CS2040S: Data Structures and Algorithms

Problem Set 1

Overview. You have two basic tasks this week.

- Your first job this week is to set up your environment for writing programs in CS2040S. By the end of this problem set, you should be able to write simple Java programs.
- Your second job is to implement a *linear feedback shift register*. This is a simple data structure that can be used as part of a simple encryption schemes, or as a part of a pseudorandom number generator.

For each problem, upload the Java files to Coursemology as individual files, and write any textual answers in the text box.

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so in Coursemology by adding a Comment. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. (**Getting up and running**)

In CS2040S, we will be writing programs in Java. We will be using *IntelliJ* as our basic development environment. IntelliJ is available as an open-source (Apache 2.0) Community version that has all the functionality needed for CS2040S, and runs on Windows, Macs, and Linux. It contains useful editing, debugging, and profiling tools. You can download it at:

`https://www.jetbrains.com/idea/download/`

Problem 1.a. Start IntelliJ and create a new Java project called `ps1` for this problem. Choose Java version 11.0.5 for Project SDK. If you do not have the SDK, download it here. Inside the `src` folder, create a new Java class called `Main.java` (the name is case sensitive!). Within this class, create the following method:

```
static int MysteryFunction(int argA, int argB) {
    int c = 1;
    int d = argA;
    int e = argB;
    while (e > 0) {
        if (2*(e/2) !=e) {
            c = c*d;
        }
        d = d*d;
        e = e/2;
    }
    return c;
}
```

Also, create the following main function:

```
public static void main(String args[]) {
    int output = MysteryFunction(5, 5);
    System.out.printf("The answer is: " + output + ".");
}
```

Run your program. What is the answer output in your solution?

Problem 1.b. (Optional.) What is MysteryFunction calculating? If you try a few examples, you might be able to guess.

Problem 1.c. Create a "Hello CS2040S!" program that is designed to introduce yourself to your tutor and name it `HelloWorld.java`. It can be as simple or complicated as you like. It should output (in some form):

- your name (as you prefer to be called),

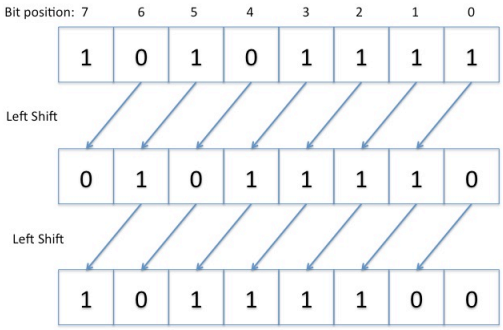
- your favorite algorithm or your favorite joke,
- a few sentences of additional information on your background and who you are (but nothing private that you would want kept secret from the rest of the class),
- the answer(s) to the previous parts.

This `HelloWorld.java` file is the only part of this problem to submit.

Problem 2. Linear Feedback Shift Register

A *shift register* is an array of bits that supports a *shift* operation which moves every bit one slot to the left. For example, below is an example of a shift register initially containing the seed "10101111". It is shifted twice. Each time it is shifted, the most significant bit is dropped, every other bit is moved one place to the left, and the least significant bit is replaced with a '0'. Notice that after it is shifted 8 times, the shift register will be all zeros.

Note : In the *usual* convention, the most significant bit is written as the leftmost bit.

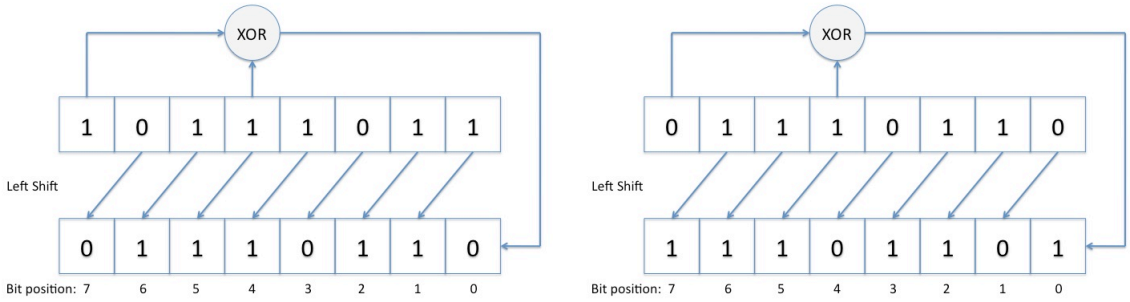


A *linear feedback shift register* is a special type of shift register that, instead of setting the least significant bit to zero, updates the least significant bit with some linear function of the other bits. That is, when a shift operation occurs, it feeds back some information into the least significant bit. We will build a linear feedback shift register based on the *exclusive-or* (XOR) function. (In Java, `(a^b)` calculates the XOR of `a` and `b`.)

