Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations
and is a required part of the degree assessment.
Laboratory work must be handed in for marking by the due date.
Late submissions may not be marked.

Exercise: 7	Working: Individual
Title: Loops in Java	
Issue date: 22nd November 2010	Due date: 29th November 2010
System: Linux	Language: Kenya/Java

Aim

- To develop simple programs requiring **for loops** in **Kenya/Java**.
- To practice working with arrays in **Kenya/Java**.

Introduction

The laboratory exercise is designed to increase your skill in writing programs with loops and to give practice in program testing.

The Problem

This exercise requires you to write four programs:

- **SWTriangle.k/SWTriangle.java** which displays a right-angled triangle, made out of asterisks, of specified size, with the right angle in the bottom left-hand ("South West") corner, e.g. for a size of 5:

```
*
* *
* * *
* * * *
* * * * *
```

  Once again the height and width (of the base) should be the same, counting in asterisks.

- **NETriangle.k/NETriangle.java** which displays a right-angled triangle of specified size with the right angle in the top right-hand ("North East") corner, e.g. for a size of 5:

```
* * * * *
  * * * *
    * * *
      * *
        *
```

- **MatrixMultiply.k/MatrixMultiply.java** which prompts the user for the dimensions and elements of two Matrices, echoes them on the screen, multiplies them together and finally outputs the resulting Matrix.

  Your program, including prompts, should produce output like that below.

```
Enter the number of rows and columns in the Matrix followed by the data:
3 4 1 2 3 4 5 6 7 8 9 1 2 3

3X4 Matrix
1 2 3 4
5 6 7 8
9 1 2 3

Enter the number of rows and columns in the Matrix followed by the data:
4 3 1 2 3 4 5 6 7 8 9 1 2 3

4X3 Matrix
1 2 3
4 5 6
7 8 9
1 2 3

The first Matrix multiplied by the second Matrix gives:

3X3 Matrix
34 44 54
86 112 138
30 45 60
```

  To multiply two Matrices you first start with the first row of the first Matrix you take each element of each column in turn and multiply it by each element in the first column of each row in the second Matrix and sum the result. This gives the first element of the first row of the result Matrix. You then multiply the first row of the first Matrix by any subsequent columns of the second Matrix and when complete this gives you the first row of the result Matrix. You get the subsequent rows of the result Matrix by multiplying the second Matrix by any subsequent rows in the first Matrix. In symbols:

$$C_{ik} = \sum_j A_{ij} B_{jk}$$

- **MagicSquares.k/MagicSquares.java** Magic squares are squares of size NxN with NxN locations. a magic square is constructed by writing the numbers 1..NxN into each location such that the rows, columns and diagonals all add up to the same value. For example a 5x5 magic square will have 25 locations and contain the numbers 1..25.

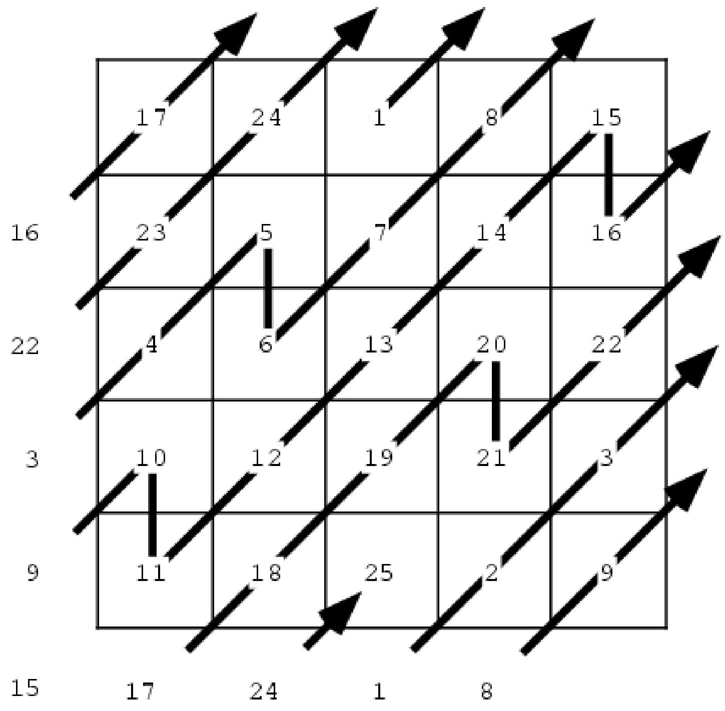
```
Enter square size (must be >= 3:
5
--------------------------
| 17 | 24 |  1 |  8 | 15 |
--------------------------
| 23 |  5 |  7 | 14 | 16 |
--------------------------
|  4 |  6 | 13 | 20 | 22 |
--------------------------
| 10 | 12 | 19 | 21 |  3 |
--------------------------
| 11 | 18 | 25 |  2 |  9 |
--------------------------
```

