NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

Practical Examination for Semester 1, AY2006/7
CS1101 — Programming Methodology

4 November 2006 Time Allowed: 2 hours 30 minutes

INSTRUCTION TO CANDIDATES

1. This is an **OPEN book** examination. You are allowed to bring to the lab hard copies of notes and books. However, you are **NOT** allowed to bring into the lab digital/communication devices (such as laptop, cell phone, PDA, etc.) and storage devices (hard disk, thumb drive, CD, etc.). There are **2** exercises in **15** printed pages. The marking scheme for each exercise is as given – you should exercise the usual good programming practices (comments, proper indentation, meaningful naming convention, good design of classes etc.).

2. **Login to PC.** Please use the given ID (**cs1101**) and password (**PnevEA4**) to log into your assigned PC with domain as **Computing**. Your NUSNET id will not be used as there is no connection to NUSNET. The PC comes with our recommended IDE **DrJava**. Java **manuals** are available on your desktop. We will not provide software/editor that are not already in the PC and we do not guarantee the availability of such software/editor. Please login to the CourseMarker (see paragraph 3) before invoking DrJava.

3. **Login to CourseMarker.** Click on the CourseMarker shortcut **c:\cmc\cs1101xPE.bat, c:\cmc\cs1101yPE.bat or c:\cmc\cs1101zPE.bat** to log into the CourseMarker. Please use the **newly assigned password** (given to you just before the practical exam) to perform the login. Once you have gained access to the CourseMarker, please click on the setup button to download the necessary files and some dummy files – please **do not** delete any of these files though you are not using them in your programming, you need them when submitting your work to the CourseMarker. With this, you are ready to start programming.

4. **Working Directory.** Please create your files in the directory **"c:\Documents and Settings\cs1101\My Documents\CMhome\studentArea\yourID\exerciseDIR"** (where **yourID** is your CourseMarker login id and **exerciseDIR** is cs1101xPEex1, cs1101xPEex2, cs1101yPEex1, cs1101yPEex2 and so forth) with the specified file names as stated in the questions. Only the **specified files** in the **specified directory** will be submitted to the CourseMarker when you click on the submit button of the CourseMarker client.

5. **Submission.** You are given **10** submissions. Only the **last version** will be graded. Note once again that the system **only** submits files of the specified filenames in the specified directory – all other files will not be captured by the CourseMarker. Please ensure you have submitted your work before the end of the examination – to avoid unnecessary stress to yourself, you should not wait till the very last minute of the exam to submit the work at the same time as many other students. The CourseMarker system will be closed shortly after the practical examination.

6. **Leaving the lab.** Please **log out** from the PC before you leave the lab. But please **do not** shut down the machine.

7. **No Communication.** You are NOT to communicate in any way (sharing of files, PCs, calculators, send emails, receive emails, use of ICQ, etc.) with anyone, other than the invigilators, during the practical examination. You must allow the invigilator access to your PC on request.

ALL THE BEST!

