ALGORITHMS & DATA STRUCTURES

ADDITIONAL INFORMATION FOR LAB 16
WEEK OF 25TH FEBRUARY 2019

1. This week you have to implement the two random number generators we discussed in lecture 15:
   - **a.** Linear Congruential Generator (LCG): $X_{i+1} = (a * X_i + c) \bmod m$
   - **b.** Xorshift generator ($y=y<<a; y=y>>b; y=y<<c$).

2. For every generator you must implement the following two methods:
   - **a.** seed(): this function sets the value of the seed of the generator (X_0 for the LCG; the very first value of *y* for the Xorshift generator).
   - **b.** next(): this function generates and returns the next random number. This number must be an **unsigned 32-bit word**. In C++ this is simple to achieve (in fact, the definition of the return data type uint_32_t is already given to you). In Java, you must take extra care (see section for Java coders at the end of this document), as Java does not have unsigned data types.

3. Regarding the LCG, do you remember the exercise we did in class regarding the pattern shown by low-order bits? It is reproduced below:

For a=29, c=35, m=256 (set of values meeting the conditions for full period) and X_0=1, the following table shows the first 20 random numbers and their binary representations (leftmost columns). In the 3 rightmost columns you can see the "cyclic behaviour" of the low-order bits: the last bit alternates between 0 and 1; the two last bits follow the same cycle over and over (0,3,2,1) and the same happens with the three last bits (cycle 0,3,2,5,4,7,6,1).

i	Xi (decimal)	Xi (binary)	Decimal b_0	Decimal b_1b_0	Decimal $b_2\,b_1b_0$
1	64	01000000	0	0 (00)	0 (000)
2	99	01100011	1	3 (11)	3 (011)
3	90	01011010	0	2 (10)	2 (010)
4	85	01010101	1	1 (01)	5 (101)
5	196	11000100	0	0 (00)	4 (100)
6	87	01010111	1	3 (11)	7 (111)
7	254	11111110	0	2 (10)	6 (110)
8	233	11101001	1	1 (01)	1 (001)
9	136	10001000	0	0 (00)	0 (000)
10	139	10001011	1	3 (11)	3 (011)
11	226	11100010	0	2 (10)	2 (010)
12	189	10111101	1	1 (01)	5 (101)
13	140	10001100	0	0 (00)	4 (100)
14	255	11111111	1	3 (11)	7 (111)
15	6	00000110	0	2 (10)	6 (110)

16	209	11010001	1	1 (01)	1 (001)
17	208	11010000	0	0 (00)	0 (000)
18	179	10110011	1	3 (11)	3 (011)
19	106	01101010	0	2 (10)	2 (010)
20	37	00100101	1	1 (01)	5 (101)

This exercise showed that the low-order bits of a sequence of random numbers generated by a LCG might exhibit a clear pattern. For example, the least significant bit alternates between 0 and 1, meaning that you always get an odd number followed by an even number in your sequence of random numbers. This is not desirable. To solve this problem, some generators produce random numbers with a much higher number of bits than required and then, they drop the low order bits by performing a shift right (>>). For example, you can generate a 64-bit random number and then drop the least significant 32 bits (applying >>32).

In this lab, you are asked to implement a LCG with the functionality of dropping the low-order bits. The signature for the new constructor is as follows:

C++ LCG::LCG(uint64_t _a, uint64_t _c, uint64_t _m, uint64_t seed, **uint64_t _shift**)
Java LCG(long _a, long _c, long _m, long seed, **long _shift**)

The last input argument (shift) corresponds to the number of low-order bits the generator has to drop (applying >>) before returning the random number. For example, if you are going to drop the last bit of your generated random number, then **before returning it** you do x=x>>1 (in this case, shift is equal to 1).

A FEW IMPORTANT THINGS FOR JAVA CODERS

In this lab, you are supposed to generate non-negative integer random numbers. In C++ there is data type called unsigned integer (i.e. non-negative integer) that uses 32 bits to represent non-negative numbers and another type called unsigned long that uses 64 bits. Thus, in C++ is not necessary to take any additional measures to make the generators work as requested. In Java, however, there are no such data types.

If you are a Java coder, you will have to work with the **long** data type that uses 64 bits to represent signed numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (i.e. between the negative number -2^{63} and the positive number $2^{63}-1$, due to the two's complement representation we will discuss/discussed in lecture 16, depending on when you read this). Thus, you will have to take extra care to make sure you do not get unexpected results, as negative numbers. Always remember that Java uses two's complement and 64 bits for the **long** data type.

There are two situations where problems might arise:

Situation 1: LCG

Situation 1 happens when the operation ($a* X_i+c$) in the LCG results in a number higher than the maximum positive integer representable by the **long** data type, that is 2^{63}-1. For example if you sum 1 to the maximum positive number 9,223,372,036,854,775,807 (represented as the 64-bit word 011111….111), the result is 10000…000 (64 bits). As Java uses two's complement, that is the negative number -9,223,372,036,854,775,808. If you then apply the module to that negative number and you still get a negative number between -1 and –(m-1), which is not desirable.

I will illustrate this point with a much smaller example, using a 3-bit word. Using two's complement, with 3 bits you can represent the integers in the range -4 to 3:

Decimal	Two's complement
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Let's assume you are using the set of parameters a=1, c=2 and m=4 and your seed is 1. Then, your first random number will be: (1*1+2) % 4 = 3 % 4= 3.
Using the just generated random number 3 as the input for the next number, we calculate (3*1+2) % 4. Now we have a problem: (3*1+2) is equal to 5, which is out of the range of the positive numbers representable with 3 bits using two's complement. In fact, the result of (3*1+2) will be 101 that corresponds to the number -3. Thus, the second random number will -3 % 4, which is -3, a negative number!

There are several ways of correcting this. **In this lab,** if you get a negative number with your Java-implemented LCG, before returning the random number, **just sum the value of *m* to it**. As the negative number will be something between -1 and –(m-1), by summing *m* units to it you are sure you get a positive number.

Situation 2: Xorshift

When you perform a shift left (as you must do twice when implementing the xorshift generator), you are actually multiplying your original number by 2^x, where x is the number of positions you are shifting the bits.

Let's see a few examples with 64-bit words. The binary representation of the decimal number 5 is 00… 00000101 (64 bits). The following table shows the different numbers you get when you perform different shift right operations to the original number 5:

Shift operation	**Resulting binary number**	**Resulting decimal number (assuming unsigned integers)**
5 << 1	00…00001010	10 (shifting 1 position to the right is equivalent to multiplying by 2)
5<<2	00…00010100	20 (shifting 2 positions to the right is equivalent to multiply by 2^2)
5<<3	00…00101000	40 (shifting 3 positions to the right is equivalent to multiply by 2^3)
5<<5	00…10100000	160 (shifting 3 positions to the right is equivalent to multiply by 2^5).

Thus, in general, when you perform a shift right in x positions, you are actually multiplying the original number by 2^x.

In this lab, you are asked to work with unsigned 32-bit words. As in Java you are working with signed 64-bit words, you have to make sure you just generate numbers that use the first (i.e. leftmost) 32 bits. That is, every time you perform a shift, the last (i.e. rightmost) 32 bits must be equal to zero. A simple way of doing this is applying module 2^{32} (4,294,967,296) every time you perform a shift (check on the Xorshift3520Test.java to see the code details).