Our linear feedback shift register takes two parameters: a *size* and a *tap*. The size refers to the number of bits. The tap refers to one of the bits and must be between 0 and *size*−1 (inclusive). *size* specifies number of bits the register should be dealing with and *tap* specifies which bit (starting from the least significant bit) is going to used in the XOR operation.

Every time a shift operation occurs, the following four things happen: (1) The feedback bit is calculated as the XOR of the most significant bit and the tap bit. (2) The most significant bit is dropped. (3) Every bit is moved one slot to the left. (4) The least significant bit is set to the feedback bit.

Here are two examples of a linear feedback shift register in action. In both cases, the size is 8 and the tap is 4 (which is the 5th least significant bit. e.g. If the tap was 0, then we would XOR with least significant bit).



Download `code.zip`. Inside, you should find the relevant files that you would require to complete the following tasks. Open the folder using IntelliJ and configure Project SDK to be Java version 11.0.5.

Problem 2.a. Implement a linear feedback shift register called `ShiftRegister` that implements the ILFShiftRegister interface:

```
public interface ILFShiftRegister {

    public void setSeed(int[] seed);

    public int shift();

    public int generate(int k);
}
```

The interface requires that the following methods be supported:

- `void setSeed(int[] seed)`: sets the shift register to the specified initial seed. The initial seed is specified as an array of 0's and 1's (represented as integers). If the seed contains any other value, that is an error. (Recall that the *seed* is the initial set of bits stored in the shift register.)

  IMPORTANT NOTE : The given input array *seed* will be such that the least significant bit is on index 0 of the integer array. For example, if the actual seed is 00101, then it will be given as $\{1, 0, 1, 0, 0\}$ in the program.

- `int shift()`: executes one shift step and returns the least significant bit of the resulting register.

- `int generate(int k)`: extracts k bits from the shift register. It executes the `shift` operation k times, saving the k bits returned. It then converts these k bits from binary into an integer. For example: if `generate` is called with a value of 3, then it calls `shift` 3 times. If the `shift` operations return 1 and then 1 and then 0, then `generate` returns the value 6 (i.e., "110" in binary).

Your implementation should also support the following constructor:

```
ShiftRegister(int size, int tap)
```

This constructor initializes the shift register with its size and with the appropriate tap. Submit your implementation.

Problem 2.b. Test your code with a variety of test cases. Make sure to test the corner cases (e.g., when the shift-register has size 1, etc.). Here are two sample test cases to use. The first tests the `shift` functionality, while the second tests the `generate` functionality.
Test 1: seed = '101111010', tap = 7

```
int[] array = new int[] {0,1,0,1,1,1,1,0,1};

ShiftRegister shifter = new ShiftRegister(9,7);
shifter.setSeed(array);

for (int i=0; i<10; i++){
    int j = shifter.shift();
    System.out.print(j);
}
```

This code should produce the following output:

```
1100011110
```

Test 2: seed = '101111010', tap = 7

```
int[] array = new int[] {0,1,0,1,1,1,1,0,1};

ShiftRegister shifter = new ShiftRegister(9,7);
shifter.setSeed(array);

for (int i=0; i<10; i++){
    int j = shifter.generate(3);
    System.out.println(j);
}
```

This code should produce the following output:

```
6
1
7
2
2
1
6
6
2
3
```

Now test your code using the provided "JUnit test." (These are a subset of the tests that the tutors will use when evaluating your code.) Load the file `ShiftRegisterTest.java` in IntelliJ, and update the method `getRegister` to use your shift register implementation. Then run the tests by clicking the button beside line number 10 or using keyboard shortcuts (Ctrl+Shift+R for Mac and Ctrl+Shift+F10 for Windows). Other useful keyboard shortcuts can be found in this reference sheet.

Look at the last test: it is testing what happens when the seed is larger than the specified register. Explain what you think a proper response is to this erroneous situation, and what is the right way to test this case? (We will cover error handling later in the semester.) **Submit your answer as in a comment in** `ShiftRegisterTest.java`

Write some additional JUnit test cases for your code in `ShiftRegisterTest.java` (you may reference the existing test cases in the test class). We expect you to sufficiently test your code. You will be expected to test for the two previously mentioned scenarios, along with some other scenario, and at least one more corner case.