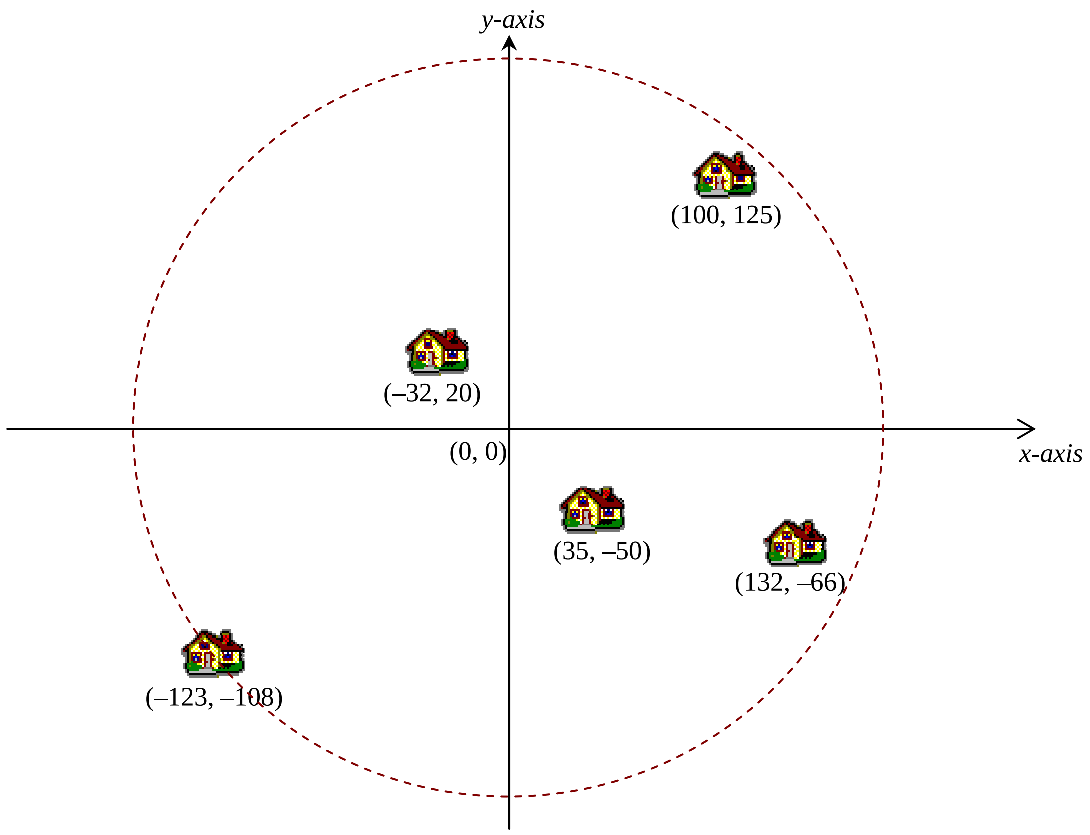
CS1101 AY2006/7 Semester 1
Practical Exam (PE)

Important notes

- There are **2** exercises in this PE. Exercise 1 constitutes 40%; Exercise 2 60%.
- You are given **10** submissions for each exercise. Please do not wait till the last minute to submit your programs. Make your first submission early.
- You are advised to spend some time thinking over the tasks to design your algorithms, instead of writing the programs right away.
- Manage your time well! Do not spend excessive time on any exercise or sub-task.

Exercise 1: Houses in a town (Town.java): 40%

In a certain country, houses in a town are located on two-dimensional coordinates with reference to the Town Central which is located at the origin (0, 0). The coordinates are integer values. An example of a small town with 5 houses are shown in the diagram below (not drawn to scale).



Write a program **Town.java** that reads in a list of house locations of a town into a **Point** array and performs the following sub-tasks:

1. Find and report the number of unique houses in the town. (Some of the houses may appear more than once in the input list.)

   Do not sort your array (there is no need to sort).

2. Find and report the location of the house that is furthest away from the Town Central (the origin). If there are more than one such house, just report anyone of them. In the above example, the house that is furthest from the Town Central is located at (–123, –108).

   The distance of a house located at (x, y) from the origin is $\sqrt{x^2 + y^2}$.

3. Compute and report the area occupied by the town. The area occupied by the town is πr^2, which is the area of the circle with centre at the origin, and radius r the distance of the furthest house from the origin. You should use **Math.PI** for π, and display the area accurate to 4 decimal places. In the above example, the town area, shown as interior of the dotted circle, is 84172.6920.

Input format

The input consists of a line that contains n, a positive integer that indicates the number of house locations, followed by n lines of data, where each line contains 2 integers that indicate the x- and y-coordinates of a house location. Note that a house location may appear more than once in the input.

You may assume that there are at most 20 house locations in the input, and the x- and y-coordinates are integers in the range [–1000, 1000].

