Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
Midterm Examination

Guy-Vincent Jourdan, Mehrdad Sabetzadeh, and Marcel Turcotte

March 2020, duration: 2 hours

Identification

Last name: ______________________ First name: ______________________

Student #: ____________ Seat #: ____ Signature: ______________ Lab Section: ______________

Instructions

1. This is a closed book examination.
2. No calculators, electronic devices or other aids are permitted.
   (a) Any electronic device or tool must be shut off, stored and out of reach.
   (b) Anyone who fails to comply with these regulations may be charged with academic fraud.
3. Write your answers in the space provided.
   (a) Use the back of pages if necessary.
   (b) You may not hand in additional pages.
4. Write comments and assumptions to get partial marks.
5. Beware, poor hand-writing can affect grades. Do not use a red pen.
6. Do not remove pages or the staple holding the examination pages together.
7. Wait for the signal to start of the examination.

Marking scheme

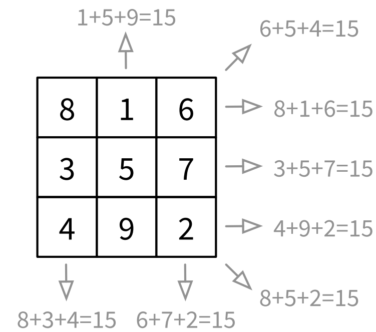
Question	Maximum	Result
1	20	
2	10	
3	15	
Total	**45**	

Question 1 (20 marks)

For this question, we develop a representation for a **Magic Square**. In its most basic form, a **Magic Square** is an $n \times n$ grid filled with the numbers $1, 2, \ldots, n^2$, such that each row and each column, as well the two main diagonals all have the same sum, called the **magic constant** M, where:

$$M = \frac{n(n^2+1)}{2}$$

In the example on the right, $n = 3$ and $M = 15$.



For this question, you must complete the implementation of the class **MagicSquare** on page 4. Just like in assignment 1, **MagicSquare** uses a one-dimensional array to store the values of the grid.

A. (4 marks) In the class declaration on page 4, declare three instance variables:

(a) A reference variable used to designate a one-dimensional array of **int** values. The designated array will be used to store the values of the grid.

(b) A variable to store the size of the grid, where the size represents both the number of rows and the number of columns of the magic square (3 in the example above).

(c) A variable to store the magic constant, as defined above (15 in this example).

B. (8 marks) In the class declaration on page 4, implement the constructor. Make sure that your implementation complies with the following guidelines:

(a) The constructor stores the values of this magic square specified by the parameter. As illustrated by the test program on the next page, an object of the class **MagicSquare** is not affected by subsequent changes to the array of values passed as an argument to the constructor. Herein, you can assume that the the value of the parameter will not be **null** and the given array forms a square. In other words, the length of the array is a power of 2 for some integer $n \geq 3$.

(b) The constructor calculates and stores the size of the **MagicSquare**, where the size represents both the number of rows and the number of columns.

(c) Finally, you must compute and store the value of the magic constant.

C. (4 marks) Implement the following three (3) access methods.

(a) Implement the instance method **getSize()** that returns the size of **this MagicSquare**, where the size is the number of rows and columns.

(b) Implement the instance method **getConstant()** that returns the value of the magic constant, as defined above.

(c) Finally, implement the method **getValue(int row, int column)**. The method returns the value stored at location **row** and **column** of the grid. The first **row** and **column** have index 0. You can assume that the values will be valid. Be careful, this a two-dimensional coordinate whereas the information is stored in our one-dimensional array.

D. (4 marks) Implement the helper method **isMagicDiagonals()**. This method returns **true** if and only if both sums, the sum of the elements on the main diagonal, and the sum of the elements on the main anti-diagonal, are equal to the magic constant. The method is called by the method **isMagic()**. You are not required to implement the other two methods called by **isMagicSquare()**. Specifically, you do not have to implement **isMagicRows()** and **isMagic()**.