Problem 2.c. Think about how you might use a shift register to perform a simple encryption scheme. (Hint: initially, the seed of the shift register should be set to your "password.") How would you encrypt a text file? How would you encrypt an image?

The two released classes `ImageEncode.java` and `SimpleImage.java` implement a simple encryption/decryption scheme using a shift register. You will need to add a couple lines of code to `ImageEncode.java` so that it properly uses your shift register. To decode the picture, use the following "code":

- Tap: `7`
- Seed: `0111011001011`

If your shift register works correctly, you should then be able to decode the included mystery picture. Who is it a picture of? (The only thing you are required to submit for this part is an identification of the decoded picture.)

Problem 2.d. (Extra, for those who are interested) Think of the output of a linear feedback shift register as the sequence of 0's and 1's that you get if you call `getBit()` after each shift. A

linear feedback shift register is only useful for encryption if the resulting output sequence does not repeat very often. Experiment with how many times a shift register has to be shifted in order to cause the output pattern to repeat. Are all taps equally good?

Also, notice that a short binary password will not provide very good security. Optionally write a modified version that takes a text string (such as, "TheCowJumpedOverTheMoon") and converts it to a (long) binary string to use as a seed for a shift register.

Another use of linear feedback shift registers is to generate pseudorandom numbers. One basic property of a pseudorandom number generator is that it outputs roughly the same number of 0's as 1's. Is that true of your linear feedback shift register?

This is an open ended question, feel free to submit anything that you think is appropriate with the relevant explanations.

Hints for question 2:

Do I need to comment my code? Yes. Be sure to write a comment explaining: the purpose of each class instance variable, and the purpose/workings of each method. You should also explain any critical steps in your code.

What data structure should I use to store the current value of the shift register? One of the advantages of encapsulation is that you can store the value of the register in any way that you want. For simplicity, we recommend that you use an array of integers (as is passed to `setSeed`). However, if you want to optimize the running time of the `shift` operation, there are better more efficient solutions. Implement the best solution you can!

For `generate`, how do I convert the bits produced by `shift` into an integer? Begin with a variable v set to zero. Every time you get a new bit from `shift`, multiply v by two, and add the new bit (either 0 or 1). The final result will be the integer representation of the bits.

How do I create a variable to store the array of integers in the register when I don't know its size? Declare it simple as an array of integers: `int[] intArray`. Then, initialize it in the constructor once you have learned the correct size.

For Test 2, my first call to `generate` produces the value 3 instead of the value 6. You are interpreting the bits in the wrong order. Notice that the binary string '110' is 6, while the binary string '011' is 3.

I get an ArrayOutOfBounds error (or a NullPointerException). Did you remember to initialize all your class instance variables?

When I try to run `ShiftRegisterTest`, it does not execute properly. Perhaps it asks if you want to run an Ant build? Perhaps it asks something else strange. Make sure that you have the JUnit package properly installed by adding a new JUnit test to your project. (You can do this under File→New→JUnit test.) This ensures that IntelliJ is properly set up to run tests. Alternatively, you can check that the libraries are there. Go to Project→Properties, then look in the section Java Build Path→Libraries. You should see two entries: JRE and JUnit. If JUnit is not there, click add

library and choose JUnit.

I am getting an error that says that each interface/class must be defined in its own file. Yes. In Java, each interface or class has to be in its own file, and the name of the file should match the name of the class (with the addition of the .java extension.)

Where should I put the mystery file? Put it in the folder that contains the src folder, i.e., the project's root folder.

I placed the mystery file in the src folder but the output is a blank window. What do I do? In the `main` function of `Main.java`, use the full path to the mystery file instead, i.e. `/path/to/mystery.bmp` instead of `mystery.bmp`

How should I debug my program, when I cannot see the internal state of my object? Wouldn't it be nice if you could just type `System.out.println(shifter)` and have it print out the value of the shift register? In fact, you can if you implement:
`public String toString()`
This method should convert the value of the register into a string and return it. Then, you can modify your test code to include:

```
System.out.println(shifter + " " + j);
```

and you will get the following output for the first test:

```
011110101 1
111101011 1
111010110 0
110101100 0
101011000 0
010110001 1
101100011 1
011000111 1
110001111 1
100011110 0
```