Sample runs

A sample run for the above example is shown below. The output is shown in **bold**.

Sample run #1:

```
      $ javac Town.java
      $ java Town
      8
      -32 20
      100 125
      132 -66
      132 -66
      -123 -108
      35 -50
      35 -50
      -32 20
      Number of houses = 5
      Furthest house at java.awt.Point[x=-123,y=-108]
      Town area = 84172.6920
```

Another sample run is shown below. There are two houses that are equally furthest from the Town Central: (-20, 20) and (20, -20); only one of them needs to be reported.

Sample run #2:

```
      $ java Town
      4
      -20 20
      15 10
      20 -20
      15 10
      Number of houses = 3
      Furthest house at java.awt.Point[x=-20,y=20]
      Town area = 2513.2741
```

Important notes

- You are to use a Point array to store the house locations. You may refer to http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Point.html
- If you are unable to do subtask 1, you should still carry on to subtasks 2 and 3, which are independent of subtask 1.
- You are to submit a program that can be compiled. You will lose 25 marks out of 50 marks right away if your program cannot be compiled (5 marks for compilability; 20 marks for correctness).
- Do not spend more than 1 hour on this task.
- A template program **Town.java** is provided on the next page and it is also available in your Working Directory (refer to paragraph 4 on page 1).
- Sample inputs (**town_sample.in1** and **town_sample.in2**) and outputs (**town_sample.out1** and **town_sample.out2**) are also available in your Working Directory.

Marking scheme: Total of 50 marks

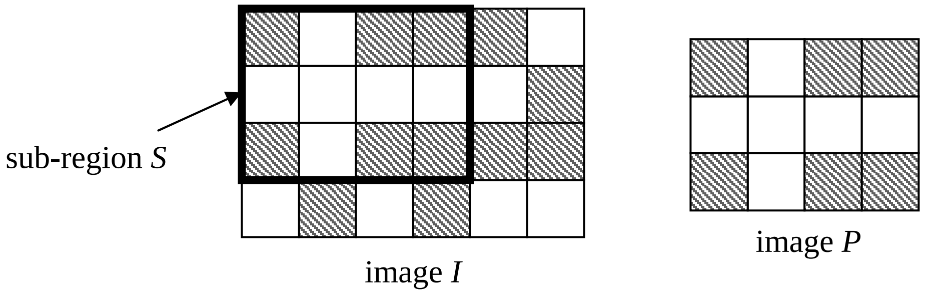
1. Can program be compiled? 5 marks
2. Use of array and values read correctly into array? 10 marks
3. Style (including your particulars, use of constant, comments, spacing/indentation, choice of variable names): 15 marks
4. Correctness of result: 20 marks (5 test data sets; 4 marks for each data set.)
5. The marks for this task will constitute **40%** of the total marks.

```
// CS1101 (AY2006/7 Semester 1)
// PE Exercise 1
// Town.java
// Author:
// Matriculation number:
// Lecture group (X, Y or Z):
// Discussion group (1-7):
/**
 * Houses in a town are represented in 2D coordinates.
 * This program reads a list of house coordinates, determines
 * the number of unique house locations, find the house that
 * is furthest from the Town Central (origin), and computes
 * the area of the town.
 */

import java.util.*;
import java.text.*;
import java.awt.*;

public class Town {

   public static void main (String[] args) {

      // fill in your code here

   }
}
```

Exercise 2: Binary image and pattern detection (PatternDetection.java & BinaryImage.java): 60%

A *binary image* is a 2D image where each pixel has a value of 0 or 1. The value 0 represents black colour and value 1 white colour.

Given a binary image *I* and a smaller binary image *P*, a rectangular *sub-region* of *I matches P* if the binary image in the sub-region is *identical* to *P*. In the following example, the sub-region *S* matches the image *P*.



The image *P* is called a *pattern image*.

