGenerator* = **new** Random ();

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

\* getARandomIntegerInclusivelyBetween provides a integer between a lower limit and an upper limit inclusive.

\*

\* **@param** theLowerLimit

\* **@param** theUpperLimit

\* **@return**

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

**protected** **static** **int** getARandomIntegerInclusivelyBetween (**int** theLowerLimit, **int** theUpperLimit)

**throws** AnIntegerOverflowException {

/\*if (theUpperLimit > THE\_STATIC\_MAXIMUM\_INTEGER + theLowerLimit - 1) {

throw new AnIntegerOverflowException (

"Exception: The proposed range [lower limit, upper limit] is too wide for a random number generator.\n" +

"A possible range is [-1,073,741,823, 1,073,741,823]."

);

}\*/

**return** *thePopularRandomNumberGenerator*.nextInt((theUpperLimit - theLowerLimit) + 1) + theLowerLimit;

}

}

**package** Com.TSL.MysterySortUtilities;

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

\* AnIntegerOverflowException represents the structure for an exception that occurs when an arithmetic operation

\* would cause an integer to be greater than the maximum integer or less than the minimum integer.

\* **@author** Tom Lever

\* **@version** 1.0

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

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

**class** AnIntegerOverflowException **extends** Exception {

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

\* AnIntegerOverflowException() is a conventional zero-parameter constructor for AnIntegerOverflowException, which

\* calls Exception's zero-parameter constructor.

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

**protected** AnIntegerOverflowException() {

**super** ();

}

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

\* AnIntegerOverflowException(String message) is a one-parameter constructor for AnIntegerOverflowException, which

\* passes an error message to Exception's one-parameter constructor.

\* **@param** message

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

**protected** AnIntegerOverflowException (String message) {

**super** (message);

}

}

package Com.TSL.MysterySortUtilities;

import java.util.Arrays;

import org.junit.jupiter.api.Test;

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

\* MainTest encapsulates a JUnit test that performs multiple sorting processes with mysterySort and an array of various

\* sizes and random integers.

\*

\* @author Tom Lever

\* @version 1.0

\* @since 06/05/21

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

public class MainTest {

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

\* testMain performs multiple sorting processes with mystery sort and an array of various sizes and random integers.

\*

\* @throws AnIntegerOverflowException

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

@Test

public void testMain() throws AnIntegerOverflowException

{

int[] theArraySizes = {0, 1, 2, 3, 5, 10};

for (int i = 0; i < theArraySizes.length; i++)

{

int[] theArrayToSort = new int[theArraySizes[i]];

for (int j = 0; j < theArraySizes[i]; j++)

{

theArrayToSort[j] =

TheRandomNumberGenerator.getARandomIntegerInclusivelyBetween(0, MysterySort.THE\_MAXIMUM\_INTEGER - 1);

}

System.out.println("The array to sort: " + Arrays.toString(theArrayToSort));

TheInputAndOutputManager.printsWhetherOrNotIsSorted(theArrayToSort);

System.out.println("Executing mysterySort.");

MysterySort.mysterySort(theArrayToSort);

System.out.println("The array after sorting: " + Arrays.toString(theArrayToSort));

TheInputAndOutputManager.printsWhetherOrNotIsSorted(theArrayToSort);

System.out.println("\n");

