1. [Basic Concept and OOP]

   a) Match the terms in the left column to their corresponding definitions/functionalities in the right column:

   | A. Class      | 1. Create a new object instance and initialize the instance variable |
   | B. An Object  | 2. A name given to a package, class, interface, method, or variable |
   | C. Identifier | 3. Store data in a program                                     |
   | D. Variable   | 4. User defined data type                                     |
   | E. Method     | 5. A value that is passed to the method                       |
   | F. Parameter  | 6. Function, or procedure                                    |
   | G. Constructor| 7. Instance of a class                                       |

   Answers: A – 4, B – 7, C – 2, D – 3, E – 6, F – 5, G - 1

   b) The function below is intended to find prime number. Find what is wrong with the code and correct the mistake.

   ```java
   /* Precondition: num can be any number
   Postcondition: true if num is prime
                   false otherwise */
   public static bool isPrime(int num) {
      for(int i=0; i<=num; i++) {
         if(num%i == 0) {
            return false;
         }
      }
      return true;
   }
   ```
2. [Modular Coding]

```java
public static int lcm (int a, int b) {
    // ANSWER
    // Find the larger number
    int num = a;
    if(num < b) {
        num = b;
    }

    // Find LCM:
    // - num has to be divisible by both a and b
    while(num%a != 0 || num%b != 0) {
        num++;
    }
    // if(num%i == 0)
    //     return false;
    return true;
}
```

a) Write a Java method to calculate the least common multiple (LCM) for two given positive integers

b) Given the function to calculate the LCM above or otherwise, write a Java method to calculate the greatest common divisor (GCD) for two given positive integers

c) Given the method to calculate GCD for two given positive integers, write a Java method to calculate the GCD for four given positive integers (Hint: you do not need to rewrite the method, you can use gcd)

```java
public static int gcd4(int a, int b, int c, int d) {
    // ANSWER
    // Using the relationship that gcd(a,b,c) = gcd(gcd(a,b),c)
    // - Any split would do
    return gcd(gcd(a,b),gcd(c,d));
    // return gcd(gcd(gcd(a,b),c),d);
    // return gcd(a,gcd(b,gcd(c,d)));
    // etc...
}
```
3. [Pass-by-Value]

a) Explain the behavior of the following three code fragments below, especially on why some of the codes managed to swap the value of the variable passed into the argument.

```java
// Is it swapped?
public static void swap1(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

public class MyInteger {
    public int x;
    public MyInteger(int n) {
        x = n;
    }
}

// Is it swapped?
public static void swap2(MyInteger a, MyInteger b) {
    int temp = a.x;
    a.x = b.x;
    b.x = temp;
}

// Is it swapped?
public static void swap3(int[] a, int i, int j) {
    a[i] = a[i] ^ a[j];
    a[j] = a[i] ^ a[j];
    a[i] = a[i] ^ a[j];
}
```

Answers:

Only `swap1` is not swapped, while `swap2` and `swap3` is swapped.

Java uses pass-by-value for function call. However, the value that is passed when the given argument is object is the address of the object. Hence, the value of `a.x` and `b.x` in `swap2` will be changed in the caller as well. This is because an object can be arbitrarily large. Thus, copying the entire object may not be efficient (imagine an object with size of 4 GB).

Similarly, since array can be very long, the value that is passed is the starting address of the array (however, unlike in C/C++, you cannot do pointer arithmetic). Hence, the value at `a[i]` is swapped with the value at `a[j]`. The algorithm given there uses XOR operator and it will swap the value of integer.
4. [Challenging Question: Game of Life]

a) In this question, we want to manipulate 2-dimensional array by simulating the Conway’s game of life. The game is a simple rule-based simulated that can be found here: http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

The rules will be accompanied by picture for clearer representation. In the picture below, black box is live cells and white box is dead cells.

1. Any live cell with fewer than two live neighbors dies (e.g. the cells at the top left and top right corner die because they have only one neighbors each)
2. Any live cell with two or three live neighbors lives on to the next generation (e.g. the two cells at the bottom has exactly two neighbors)
3. Any live cell with more than three live neighbors dies (e.g. the center middle cell)
4. Any dead cell with exactly three live neighbors becomes a live cell (e.g. top middle cell and center right cell)
import java.util.*;

public class Conway {
    // Data Member
    boolean[][] board;
    int length;

    public Conway(int[][] init, int l) {
        length = l;
        board = new boolean[length][length];

        // Convert from int to boolean
        for(int i=0; i<length; i++) {
            for(int j=0; j<length; j++) {
                if(init[i][j] == 1) {
                    board[i][j] = true; // ALIVE
                } else {
                    board[i][j] = false; // DEAD
                }
            }
        }

        // Compute the next step from board
        // - Save the result in board
        public void next() {
            // Your code here...
        }

        // Count the number of neighbors at index (i,j)
        // - Index starts from (0,0) to (length-1, length-1)
        public int countNeighbor(int i, int j) {
            // Your code here...
        }
    }
}

Below is a code fragments for you to complete.
To simulate the game, you can add the following code to your file as the main function.

```java
// Simulation
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int length = sc.nextInt();
    int[][] in = new int[length][length];

    // Read input
    for (int i=0; i<length; i++) {
        for (int j=0; j<length; j++) {
            in[i][j] = sc.nextInt();
        }
    }
    Conway gameOfLife = new Conway(in, length);

    // Simulate
    int steps = sc.nextInt();
    for (int i=0; i<steps; i++) {
        gameOfLife.next();
    }

    // Print
    boolean[][] out = gameOfLife.board;
    for (int i=0; i<length; i++) {
        for (int j=0; j<length; j++) {
            if (out[i][j]) {
                System.out.print("X");
            } else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }
}
```
// Compute the next step from board
// - Save the result in board
public void next() {
    // Create the next board
    boolean[][] next = new boolean[length][length];

    // For each element in the board, compute the next step
    for(int i=0; i<length; i++) {
        for(int j=0; j<length; j++) {
            // Count the neighbor (to avoid repeated computation)
            int neigh = countNeighbor(i, j);

            // Rules:
            if(neigh < 2) { // < 2
                next[i][j] = false;
            } else if(neigh == 3 && board[i][j] == false) { // 3
                // This is computed before the rule below
                // to avoid conflict since both requires neigh == 3
                next[i][j] = true;
            } else if(neigh == 2 || neigh == 3) { // 2-3
                next[i][j] = board[i][j];
            } else if(neigh > 3 && board[i][j] == true) { // >= 4
                next[i][j] = false;
            }
        }
    }

    // Assign the board to be the next board
    board = next;
}

// Count the number of neighbors at index (i,j)
// - Index starts from (0,0) to (length-1, length-1)
public int countNeighbor(int i, int j) {
    int count = 0;

    // Loop for surrounding 3*3 cells
    for(int ii=-1; ii<=1; ii++) {
        for(int jj=-1; jj<=1; jj++) {

            // Check Boundary
            if((i+ii < 0 || j+jj < 0 || i+ii >= length || j+jj >= length) {
                continue; // Outside the boundary
            }

            // Check for the same cell
            if(ii == 0 && jj == 0) {
                continue; // The same cell (not neighbor)
            }

            // Check if neighbor is alive
            if(board[i+ii][j+jj]) {
                count++;
            }
        }
    }

    return count;
}