Write a Java program **PatternDetection.java** that reads in a binary image *I*, a pattern image *P*, and finds all the sub-regions in *I* that match *P*. Your program must also find sub-regions in *I* that match the rotated versions of *P* by 90°, 180° and 270° clockwise, respectively. The following diagram shows the image *P* and its rotated versions.

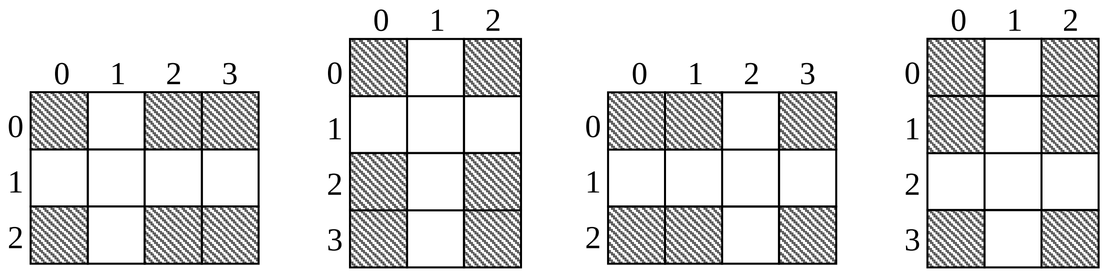


image *P* — image *P* rotated 90° clockwise — image *P* rotated 180° clockwise — image *P* rotated 270° clockwise

Each sub-region is described by the pixel locations where its top-leftmost pixel and bottom-rightmost pixel are located in *I*. A pixel location is described by the coordinates (*row*, *column*). The top row of *I* is Row 0 and the leftmost column of *I* is Column 0. For example, given the image *I* and pattern *P* shown in the following diagram, the three sub-regions in image *I* that match the pattern *P* or its rotated versions are

- `(0, 0)--(3, 2)`
- `(0, 3)--(2, 6)`
- `(2, 0)--(5, 2)`

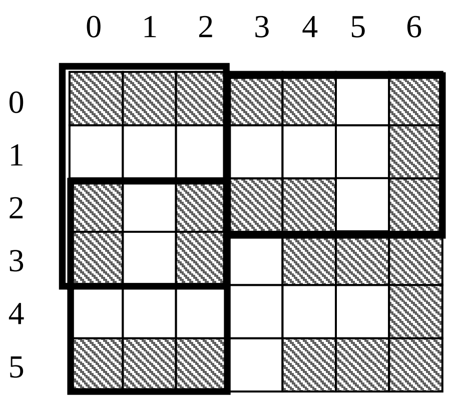


image *I*

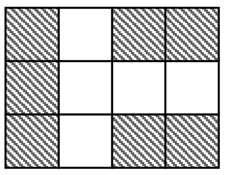


image *P*

Your program must make use of the **BinaryImage** class defined in the file **BinaryImage.java**. The partial code for PatternDetection.java and BinaryImage.java are given to you, and you need to fill in your code at places indicated in the files. You may write additional private methods in BinaryImage.java if necessary.

Input format
Please see the sample runs to find out the input format.

Output format
After all the inputs have been read, your program should output the followings in the order shown:

1. The input image.
2. The input pattern image, and all the sub-regions that match it. Each sub-region is displayed as a pair of pixel locations. The sub-regions must be output such that the row numbers of the top-leftmost pixel locations are in non-descending order. If the row numbers are the same, the column numbers must be in increasing order.
3. The pattern image rotated 90° clockwise, and all the sub-regions that match it. See Note (2) for the output order of the sub-regions.
4. The pattern image rotated 180° clockwise (i.e. rotated 90° two times), and all the sub-regions that match it. See Note (2) for the output order of the sub-regions.
5. The pattern image rotated 270° clockwise (i.e. rotated 90° three times), and all the sub-regions that match it. See Note (2) for the output order of the sub-regions.
6. The total number of sub-regions that match the pattern or its rotated versions.