- There are 3 algorithms for constructing a magic square of length n, for the cases when n is odd, doubly even (multiple of 4), and singly even (multiple of 2, not 4). We shall start by considering the odd order magic square. You should complete the odd and doubly even order. The single even order magic square is left as an optional unassessed part of the exercise.

- **Odd Order Magic Square**

  In this you start by placing a 1 in any location (by convention it seems to be the centre column on the top row), then continue moving in an upward-right diagonal direction, placing successive numbers in each cell you arrive at. When reaching the edge of a table, you simply wrap round to the other side of it. When eventually reaching a square that's already been filled before, you simply move down a square instead of up-right, place the next number, then carry on in an up-right direction. You will never reach the situation where **both** the square to the above-right, and the square below are both occupied, so these situations cover everything your algorithm needs to do. You simply terminate when you have filled in n^2 cells.

Figure 1: Odd Order Magic Square



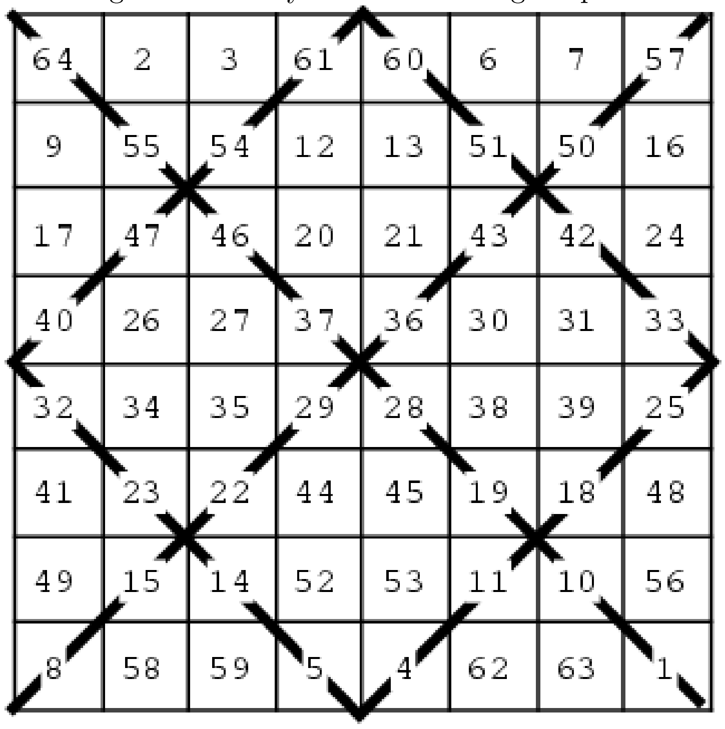
- **Doubly Even Order Magic Square**
  While the odd order case is arguably easier to code than this one, this case is probably easier to do by hand on paper. The first step is simply to fill in the whole square row by row (left to right) with successive integers, leaving you with a simple NxN grid of increasing numbers, e.g. for N=8:

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	20	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

  From here, since n is a multiple of 4, we can divide the whole grid into a number of 4x4 subsquares (in the above case, since our grid is 8x8, we have four 4x4 sub squares). In each of these subsquares, we now draw lines through the leading diagonals (see the diagram).

  Now, for every number m which has a line through it, we replace its value with its complement $(1 + n^2 - m)$. Thus, in the attached picture, the first cell in the grid before the replacements were done was 1 (which was replaced by 64), and similarly the 61, 60 and 57 on the first row had values 4,5, and 8 respectively before being replaced.