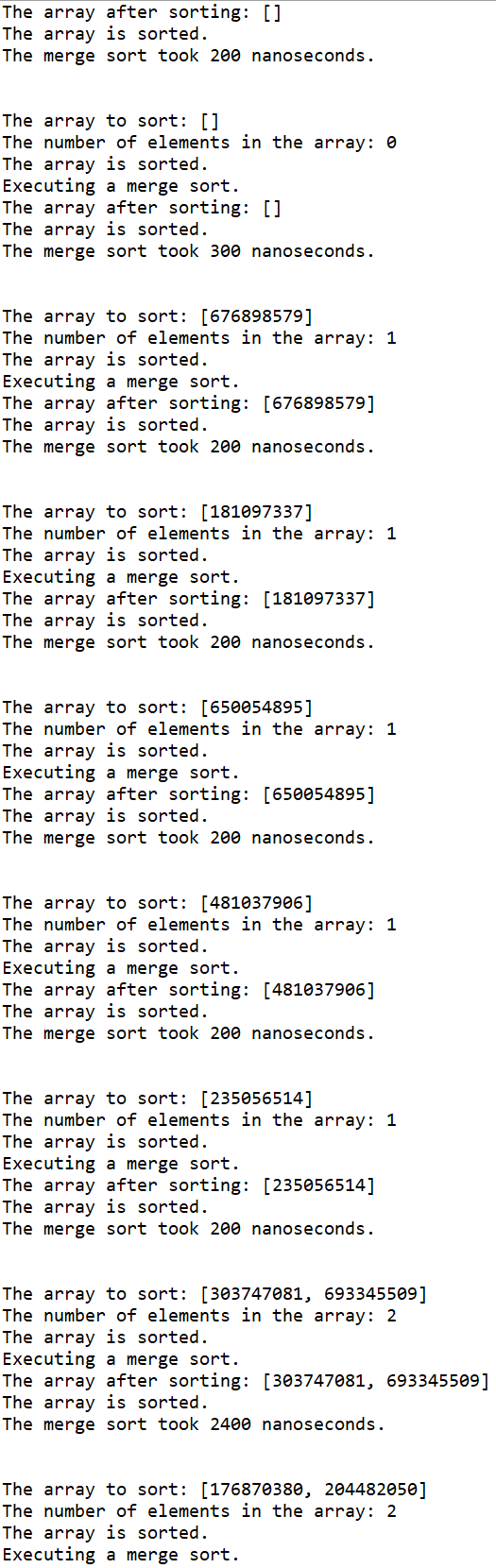
}

}

}

**Problem 2B: Merge Sort**

Per “Object Oriented Data Structures Using Java” and “Comparison of Sorting Algorithms”, the average order of growth of mergeSort is .



**package** Com.TSL.MergeSortUtilities;

**import** java.util.Arrays;

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

\* MergeSort encapsulates the entry point of this program, which creates an array of integers based on command-line

\* arguments; displays the array, its length, and whether or not it is sorted; performs a merge sort; and, after the

\* sort, displays the array and whether or not it is sorted.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/05/21

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

**class** MergeSort

{

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

\* main is the entry point of this program, which creates an array of integers based on command-line arguments;

\* displays the array, its length, and whether or not it is sorted; performs a merge sort; and, after the sort,

\* displays the array and whether or not it is sorted.

\*

\* **@param** args

\* **@throws** AnInvalidArraySizeException

\* **@throws** AnInvalidLimitsException

\* **@throws** AnIntegerOverflowException

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

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

**throws** AnInvalidArraySizeException, AnInvalidLimitsException, AnIntegerOverflowException

{

**int** theArraySize = TheInputAndOutputManager.*providesTheArraySizeAsAnIntegerBasedOn*(args[0]);

**int** theLowerLimitForAnInteger = Integer.*parseInt*(args[1]);

**int** theUpperLimitForAnInteger = Integer.*parseInt*(args[2]);

**if** (theLowerLimitForAnInteger > theUpperLimitForAnInteger)

{

**throw** **new** AnInvalidLimitsException(

"Exception: The desired lower limit for an integer is greater than the desired upper limit.");

}

**int**[] theArrayToSort = **new** **int**[theArraySize];

**for** (**int** i = 0; i < theArraySize; i++)

{

theArrayToSort[i] = TheRandomNumberGenerator.*getARandomIntegerInclusivelyBetween*(

theLowerLimitForAnInteger, theUpperLimitForAnInteger

);

}

System.***out***.println("The array to sort: " + Arrays.*toString*(theArrayToSort));

System.***out***.println("The number of elements in the array: " + theArrayToSort.length);

TheInputAndOutputManager.*printsWhetherOrNotIsSorted*(theArrayToSort);

System.***out***.println("Executing a merge sort.");

**long** theStartTime = System.*nanoTime*();

*mergeSort*(theArrayToSort);

**long** theEndTime = System.*nanoTime*();

System.***out***.println("The array after sorting: " + Arrays.*toString*(theArrayToSort));

TheInputAndOutputManager.*printsWhetherOrNotIsSorted*(theArrayToSort);

System.***out***.println("The merge sort took " + (theEndTime - theStartTime) + " nanoseconds.");

}

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

\* mergeSort performs a merge sort of an array of integers.

\*

\* **@param** arr

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

**public** **static** **void** mergeSort(**int**[] arr)

{

*mergeSortRec*(arr, 0, arr.length-1);

}

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

\* mergeSortRec recursively performs a merge sort of a subarray of integers between two integers inclusive.