The program below shows you the intended use and behaviour for the class **MagicSquare**.

```
MagicSquare s1, s2;

int[] values = new int[] {1,8,6,3,5,7,4,9,2};

s1 = new MagicSquare(values);

values[0] = 8;
values[1] = 1;

s2 = new MagicSquare(values);

System.out.println(s1);
System.out.println(s1.getSize());
System.out.println(s1.getConstant());
System.out.println(s1.isMagic());

System.out.println();

System.out.println(s2);
System.out.println(s2.getSize());
System.out.println(s2.getConstant());
System.out.println(s2.isMagic());
```

The execution of the above program displays the following on the console:

```
 1  8  6
 3  5  7
 4  9  2
3
15
false

 8  1  6
 3  5  7
 4  9  2
3
15
true
```

Reminders: In Java, **Math.sqrt(double a)** can be used to calculate the square root of a number, whereas **Math.pow(double a, double)** returns the value of the first argument raised to the power of the second one. You can force types from **double** to **int**, if you need to.

```
public class MagicSquare {

    // Part A. Instance variables

    // Part B. Constructor

    public MagicSquare(int[] values) {

    }
```

```
// Part C. Access methods
```

```
    // Part D. isDiagonals()

    public boolean isMagic() {
        return isMagicRows() && isMagicColumns() && isMagicDiagonals();
    }

    // The source code for the other methods, including isMagicRows()
    // and isMagicColumns(), is hidden.

    private boolean isMagicDiagonals() {

    }
}
```

Question 2 (10 marks)

You must complete the implementation of the class method **reverse(String[] a)**. After a call to this method, the values of the array designated by the parameter **a** are in the reverse order, as shown in the example below. Your implementation **must** make use a stack to do the work of reversing the order of the elements. Herein,

- **Stack** is an interface, with the usual methods: **push**, **pop**, and **isEmpty**;
- **StackImplementation** is a class that implements the interface **Stack**. This implementation can store an arbitrarily large number of elements. You don't need to implement this class.

The small Java program below illustrates the expected behaviour.

```java
String[] alphabet;

alphabet = new String[] {"alpha", "bravo", "charlie", "delta", "echo"};

System.out.println(Arrays.toString(alphabet));

StringUtils.reverse(alphabet);

System.out.println(Arrays.toString(alphabet));
```

Its execution produces the following output:

```
[alpha, bravo, charlie, delta, echo]
[echo, delta, charlie, bravo, alpha]
```

Give your solution in the box provided for this purpose on the following page.

```
public class StringUtils {

    public _________________ _________________ reverse(String[] a) {


        Stack<_________________> s;





























    }
}
```

Question 3 (15 marks)

For this question, you must implement a class called **Token**. An object of the class **Token** can either store the reference of **String** or a primitive value of type **int**. Accordingly, the class has two constructors, one for each type. Here is a detailed specification of its implementation.

- **A.** (1 mark) Write the class declaration for **Token**.
- **B.** (3 marks) Declare the necessary instance variables so that an object of the class **Token** can store the reference of a **String**. It can store a value of the primitive type **int** (the value must be stored as an **int**). Finally, the object must know if it is storing the reference of a **String** or a value of type **int**.
- **C.** (2 marks) Give the implementation of the constructor **Token(String value)**. The constructor stores the reference of the object designated by the parameter **value**. The object must remember that it stores a value of type **String**.
- **D.** (2 marks) Give the implementation of the constructor **Token(int value)**. The constructor stores the value of the parameter **value**. The object must remember that it stores a value of type **int**.
- **E.** (1 mark) Implement the instance method **isString()** that returns **true** if this object stores a value of type **String**, and **false** otherwise.
- **F.** (2 marks) Implement the instance method **toString()** that returns a **String** representation of this object. See below for examples.
- **G.** (4 marks) Implement the class method **equals(Token a, Token b)** that returns **true** if the objects designated by the parameters **a** and **b** are logically equivalent (have the same content) and **false** otherwise.

```java
Token a, b, c;

a = new Token("alpha");
b = new Token("alpha");
c = new Token(42);

System.out.println(a);
System.out.println(b);
System.out.println(c);

System.out.println();

System.out.println("a.isString() is " + a.isString());
System.out.println("c.isString() is " + c.isString());

System.out.println();

System.out.println("Token.equals(a,b) is " + Token.equals(a,b));
System.out.println("Token.equals(b,c) is " + Token.equals(b,c));
System.out.println("Token.equals(c,null) is " + Token.equals(c,null));
```

```
Token: alpha
Token: alpha
Token: 42
```

```
a.isString() is true
c.isString() is false

Token.equals(a,b) is true
Token.equals(b,c) is false
Token.equals(c,null) is false
```

Give the implementation of the class **Token** in the space below.