Algorithms Problem Sheet (Java)

Course Name: "Higher Diploma in Data Analytics
By: Marco Men (G00221420)
Due Date: 21 / 03 / 2019

Question 1

The output of the call to mystery(1) is a printout of an array(1,2,3,4,4,3,2,1). The code and output will be displayed below. A minor adjustment was made by me regarding the "print" statements, I used "println" instead to give a vertical output of the different integers, rather than in a horizontal line without any commas separating them.

Reasons For Answer

In the code below, the method 'mystery' is a recursive method. 4
The main method calls for mystery function where n =1 and prints it. It then loops through a conditional statement (if n is less than 4, add 1 to n). This condition imposed on the variable n, will be true until n reaches the value 4 (known as recursion base case) rendering the conditional statement false, since n is equal to 4 and not less. At this point the function is exited,because there are no more calls to make.

The last "println" statement (outside the function) traces back the return of n to the caller by -1 (from 4 to 1 and exits). This would not be visible if the last "println" statement would not be present (it would simply return). In other words, we are able to visualise both the caller and return functions using the recursive method.

I must mention the stack and its principle of LIFO.
To quote Mark Allen Weiss (in "Data Structures & Problem Solving Using Java") where he wirites: "The stack is a data structure in which access is restricted to the most recently inserted item... The last item added to the stack is placed on top and is easily accessible, whereas items that have been in the stack for a while are more difficult to access. Thus the stack is appropriate if we expect to access only the top item; all the other items are inaccessible."

When a call is made using recursion, its called values are stored in stack memory.
The recursion ends when it reaches its base case, in this example
n = 4. All the values stored in the stack must be removed, using the principle of LIFO (Last In, First Out). So the last n value stored in the stack was 4 (at the **top** of the stack) After this value of n is removed from the **stack,** the **top** is updated to n -1 ... Until n = 1. At this point no more n values to be removed, so a blank line is returned. This means that the stack memory is emptied.

Java Code

```
public class CTJava1 {
    // Method call
    public static void main(String args[]) {
```

```
            mystery( 1 );              // it returns the first element in the series
    }
    // Arguments for objects in method
    public static void mystery ( int n ) {
        System.out.println( n );       // prints it
        if ( n < 4 ) {                 // if condition is true, it jumps to the next
                                       element by adding 1 to n
            mystery( n + 1 );          // keeps adding until n = 4, prints it and
                                       exits the call function (condition no longer
                                       satisfied)
        }
        System.out.println(n);         // prints n in a return function and reverts back
                                       to caller ( n =1 ) before exiting – no more
                                       returns to be made
    }
}
```

Output of Code

```
/usr/lib/jvm/java-8-oracle/bin/java -ProblemSheet1 com.examples.CTJava1
1
2
3
4
4
3
2
1

Process finished with exit code 0
```

Note: There is a blank space before the process finish, where no value is returned.

Debugging Test Code For Stack Trace

For my debugging code, I want to explicitly show what is the value of ‘n’in the Stack memory each time ‘n’ is updated. We will see the insertion and deletion in the stack memory and its updated stack top.

```
public class CTJava1test {
    // Method call
    public static void main ( String args[] ) {
        mystery( 1 ); // it returns the first element in the series
    }
    // Arguments for objects in method
    public static void mystery ( int n ) {
        /**Stack Trace Code Adapted From:
         * " http://www.benmccann.com/printing-a-stack-trace-anywhere-in-java/ "
         */
        // Call Stack Trace for 'n' values
        System.out.println("Call Stack Trace:");
        System.out.println( "n =  " + n );  // prints it
```

```
        if ( n < 4 ) {
            StackTraceElement[] cause = Thread.currentThread().getStackTrace();
            mystery( n + 1 );
            StackTraceElement s = cause[n];
            System.out.println("\tat " + s.getClassName() + "." +s.getMethodName() +
"(" + s.getFileName() +":" +
                    s.getLineNumber() + ")");
        }
        // Call Stack Trace for 'n' values returning to caller - LIFO method
        System.out.println("Stack Trace Return To Caller:");
        System.out.println( "n =  " + n );
    }
}
```