\*

\* **@param** arr

\* **@param** first

\* **@param** last

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

**private** **static** **void** mergeSortRec(**int**[] arr, **int** first, **int** last)

{

**if**(first < last)

{

**int** mid = (first + last) / 2;

*mergeSortRec*(arr, first, mid);

*mergeSortRec*(arr, mid + 1, last);

*merge*(arr, first, mid, mid + 1, last);

}

}

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

\* merge merges a sorted subarray of integers between two integers inclusive with another sorted subarray between

\* another two integers inclusive.

\*

\* **@param** arr

\* **@param** leftFirst

\* **@param** leftLast

\* **@param** rightFirst

\* **@param** rightLast

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

**private** **static** **void** merge(**int**[] arr, **int** leftFirst, **int** leftLast, **int** rightFirst, **int** rightLast)

{

**int**[] aux = **new** **int**[arr.length];

//extra space, this is downside of this algorithm

**int** index = leftFirst;

**int** saveFirst = leftFirst;

**while**(leftFirst <= leftLast && rightFirst <= rightLast)

{

**if**(arr[leftFirst] <= arr[rightFirst])

{

aux[index++] = arr[leftFirst++];

}

**else**

{

aux[index++] = arr[rightFirst++];

}

}

**while**(leftFirst <= leftLast)

{

aux[index++] = arr[leftFirst++];

}

**while** (rightFirst <= rightLast)

{

aux[index++]=arr[rightFirst++];

}

**for**(index=saveFirst; index<=rightLast; index++)

{

arr[index]=aux[index];

}

}

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

\* isSorted indicates whether or not an array is sorted.

\*

\* **@param** arr

\* **@return**

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

**public** **static** **boolean** isSorted(**int**[] theArray)

{

**for**(**int** i = 1; i < theArray.length; i++)

{

**if**(theArray[i] < theArray[i-1]) {

**return** **false**;

}

}

**return** **true**;

}

}

**package** Com.TSL.MergeSortUtilities;

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

\* TheInputAndOutputManager encapsulates functionality to provide an array size as an integer based on a command-line

\* argument and to print whether or not an array is sorted.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 06/05/21

\*

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

**class** TheInputAndOutputManager

{

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

\* providesTheArraySizeAsAnIntegerBasedOn provides an array size based on a command-line argument.

\*

\* **@param** theArraySizeAsAString

\* **@return**

\* **@throws** AnInvalidArraySizeException

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

**static** **int** providesTheArraySizeAsAnIntegerBasedOn(String theArraySizeAsAString) **throws** AnInvalidArraySizeException

{

**int** theArraySizeAsAnInteger = Integer.*parseInt*(theArraySizeAsAString);

**if** (theArraySizeAsAnInteger < 0)

{

**throw** **new** AnInvalidArraySizeException("Exception: The array size is negative.");

}

**return** theArraySizeAsAnInteger;

}

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

\* printWhetherOrNotIsSorted prints whether or not an array is sorted.

\*

\* **@param** theArray

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

**static** **void** printsWhetherOrNotIsSorted(**int**[] theArray)

{

System.***out***.println("The array " + ((MergeSort.*isSorted*(theArray)) ? "is " : "is not ") + "sorted.");

}

}

**package** Com.TSL.MergeSortUtilities;

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

\* AnInvalidArraySizeException represents the structure for an exception that occurs if the integer representing an array

\* size is negative.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/18/21

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

**class** AnInvalidArraySizeException **extends** Exception {

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

\* AnInvalidArraySizeException() is a conventional zero-argument constructor for AnInvalidArraySizeException, which

\* calls Exception's zero-argument constructor.

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

AnInvalidArraySizeException() {

**super**();

}

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

\* AnInvalidArraySizeException(String message) is a one-argument constructor for AnInvalidArraySizeException, which

\* passes an error message to Exception's one-argument constructor with

\* a message argument.

\*

\* **@param** message

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

AnInvalidArraySizeException(String message) {

**super**(message);

}

}

**package** Com.TSL.MergeSortUtilities;

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

\* AnInvalidLimitsException represents the structure for an exception that occurs if the a lower limit is greater than

\* an upper limit.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/18/21

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

**class** AnInvalidLimitsException **extends** Exception {

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

\* AnInvalidLimitsException() is a conventional zero-argument constructor for AnInvalidLimitsException, which calls

\* Exception's zero-argument constructor.