Note that a sub-region may match more than one version of the pattern image if the pattern image is rotationally symmetrical. In this case, the sub-region must be output multiple times, one for each version of the pattern image that it matches.

See the sample runs to find out the exact output format.

Sample runs

Two sample runs are shown below. The first one corresponds to the example given above. The outputs are shown in **bold**.

Sample run #1:

```
$ javac PatternDetection.java BinaryImage.java
$ java PatternDetection
Enter image height: 6
Enter image width: 7
Enter image row 0: 0 0 0 0 0 1 0
Enter image row 1: 1 1 1 1 1 1 0
Enter image row 2: 0 1 0 0 0 1 0
Enter image row 3: 0 1 0 1 0 0 0
Enter image row 4: 1 1 1 1 1 1 0
Enter image row 5: 0 0 0 1 0 0 0
Enter pattern height: 3
Enter pattern width: 4
Enter pattern row 0: 0 1 0 0
Enter pattern row 1: 0 1 1 1
Enter pattern row 2: 0 1 0 0
Image:
 0 0 0 0 0 1 0
 1 1 1 1 1 1 0
 0 1 0 0 0 1 0
 0 1 0 1 0 0 0
 1 1 1 1 1 1 0
 0 0 0 1 0 0 0

Pattern 0:
 0 1 0 0
 0 1 1 1
 0 1 0 0
Matching sub-regions:

Pattern 1:
 0 0 0
 1 1 1
 0 1 0
 0 1 0
Matching sub-regions:
(0, 0)--(3, 2)

Pattern 2:
 0 0 1 0
 1 1 1 0
 0 0 1 0
Matching sub-regions:
(0, 3)--(2, 6)
```

```
Pattern 3:
 0 1 0
 0 1 0
 1 1 1
 0 0 0
Matching sub-regions:
(2, 0)--(5, 2)

Total number of matching sub-regions: 3
```

Sample run #2:

```
$ javac PatternDetection.java BinaryImage.java
$ java PatternDetection
Enter image height: 7
Enter image width: 8
Enter image row 0: 0 0 0 0 0 0 0 0
Enter image row 1: 0 1 1 1 1 0 0 0
Enter image row 2: 0 1 0 0 1 0 0 0
Enter image row 3: 0 1 0 0 1 1 1 0
Enter image row 4: 0 1 0 0 0 0 1 0
Enter image row 5: 0 1 1 1 1 1 1 0
Enter image row 6: 0 0 0 0 0 0 0 0
Enter pattern height: 3
Enter pattern width: 3
Enter pattern row 0: 0 1 0
Enter pattern row 1: 0 1 1
Enter pattern row 2: 0 0 0
Image:
 0 0 0 0 0 0 0 0
 0 1 1 1 1 0 0 0
 0 1 0 0 1 0 0 0
 0 1 0 0 1 1 1 0
 0 1 0 0 0 0 1 0
 0 1 1 1 1 1 1 0
 0 0 0 0 0 0 0 0

Pattern 0:
 0 1 0
 0 1 1
 0 0 0
Matching sub-regions:
(2, 3)--(4, 5)
(4, 0)--(6, 2)

Pattern 1:
 0 0 0
 0 1 1
 0 1 0
Matching sub-regions:
(0, 0)--(2, 2)
```

```

Pattern 2:
 0 0 0
 1 1 0
 0 1 0
Matching sub-regions:
(0, 3)--(2, 5)
(2, 5)--(4, 7)

Pattern 3:
 0 1 0
 1 1 0
 0 0 0
Matching sub-regions:
(4, 5)--(6, 7)

Total number of matching sub-regions: 6
```

Important notes

- There is no need to sort the sub-regions to be output.
- You are to submit a program that can be compiled. You will lose 25 marks out of 50 marks right away if your program cannot be compiled (5 marks for compilability; 20 marks for correctness).
- Template programs **BinaryImage.java** and **PatternDetection.java** are provided on the next few pages and they are also available in your Working Directory (refer to paragraph 4 on page 1).
- Sample inputs (**image_sample.in1** and **image_sample.in2**) and outputs (**image_sample.out1** and **image_sample.out2**) are also available in your Working Directory.