Output For Test Code

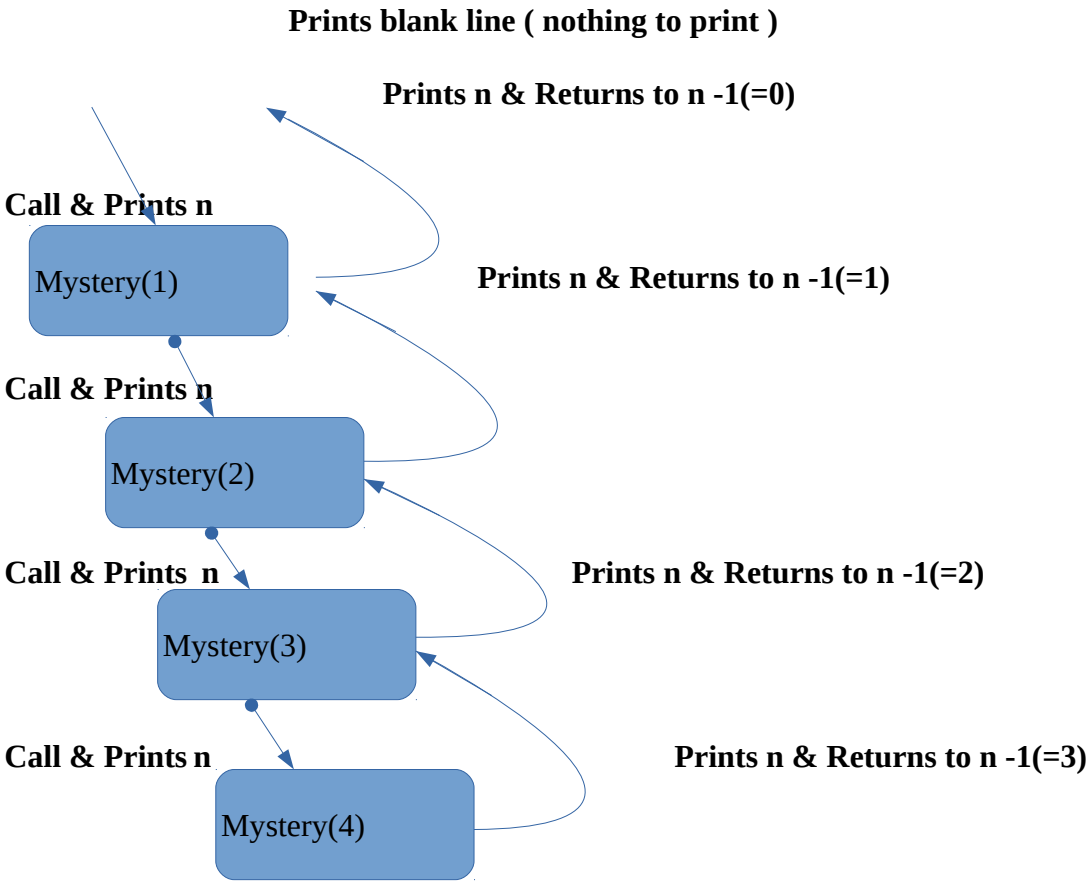
/usr/lib/jvm/java-8-oracle/bin/java ...

Call Stack Trace:
n = 1
Call Stack Trace:
n = 2
Call Stack Trace:
n = 3
Call Stack Trace:
n = 4
Stack Trace Return To Caller:
n = 4
	at com.examples.CTJava1test.mystery(CTJava1test.java:27)
Stack Trace Return To Caller:
n = 3
	at com.examples.CTJava1test.mystery(CTJava1test.java:27)
Stack Trace Return To Caller:
n = 2
	at com.examples.CTJava1test.mystery(CTJava1test.java:25)
Stack Trace Return To Caller:
n = 1

Process finished with exit code 0

The IDE (IntelliJ IDEA) used for the Java codes depicted in this assesment provide tools for debugging, including stack call trace. I chose to achieve through coding.