Figure 2: Doubly Even Order Magic Square



64	2	3	61	60	6	7	57
9	55	54	12	13	51	50	16
17	47	46	20	21	43	42	24
40	26	27	37	36	30	31	33
32	34	35	29	28	38	39	25
41	23	22	44	45	19	18	48
49	15	14	52	53	11	10	56
8	58	59	5	4	62	63	1

Submit by Monday 29th November 2010

What to do

- **SWTriangle.k/SWTriangle.java** Write a program which displays a ”South West” triangle using appropriate support methods and *for* loops. Your program should prompt the user for the size of the triangle anf then output the triangle.
- **NETriangle.k/NETriangle.java** Write a program which displays a ”North East” triangle using appropriate support methods and *for* loops. Your program should prompt the user for the size of the triangle anf then output the triangle.
- **MatrixMultiply.k/MatrixMultiply.java** To represent a Matrix you should use a 2 dimensional array of integer. The syntax for 2 dimensional arrays is:
  `int [] [] Matrix = new int [N] [M] ;` where `N` and `M` are replaced by the number of elements used in each array. By convention the first array specifies the number of rows and the second array the number of columns. Write your program using nested `for` loops to do the Matrix multiplication.
- Java allows arrays to be declared at run-time and this allows you to declare your array after you have read in the Matrix dimensions.
- You can find out the size of an array using the built in function `length`.
- When you multiply two matrices of size $M \times N$ and $N \times P$ the result Matrix will be of size $M \times P$. The number of columns in the first Matrix must be equal to the number

of rows in the second Matrix. However you can assume that the input supplied to your program is correct and you do not need to do any error checking.

- **MagicSquares.k/MagicSquares.java** Copy the skeleton file **MagicSquares.k** into your working directory using the command:
  exercise 7.
- Note that this program has a **main** method and various support methods provided but that the methods **oddOrder**, **doublyEvenOrder** and **singleEvenOrder** are left as stubs. The program will compile and run but it will not construct any magic squares until you have completed the code for these methods.
- As the code for doing I/O is provided you should not alter this or add any additional I/O. This is for autotesting purposes and you should use the method **printSquare** for displaying the completed magic square.
- You are also provided with a method - **checkMagicSquare** that checks that a magic square is correctly formed. This method is used in the main method to ensure that only correctly formed magic squares are displayed. If you wish to comment out calls to this method for debugging purposes you should make sure that it is reinstated in any code that you submit.
- You are also provided with a method **mod** which you should use when you want to wrap around the array when you are constructing odd order magic squares - **mod(-1, 8)** will return **7** which is the value needed for wrapping around wheras Java's built in **%** returns **-1**.
- Examine the file **MagicSquares.k** and write code to fill in the stubs for **oddOrder** and **doublyEvenOrder**.
- You need to consider the design of your methods **oddOrder** and **doublyEvenOrder** and you will find it much simpler to code these methods if you use a number of simple auxiliary methods. For example a method for returning the co-ordinates of the next square to be filled in for **oddOrder**. For **doublyEvenOrder** you may find it helpful to have an auxiliary method that processes each 4x4 sub-square.

Unassessed

- **Singly Even Order Magic Square**
  A singly even order magic square is much more complicated to code than the previous two you have written but is included because it completes the program and would allow you to construct a magic square of any size.

  The algorithm first works by splitting the square into 2x2 subgrids (see the diagram above). Since the whole grid is singly even, it will contain an **odd** number of 2x2 subgirds on each side. For example a 10x10 grid contains 5x5 subgrids of size 2x2.