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

AnInvalidLimitsException() {

**super**();

}

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

\* AnInvalidLimitsException(String message) is a one-argument constructor for AnInvalidLimitsException, which passes

\* an error message to Exception's one-argument constructor with a message argument.

\*

\* **@param** message

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

AnInvalidLimitsException(String message) {

**super**(message);

}

}

**package** Com.TSL.MergeSortUtilities;

**import** java.util.Random;

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

\* TheRandomNumberGenerator represents a random number generator.

\*

\* **@author** Tom Lever

\* **@version** 1.0

\* **@since** 05/28/21

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

**class** TheRandomNumberGenerator {

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

\* THE\_MAXIMUM\_INTEGER is an attribute of ARandomNumberGenerator.

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

**private** **static** **final** **int** ***THE\_STATIC\_MAXIMUM\_INTEGER*** = 2147483647;

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

\* thePopularRandomNumberGenerator is a component of ARandomNumberGenerator.

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

**private** **static** Random *thePopularRandomNumberGenerator* = **new** Random ();

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

\* getARandomIntegerInclusivelyBetween provides a integer between a lower limit and an upper limit inclusive.

\*

\* **@param** theLowerLimit

\* **@param** theUpperLimit

\* **@return**

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

**protected** **static** **int** getARandomIntegerInclusivelyBetween (**int** theLowerLimit, **int** theUpperLimit)

**throws** AnIntegerOverflowException {

/\*if (theUpperLimit > THE\_STATIC\_MAXIMUM\_INTEGER + theLowerLimit - 1) {

throw new AnIntegerOverflowException (

"Exception: The proposed range [lower limit, upper limit] is too wide for a random number generator.\n" +

"A possible range is [-1,073,741,823, 1,073,741,823]."

);

}\*/

**return** *thePopularRandomNumberGenerator*.nextInt((theUpperLimit - theLowerLimit) + 1) + theLowerLimit;

}

}

**package** Com.TSL.MergeSortUtilities;

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

\* AnIntegerOverflowException represents the structure for an exception that occurs when an arithmetic operation

\* would cause an integer to be greater than the maximum integer or less than the minimum integer.

\* **@author** Tom Lever

\* **@version** 1.0

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

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

**class** AnIntegerOverflowException **extends** Exception {

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

\* AnIntegerOverflowException() is a conventional zero-parameter constructor for AnIntegerOverflowException, which

\* calls Exception's zero-parameter constructor.

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

**protected** AnIntegerOverflowException() {

**super** ();

}

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

\* AnIntegerOverflowException(String message) is a one-parameter constructor for AnIntegerOverflowException, which

\* passes an error message to Exception's one-parameter constructor.

\* **@param** message

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

**protected** AnIntegerOverflowException (String message) {

**super** (message);

}

}

package Com.TSL.MergeSortUtilities;

import java.util.Arrays;

import org.junit.jupiter.api.Test;

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

\* MainTest encapsulates a JUnit test of the main program that creates arrays of array sizes, lower limits for integers,

\* and upper limits for integers, and performs a merge sort for an array of random integers for each array size and

\* pair of limits.

\*

\* @author Tom Lever

\* @version 1.0

\* @since 06/05/21

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

public class MainTest {

private final int THE\_LOWER\_LIMIT\_FOR\_AN\_INTEGER = -1073741823;

private final int THE\_UPPER\_LIMIT\_FOR\_AN\_INTEGER = 1073741823;

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

\* testMain creates arrays of array sizes, lower limits for integers, and upper limits for integers, and performs a

\* merge sort for an array of random integers for each array size and pair of limits.

\*

\* @throws AnIntegerOverflowException

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

@Test

public void testMain() throws AnIntegerOverflowException

{

int[] theArraySizes = {0, 1, 2, 3, 5, 10};

int[] theLowerLimits = new int[5];

int[] theUpperLimits = new int[theLowerLimits.length];

for (int i = 0; i < theLowerLimits.length; i++)

{

theLowerLimits[i] = TheRandomNumberGenerator.getARandomIntegerInclusivelyBetween(

THE\_LOWER\_LIMIT\_FOR\_AN\_INTEGER, this.THE\_UPPER\_LIMIT\_FOR\_AN\_INTEGER

);

theUpperLimits[i] = TheRandomNumberGenerator.getARandomIntegerInclusivelyBetween(

theLowerLimits[i], this.THE\_UPPER\_LIMIT\_FOR\_AN\_INTEGER

);