Diagram



Question 2

The Code

```
public class CTJava2 {
    public static void main ( String args[] ) {
        int inputArr[] = {0, -247, 341, 1001, 741, 22};
        System.out.print(finder(inputArr));
    }
    public static int finder ( int [] input ) {
        return finderRec( input, input.length - 1 );
    }
    // Binary Search below
    public static int finderRec(int[] input, int x)  {
        if(x == 0)  {
            return input[x];          // if the value of the integer in the array is 0,
                                      it will start the search
                                       // from the last element of the array ( x =
                                      input.length - 1 ) until there is only
                                       // one integer( input[0] )
                                       // The search will continue In this case until no
                                      more integers
                                       // in the array to to compare ( input[0] = 1
                                      integer in the array)
        }
        int v1 = input[x];
```

```
        int v2 = finderRec(input,x-1);
        if(v1>v2)  {
            return v1;                // it returns the highest values either v1 or v2
                                      and eliminates the lowest values
        }
        else {
            return v2;                // if there are two equal integers ( let's say
                                      the highest ),
                                      // we will achieve the same result. The process
                                      of elimination grants v2 to be returned.
                                      // Example: v1 = 1001 and v2 = 1001, v2 is
                                      returned
        }
    }
}
```

a) The Output of the Code

/usr/lib/jvm/java-8-oracle/bin/java...

1001
Process finished with exit code 0

b) Reasons For Answer

The finder method determines the highest value in the array.
It does this by splitting the array in half,by grouping the integers in pairs and comparing them 2 by 2s. The highest integers will be returned, either v1 or v2, until no more comparisons are available (x==0), which indicates that the array has been fully searched. This is called a **binary search** recursive tree.

The big advantage to this method is that runtimes are cut-off by half, due to the subsetting the elements in the array by 2. The comparison is between the 2 elements, where there are only 2 possiblities ('Win' or 'Lose'). When the 2 elements are being compared, the *winning element* gets "pushed up" to the next level, whereas the *losing element* gets left behind and stored in the memory stack. The same operation is repeated, where all the wining elements are again paired with each other, compared and only the highest is returned, until there are no more elements in the array to pair. Which means there is only one element which is the highest.

c) Inline Comments

```
public class CTJava2 {
    public static void main ( String args[] ) {
        int inputArr[] = {0, -247, 341, 1001, 741, 22};
        System.out.print(finder(inputArr));
    }
    public static int finder ( int [] input ) {
        return finderRec( input, input.length - 1 );
    }
    // Binary Search below
```

```
    public static int finderRec(int[] input, int x)  {
        if(x == 0)  {
            return input[x];        // if the value of the integer in the array is 0,
it will start the search
                                    // from the last element of the array ( x =
input.length - 1 ) until there is only
                                    // one integer( input[0] )
                                    // The search will continue In this case until no
more integers
                                    // in the array to to compare ( input[0] = 1
integer in the array)
        }
        int v1 = input[x];
        int v2 = finderRec(input,x-1);
        if(v1>v2)  {
            return v1;  // it returns the highest values either v1 or v2 and eliminates
the lowest values
        }
        else {
            return v2;  // if there are two equal integers ( let's say the highest ),
                        // we will achieve the same result. The process of elimination
grants v2 to be returned.
                        // Example: v1 = 1001 and v2 = 1001, v2 is returned
        }
    }
}
```

d) The Iteration Method

```
    /** Adapted from GeekForGeeks:
" https://www.geeksforgeeks.org/java-program-for-program-to-find-largest-element-in-an-
array/ "
     */

public class CTJava2d {
   static int inputArr[] = {0, -247, 341, 1001, 741, 22};
    static int highest() {
        int x;
        int v2 = inputArr[0];
        for(x = 1; x < inputArr.length; x++)
            if(inputArr[x] > v2)
              v2 = inputArr[x];
        return v2;
    }
    public static void main(String args[])  {
        System.out.println(highest());
    }
}
```

The Output

```
/usr/lib/jvm/java-8-oracle/bin/java ...
1001

Process finished with exit code 0
```

As evidenced both methods achieve the same result! But what exactly happens in the background? I performed a few debugging codes for testing.