  We now need to consider how to fill in each subsquare within the whole square. The algorithm works by filling in a whole subgrid before moving onto the next subgrid, so

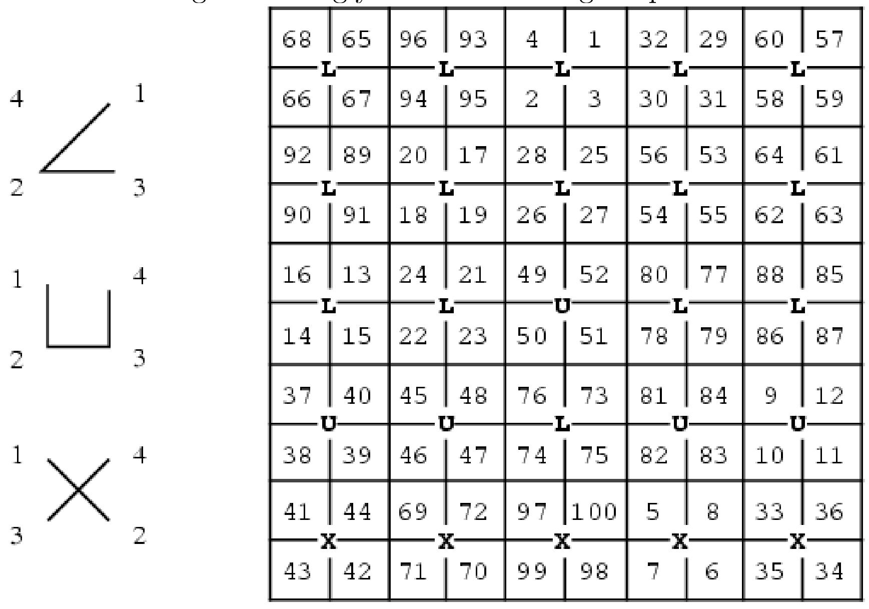


Figure 3: Singly Even Order Magic Square

68	65	96	93	4	1	32	29	60	57
66	67	94	95	2	3	30	31	58	59
92	89	20	17	28	25	56	53	64	61
90	91	18	19	26	27	54	55	62	63
16	13	24	21	49	52	80	77	88	85
14	15	22	23	50	51	78	79	86	87
37	40	45	48	76	73	81	84	9	12
38	39	46	47	74	75	82	83	10	11
41	44	69	72	97	100	5	8	33	36
43	42	71	70	99	98	7	6	35	34

the first thing we need to decide is what order we do the subgrids in.

To do this, we apply the first algorithm above (for odd order) to the whole square, treating each 2x2 subgrid as a **single** cell. The number this algorithm assigns to each 2x2 block of cells represents the order in which we attempt to fill in each subgrid. Thus, since the odd-order algorithm starts in the centre cell of the top row, then for the singly-even algorithm, the first 2x2 subgrid we fill in is the centre 2x2 subgrid on the top row of subgrids in our main square. We fill in successive subgrids based on the number assigned to the corresponding cell in the odd-order algorithm.

Now the only question that remains is how (ie. in what order) we fill in each 2x2 subgrid, before moving on to the next. Well, this is where the letters 'L', 'U' and 'X' come into the algorithm. Each subgrid within the main square is assigned a letter, either 'L', 'U' or 'X' that determines in which order the cells within the subgrid are filled in. The letters are assigned to each subgrid as follows:

– All subgrids up to and including the centre row of subgrids are assigned the letter 'L'

– The row of subgrids below the centre row are all assigned the letter 'U'

– The rest of the rows below are assigned the letter 'X'

– Finally, the subgrid at the centre of the square swaps its letter (currently 'L'), with the subgrid below it (a 'U').

For a 10x10 square like the one given in the diagram above, you can see the assignment of characters to each 2x2 subgrid. Once letters have been assigned to each subgrid, we also know which order we fill in the subgrids, we just fill in each subgrid individually based on the letter assigned to it... You can see what order each character dictates from the diagrams to the left side of the diagram above. (Their shapes are suggestive of the letters 'L', 'U', 'X' respectively.)

Submission

- Before submitting, you should **test your programs on a wide range of user inputs**.

  Submit your programs:
  SWTriangle.k/SWTriangle.java, **NETriangle.k/NETriangle.java** , **MatrixMultiply.k/MatrixMultiply.java**, and **MagicSquares.k/MagicSquares.java**, using the **CATE** system in the usual way.

Assessment

SWTriangle	1
NETriangle	1
MatrixMultiply	1.5
MagicSquares	1.5
Design, style, readability	5
Total	10