}

for (int i = 0; i < theArraySizes.length; i++)

{

for (int j = 0; j < theLowerLimits.length; j++)

{

int[] theArrayToSort = new int[theArraySizes[i]];

for (int k = 0; k < theArraySizes[i]; k++)

{

theArrayToSort[k] = TheRandomNumberGenerator.getARandomIntegerInclusivelyBetween(

theLowerLimits[j], theUpperLimits[j]

);

}

System.out.println("The array to sort: " + Arrays.toString(theArrayToSort));

System.out.println("The number of elements in the array: " + theArrayToSort.length);

TheInputAndOutputManager.printsWhetherOrNotIsSorted(theArrayToSort);

System.out.println("Executing a merge sort.");

long theStartTime = System.nanoTime();

MergeSort.mergeSort(theArrayToSort);

long theEndTime = System.nanoTime();

System.out.println("The array after sorting: " + Arrays.toString(theArrayToSort));

TheInputAndOutputManager.printsWhetherOrNotIsSorted(theArrayToSort);

System.out.println("The merge sort took " + (theEndTime - theStartTime) + " nanoseconds.");

System.out.println("\n");

}

}

}

}

**Problem 3B: Short Answer Questions**

Starting with Bubble Sort, the implementation of BubbleSort presented in Problem 1B is a “worst-case” implementation. Every execution of this implementation has the same number of comparisons. For an array containing ten elements, the number of comparisons

We consider Selection Sort and the following array with 8 integers.

|  |  |
| --- | --- |
| **0** | 43 |
| **1** | 7 |
| **2** | 190 |
| **3** | 23 |
| **4** | 18 |
| **5** | 5 |
| **6** | 86 |
| **7** | 14 |

Following the algorithm presented in “Object-Oriented Data Structures Using Java”:

During the first iteration, a current index is 0. The index of the minimum integer in the subarray between the current index and the index of the last integer in the array is found to be 5. The integer at the index of the minimum integer 5, also 5, is swapped with the integer at the current index 0, 43. Below is the array of integers after one iteration of sorting.

|  |  |
| --- | --- |
| **0** | 5 |
| **1** | 7 |
| **2** | 190 |
| **3** | 23 |
| **4** | 18 |
| **5** | 43 |
| **6** | 86 |
| **7** | 14 |

During the second iteration, a current index is 1. The index of the minimum integer in the subarray between the current index and the index of the last integer array is found to be 1. The integer at the index of the minimum integer 1, 7, is swapped with the integer at the current index 1 (i.e., itself), 7. Below is the array of integers after two iterations of sorting.

|  |  |
| --- | --- |
| **0** | 5 |
| **1** | 7 |
| **2** | 190 |
| **3** | 23 |
| **4** | 18 |
| **5** | 43 |
| **6** | 86 |
| **7** | 14 |

Considering an array of 100 elements and a Sequential Search, per “Object-Oriented Data Structures Using Java”, “On average, assuming that there is an equal probability of searching for any element in the collection, we will make comparisons for a successful search; that is, on average we must search half of the collection”.

Suppose we have an array of 100 unique integers sorted in ascending order. We choose a specific integer *i*. During each of an infinite number of trials, we randomize the ordering of the integers in the array and search for integer *i*. On average, we will make comparisons for a successful search.

Suppose we have an array of 100 unique integers sorted in ascending order. During each of an infinite number of trials, we choose a random integer *i* that is in our array. On average, we will make comparisons for a successful search.

Suppose we have an array of 100 unique integers sorted in ascending order. We choose the smallest integer. During any trial, we will make 1 comparison for a successful search.

Suppose we have an array of 100 unique integers sorted in ascending order. We choose the largest integer. During any trial, we will make *N* comparisons for a successful search.

Suppose we have an array of 100 unique integers sorted in descending order. During each of an infinite number of trials, we choose a random integer *i* that is in our array. On average, we will make comparisons for a successful search.

Suppose we have an array of 100 unique integers sorted in descending order. We choose the largest integer. During any trial, we will make 1 comparison for a successful search.

Suppose we have an array of 100 unique integers sorted in descending order. We choose the smallest integer. During any trial, we will make *N* comparisons for a successful search.