In my view, one of the most appropriate methods for debugging an Object Oriented Programming language code is to reveal its objects (otherwise known as variables) by “instructing ” the variables to be printed. From time to time, the search may be for the index of elements or for the exact elements in an array. And this is what I attempted to achieve in all the questions debugging codes.

In a program, objects can take many forms before reaching the final output. To the viewer of the output, we only see the final result. It is never considered or thought of, what operations are hidden from the viewer. So why not reveal the forms the objects (or variables) before the final product , in order to better understand the code that brings the output.

Debugger Test Code

```
public class CTJava2test1 {
    public static void main ( String args[] ) {
        int inputArr[] = {0, -247, 341, 1001, 741, 22};
        System.out.print(finder(inputArr));
    }
    public static int finder ( int [] input ) {
        return finderRec( input, input.length - 1 );
    }
    // Binary Search below where q is where x = 341 in the array
    public static int finderRec(int[] input, int x) {
        if ( x == 0 ) {
            return input[x];
        }
        int v1 = input[x];
        int v2 = finderRec( input, x - 1 );
        // The 2 code strings below will trace both variable positions in the array
        System.out.println("This is v1: " + v1);
        System.out.println("This is v2: " + v2);
        if ( v1 > v2 ) {
            return v1;
        } else {
            return v2;
            }
    }
}
```

Output For Debugger Test Code

/usr/lib/jvm/java-8-oracle/bin/java ...

This is v1: -247

```
This is v2: 0
This is v1: 341
This is v2: 0
This is v1: 1001
This is v2: 341
This is v1: 741
This is v2: 1001
This is v1: 22
This is v2: 1001
1001
Process finished with exit code 0
```

The code output shows a the exact positions in the array for both variables (**v1** and **v2**) throughout the running of the algorithm.

Recursion vs. Iteration (on Runtimes)

Next, the runtimes between the recursive method and the iteration method to ascertain which method runs faster on the exact same array. I will be measuring time here and throughout this answer sheet in nanoseconds for precision purposes. There are pros and cons to the usage of nanoseconds versus milliseconds. But for the purposes of measuring runtimes, this should be irrelevant.

Java Code Testing Runtime For Recursive Method

```java
public class CTJava2test {
    public static void main ( String args[] ) {
        // Timing  array runtime in nanoseconds
        final long startTime = System.nanoTime();
        int inputArr[] = {0, -247, 341, 1001, 741, 22};
        System.out.println(finder(inputArr));
        final long endTime = System.nanoTime();
        final long totalTime = endTime - startTime;
        System.out.println( "Using Recursion method takes " + totalTime + " nanoseconds
to run");
        // Finished timing the  array
    }
    public static int finder ( int [] input ) {
        return finderRec( input, input.length - 1 );
    }
    public static int finderRec(int[] input, int x)  {
        if(x == 0)  {
            return input[x];
        }
        int v1 = input[x];
        int v2 = finderRec(input,x-1);
        if(v1>v2)  {
            return v1;
        }
        else {
```

```
            return v2;
        }
    }
}
```

Output

```
/usr/lib/jvm/java-8-oracle/bin/java ...
1001
Using Recursion method takes 602441 nanoseconds to run

Process finished with exit code 0
```

Java Code Testing Runtime For Iterative Method

```
    /** Adapted from GeekForGeeks:
" https://www.geeksforgeeks.org/java-program-for-program-to-find-largest-element-in-an-array/ "
     */
    public class CTJava2d {
    static int inputArr[] = {0, -247, 341, 1001, 741, 22};
        static int highest () {
            int x;
            int v2 = inputArr[0];
            for (x = 1; x < inputArr.length; x++)
                if ( inputArr[x] > v2 )
                    v2 = inputArr[x];
                return v2;
            }
            public static void main (String args [] ) {
                final long startTime = System.nanoTime();
            System.out.println(highest());
                final long endTime1 = System.nanoTime();
                final long totalTime1 = endTime1 - startTime;
                System.out.println( "Using the iteration method takes " + totalTime1 +
" nanoseconds to run" );
        }
    }
```

Output

```
/usr/lib/jvm/java-8-oracle/bin/java ...
1001
Using the iteration method takes 6123674 nanoseconds to run

Process finished with exit code 0
```

So recursive method is quicker than the iterative method on the same array. One point I must make is each time I have run these codes, different runtimes are shown, but the recursive is always faster than the iterative method.

