Problem Set Three

En, Lin

1. A self-shrinking LFSR of length 12 employs a primitive tap polynomial. The output begins 00000000000
2. What’s the probability that the next output bit is equal to a 1? (So, here, the key is not just the fill but also the tap locations.)

* Since the length of LFSR is only 12, we can iterate through all possible polynomial taps, and mark those polynomial taps that generate 2^12 – 1 different states. This process takes roughly 2^24 steps (shifting doesn’t have huge impact since there is only 12 bits). I found that there are 144 such primitive tap polynomials
* Now we try 2^12 possible initial non-zero fills for those 144 primitive polynomials. We record total number of (primitive polynomials tap, initial fill) pairs that starts with 00000000000, and the number of such pair that begins with 000000000001. The result was 232 and 143. Since those pairs are equally possible, the probability that next output bit is 1 is

1. If the first k bits of the output are 0’s, what is the largest value of k such that there is a self-shrinking LFSR that generates the sequence from a non-zero fill?

* We can solve this problem by brute force. We iterate through all possible (primitive polynomials, initial fill) setup for LFSR. We run the LFSR with each setup, and then record the longest consecutive zeros. The answer was 15.

(The output from a self-shrinking linear feedback shift register is equal to the second highest position bit if the high bit is a 1. If the high bit is a zero, there is no output. The register is then stepped twice, and this process repeated. For this question, you should assume that the output begins from an initial fill and that all possible initial fills with associated primitive tap polynomials that produce the given output are equally likely – and that the all zero fill is never considered)

1. NIST Special Publication 800-22revla (April 2010), *A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications* identifies fifteen statistical tests for judging the “quality of a random number generator.” Design your own LFSR-based random bit generator that passes at least eleven of these tests based upon on million bits of outputs. And make your generator as efficient and unpredictable as you can.

* The design is based on 4 LFSR with primitive tap polynomials. They are x^16 + x^7 + x^5 + x^4 + x^3 + x^2 + 1, x^20 + x^6 + x^5 + x^4 + x^3 + x + 1, x^21 + x^6 + x^5 + x^3 + x^2 + x + 1, x^23 + x^5 + 1. I chose the length of LFSRs to be around 20 so the maximal number of states they can have are comparable to 1 million (number of bits to be tested on). I got those primitive tap polynomials by trying all possible tap positions, and then simulated the LFSR to see whether it had maximal span. The output of the bit stream is the result XOR first bit of four LFSRs.
* **Clocking rule:**  each LFSRmaintains a variable “ones” (how many ones in the current fill). Maintaining this variable is efficient since it only requires two operations per LFSR stepping. Then based on how many ones we have on each LFSR fill we choose which LFSR to step. Following code snippet describes the detail.

*public void next() {*

*int scramble = L[0].ones;*

*scramble = scramble \* 13 + L[1].ones;*

*scramble = scramble \* 13 + L[2].ones;*

*scramble = scramble \* 13 + L[3].ones;*

*L[**( scramble + 1 ) % L.length ].step();*

*If( scramble % 2 == 0 ) {*

*L[**( scramble \* scramble ) % L.length ].step();*

}

*}*

A prime number 13 is chosen to make sure gcd(13, number of LFSRs) = 1. For efficiency we don’t step more than two LFSRs in one round. It could happen that only one LFSR steps, or the same LFSR steps twice.

* **Result:** My Pseudo random number generator produced 15 million bits in 2.5 seconds. I tested the output on NIST statistic test suite with 15-bit stream, each containing 1 million bits. The result of testing is appended at the end of this report.

3a). Suppose g is a primitive root (mod p). What is the value of the Legendre symbol (g/p)? Provide a proof.

* (g/p) = g(p-1)/2
* Since g is primitive root (mod p), g(p-1)/2 = -1
* the answer is -1

3b). Suppose g is a primitive root (mod p) and we want to solve the DLP g^x = a (mod p). Consider the following approach to determine (k bits of)x: First, find(a/p) and, so, the parity of x. If x is even rewrite x as 2x and take a square root of both sides and go back to the previous step. If x is odd, multiply both sides of the DLP by g and rewrite x+1 as 2x and take a square root of both sides and go back to the first step. Will this work? Can it help find x at all?

* This recursive method indeed works. I wrote the program to solve 2^x = 48(mod 107) and got a correct answer 74. The code will be appended at the end of this report.
* My program assumes that p 3 (mod 4). However, we know there exits an efficient randomized algorithm for square root even when p 1 (mod 4).
* One problem with this recursive method is that it is not efficient. The time it takes to solve DLP could be as bad as brute force method. Each step we get the corresponding square root +r, -r, the program has to search both path (one for +r, one for -r). Imagine the program as a tree, the depth of the tree could be log2p. In worst case, the total running time of the program could be O(p).

A close up of a piece of paper

Description automatically generated

A close up of a window

Description automatically generated

A close up of a logo

Description automatically generated