Marking scheme: Total of 50 marks

1. Can program be compiled? 5 marks
2. The use of loop to cycle through the pattern and its rotated versions; the correct order of iterating through the image to find all matching sub-regions for each pattern image: 10 marks
3. Style (including your particulars, use of constant, comments, spacing/indentation, choice of variable names): 15 marks
4. Correctness of result: 20 marks (4 test data sets; 5 marks for each data set.)
5. The marks for this task will constitute **60%** of the total marks.

```
// CS1101 (AY2006/7 Semester 1)
// PE Exercise 2
// BinaryImage.java
// Author:
// Matriculation number:
// Lecture group (X, Y or Z):
// Discussion group (1-7):

//*****************************************************************
//    WE HAVE MARKED THE TWO PLACES FOR YOU TO WRITE YOUR CODE.
//*****************************************************************

/**
 * This is the definition of the BinaryImage class.
 */
class BinaryImage {

    //  Data Members
    //----------------------------------------------------------------
    // The 2D array of bytes for storing the binary image pixels.
    // The first index is the row number and
    // the second index is the column number.
    // Row 0 is on the top and Column 0 is on the left.
    // Each pixel can have value 0 (black) or 1 (white).
    //----------------------------------------------------------------
    private byte[][] pixel;

    //----------------------------------------------------------------
    // Number of rows in the 2D image (image height).
    //----------------------------------------------------------------
    private int numRows;

    //----------------------------------------------------------------
    // Number of columns in the 2D image (image width).
    //----------------------------------------------------------------
    private int numCols;

    //----------------------------------------------------------------
    // Constructor:
    // Initializes the BinaryImage object with a copy of
    // the content of the input 2D array imageArray.
    // It is assumed that the input array is at least 1 x 1 in size.
    //----------------------------------------------------------------
    public BinaryImage( byte[][] imageArray ) {

        numRows = imageArray.length;
        numCols = imageArray[0].length;
        pixel = new byte[numRows][numCols];

        // Copy the content of the input array.
        for ( int r = 0; r < numRows; r++ )
            for ( int c = 0; c < numCols; c++ )
                pixel[r][c] = imageArray[r][c];
    }
```

```
//-------------------------------------------------------------------
//  Public Methods:
//
//      int         getNumRows ( );
//      int         getNumCols ( );
//      boolean     matches ( int, int, BinaryImage );
//      BinaryImage rotate90CW ( );
//      String      toString ( );
//-------------------------------------------------------------------

    //---------------------------------------------------------------
    // Returns the number of rows in the binary image.
    //---------------------------------------------------------------
    public int getNumRows() {
        return numRows;
    }

    //---------------------------------------------------------------
    // Returns the number of columns in the binary image.
    //---------------------------------------------------------------
    public int getNumCols() {
        return numCols;
    }

    //-----------------------------------------------------------------------
    // Returns true if and only if the specified sub-region of this image
    // matches the input pattern image. The top-leftmost pixel of the
    // sub-region is located at (row, col) of this image, and it has the
    // same size as the pattern image. If part of the input pattern image
    // falls outside this image, then returns false.
    //-----------------------------------------------------------------------
    public boolean matches( int row, int col, BinaryImage pattern ) {

        //****************************
        //    WRITE YOUR CODE HERE.
        //****************************

        return true;
    }

    //---------------------------------------------------------------
    // Returns a new BinaryImage object which is this BinaryImage
    // rotated 90 degrees clockwise.
    //---------------------------------------------------------------
    public BinaryImage rotate90CW() {
        byte[][] result = new byte[numCols][numRows];

        //****************************
        //    WRITE YOUR CODE HERE.
        //****************************

        return new BinaryImage( result );