Question 3

a) The best – case time complexity for this method is constant (O(1)) because it will depend on the position of the duplicate elements in the array. For example, if the first 2 elements are duplicates, the “containDuplicates” function will return true and exit, without having to iterate on all elements of the array. So the runtimes will be the same if it is an array of 5,500 or 5 000, because only the first 2 elements are compared.

b) The worst – case time complexity for this method is depends on the squre of the number of elements of the array ($O(n^2)$), because this is a binary search that uses a nested inner loop. This means that all elements of the array must be iterated fully by both variables (in this case, ***i*** and ***j***). In the worst case, no duplicates are found before the function is exited

This is often referred to as the “Element Uniqueness problem”.

c) Code That Returns Number Of Comparisons

```
public class CTjava3c {
    public static void main(String args []) {
        int Array[] = {10, 0, 5, 3, -19, 5};
        int Array1[] = {0, 1, 0, -127, 346, 125};
        System.out.println( containsDuplicates( Array ) );        // 4 comparisons
        System.out.println( "The number of comparisons in Array before a duplicate is
found is " + compare );
        compare = 0;    // reset comparator to 0 after first array comparison
                        // before starting the second array comparison
        System.out.println(containsDuplicates( Array1 ));       // 2 comparisons
        System.out.println("The number of comparisons in Array1 before a duplicate is
found is " + compare);
    }
    public static int compare = 0;
    public static boolean containsDuplicates ( int elements[] ) {
        for (int i = 0; i < elements.length; i++) {
            for (int j = 0; j < elements.length; j++) {
                compare ++;
                if ( i == j ) {
                    compare--;      //eliminate the self comparison count
                    continue;
                }
                if ( elements[i] == elements[j] ) {     // includes the last
                                                          comparison, where the
                                                          duplicates are found
                    return true;
                }
            }
```

```
        }
        return false;
    }
}
```

Output For The Code

```
/usr/lib/jvm/java-8-oracle/bin/java ...
true
The number of comparisons in Array before a duplicate is found is 15
true
The number of comparisons in Array1 before a duplicate is found is 2

Process finished with exit code 0
```

d) Input instance with 5 elements for which this method would exhibit the best case runtime is as follows:

[5,5,-19,0,3]
The reason being is that the two duplicates are the two starting points (or pointers) for *i* and *j*. So the duplicates would be found on the first instance of the search and exitted. It would not be necessary to run a full sweep search in all elements of the array. O(1) runtime for the best case

e) The input instance with 5 elements for which this method would exhibit the worst case runtime is as follows:

[125,1,-127,346,0,0] or in the event of an array without any duplicates.

The reason being is that the two duplicates are positioned in the first and last position. The pointers *i* and *j* will have to iterate over all the elements in the array. Or in the event of no duplicates in the array, all elements must be iterated and compared before the method returns false. O(n^2) is the worst case

f) The input instance that would take longer from the two input instances given in this part of the question is : [10,0,5,3,-19,5]

The reason being is that are more iterations and comparisons on this array before the duplicates are found. I will show this in my “Debugging Test Code” and “Outputs” section below.

For now, the simplified justification is that : the pointer *i* is pointed to the first element of the array (element[0] = 10) and pointer *j* pointed to the second element (element[1] = 0). Pointer *i* will remain pointed to element[0], until pointer *j* iterates over the array (where both *i* and *j* meet – element[0]). Pointer *i* will iterate 1 position forward and will remain there until *j* iterates again the array completely, and *i* iterates again 1 position forward. The process repeates itself, until pointer *i* is pointing at element[2], which takes the value of 5 and pointer *j* pointing at element[5], which also takes the value of 5.

Debugging Test Code

I will be providing 2 types of test for part f), and they are runtime test for the two input instances given and the iteration for both pointers(***i*** and ***j***) in the arrays.

Runtime Test Code

```
public class CTJava3 {
    public static void main(String args []) {
        // Timing first array runtime in nanoseconds
        final long startTime = System.nanoTime();
        int Array[] = {10, 0, 5, 3, -19, 5};
        int Array1[] = {0, 1, 0, -127, 346, 125};
        System.out.println(containsDuplicates(Array));
        final long endTime = System.nanoTime();
        final long totalTime = endTime - startTime;
        System.out.println( "The Array list takes " + totalTime + " nanoseconds as
runtime");
        // Finished timing the first array
        //Timing second array in nanoseconds
        final long startTime1 = System.nanoTime();
        System.out.println(containsDuplicates( Array1 ));
        final long endTime1 = System.nanoTime();
        final long totalTime1 = endTime1 - startTime1;
        System.out.println( "The Array1 list takes " + totalTime1 + " nanoseconds as
runtime" );
    }
    public static boolean containsDuplicates ( int elements[] ) {
        for (int i = 0; i < elements.length; i++) {
            for (int j = 0; j < elements.length; j++) {
                if ( i == j ) {
                    continue;
                }
                if ( elements[i] == elements[j] ) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Output

/usr/lib/jvm/java-8-oracle/bin/java ...

true
The Array list takes 492655 nanoseconds as runtime
true

The Array1 list takes 35269 nanoseconds as runtime

Process finished with exit code 0

Each time I run the runtime test codes I get different times, but as a reference point “Array1” which is [0,1,0,-127,346,125] has the lowest runtimes. So “Array” runtime is ever so slightly longer than “Array1”. The list is small with only 6 elements, if in the event of a list array with a much larger input elements the runtime differences would be significantly bigger.

Iteration Positions For Both Pointers Test Code

To best understand my justification in part f) I will provide code to show the iteration of both variables ***i*** and ***j***.

```
public class CTJava3test3 {
    public static void main ( String args[] ) {
        int Array[] = {10, 0, 5, 3, -19, 5};
        int Array1[] = {0, 1, 0, -127, 346, 125};
        // Adapted from : " http://tutorials.jenkov.com/java/arrays.html "
        System.out.println("The iteration of 'i' and 'j' on the Array: \n");
        containsDuplicates( Array );
        System.out.print("\n\n");       // Give a couple of blank lines
        System.out.println("The iteration of 'i' and 'j' on the Array1: \n ");
        containsDuplicates( Array1 );
    }
    public static boolean containsDuplicates ( int elements[] ) {
        for (int i = 0; i < elements.length; i++) {
            for (int j = 0; j < elements.length; j++) {
                if ( i == j ) {
                    continue;
                }
                if ( elements[i] == elements[j] ) {
                    return true;
                }
                    System.out.println( "elements[i]: " + elements[i] + ",
elements[j]: " + elements[j]);
            }
            }
            return false;
        }
}
```

Output For Test Code

/usr/lib/jvm/java-8-oracle/bin/java ...

The iteration of 'i' and 'j' on the Array:

elements[i]: 10, elements[j]: 0
elements[i]: 10, elements[j]: 5
elements[i]: 10, elements[j]: 3
elements[i]: 10, elements[j]: -19
elements[i]: 10, elements[j]: 5
elements[i]: 0, elements[j]: 10
elements[i]: 0, elements[j]: 5

elements[i]: 0, elements[j]: 3
elements[i]: 0, elements[j]: -19
elements[i]: 0, elements[j]: 5
elements[i]: 5, elements[j]: 10
elements[i]: 5, elements[j]: 0
elements[i]: 5, elements[j]: 3
elements[i]: 5, elements[j]: -19

The iteration of 'i' and 'j' on the Array1:

elements[i]: 0, elements[j]: 1

Process finished with exit code 0

Literature Review

Data Structures & Algorithms in Java - Michael T. Goodrich, Robert Tamssia, Michael H. Goldwasser

Java – An introduction to Problem Solving and Programming – Walter Savitch

Java – The Complete Reference (Ninth Edition) – Herbert Schildt

Data Structures & Problem Solving Using Java (Fourth Edition) – Mark Allen Weiss

Algorithms In A Nutshell – George T. Heineman, Gary Pollice & Stanley Selkow

Introduction To Algorithms (Third Edition) - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Algorithms Unlocked – Thomas H. Cormen