    }
    //-----------------------------------------------------------------
    // Returns a string that shows the binary image.
    //-----------------------------------------------------------------
    public String toString() {
        String s = "";
        for ( int r = 0; r < numRows; r++ ) {
            for ( int c = 0; c < numCols; c++ )
                s += " " + pixel[r][c];
            s += "\n";
        }
        return s;
    }

//-----------------------------------
//    Private Methods
//-----------------------------------

    //******************************************
    //    ADD PRIVATE METHODS IF NECESSARY.
    //******************************************


} // BinaryImage
```

```
// CS1101 (AY2006/7 Semester 1)
// PE Exercise 2
// PatternDetection.java
// Author:
// Matriculation number:
// Lecture group (X, Y or Z):
// Discussion group (1-7):

//*************************************************************
//    WE HAVE MARKED THE PLACE FOR YOU TO WRITE YOUR CODE.
//*************************************************************

/**
 * This is the definition of the PatternDetection class.
 */

import java.util.*;  // Scanner

class PatternDetection {

    public static void main( String[] args ) {

        Scanner sc = new Scanner( System.in );

    //-------------------------------------------------------------
    // Get image height and width.
    //-------------------------------------------------------------
        System.out.print( "Enter image height: " );
        int imageRows = sc.nextInt();
        System.out.print( "Enter image width: " );
        int imageCols = sc.nextInt();

    //-------------------------------------------------------------
    // Get image pixel values.
    //-------------------------------------------------------------
        byte[][] imageArray = new byte[imageRows][imageCols];

        for ( int r = 0; r < imageRows; r++ ) {
            System.out.print( "Enter image row " + r + ": " );
            for ( int c = 0; c < imageCols; c++ )
                imageArray[r][c] = sc.nextByte();
        }

    //-------------------------------------------------------------
    // Get pattern height and width.
    //-------------------------------------------------------------
        System.out.print( "Enter pattern height: " );
        int pattRows = sc.nextInt();
        System.out.print( "Enter pattern width: " );
        int pattCols = sc.nextInt();
    //------------------------------------------------------------
    // Get pattern pixel values.
    //------------------------------------------------------------
        byte[][] pattArray = new byte[pattRows][pattCols];

        for ( int r = 0; r < pattRows; r++ ) {
            System.out.print( "Enter pattern row " + r + ": " );
            for ( int c = 0; c < pattCols; c++ )
                pattArray[r][c] = sc.nextByte();
        }

    //------------------------------------------------------------
    // Create BinaryImage object for the image.
    //------------------------------------------------------------
        BinaryImage image = new BinaryImage( imageArray );

    //-----------------------------------------------------------------
    // Create BinaryImage objects for the pattern and its rotated versions.
    //-----------------------------------------------------------------
        BinaryImage[] patt = new BinaryImage[4];
        patt[0] = new BinaryImage( pattArray );
        patt[1] = patt[0].rotate90CW();
        patt[2] = patt[1].rotate90CW();
        patt[3] = patt[2].rotate90CW();

    //------------------------------------------------------------
    // Display image.
    //------------------------------------------------------------
        System.out.println( "Image:");
        System.out.println( image );

    //------------------------------------------------------------------
    // Display each pattern and find all sub-regions that match it,
    // and output the locations of the matching sub-regions.
    //------------------------------------------------------------------
        int numMatches = 0; // Total number of matching sub-regions.


        //****************************
        //    WRITE YOUR CODE HERE.
        //****************************



    //------------------------------------------------------------------
    // Output the total number of sub-regions that match the pattern
    // and its rotated versions.
    //------------------------------------------------------------------
        System.out.println( "Total number of matching sub-regions: "
                            + numMatches );

    } // main

} // PatternDetection
```