CMDA 3634 Spring 2018 Homework 02

Zorian Thornton

March 1, 2018

You must complete the following task by 5pm on Tuesday 02/27/18.

Your write up for this homework should be presented in a LaTeX formatted PDF document. You may copy the LaTeX used to prepare this report as follows

1. Click on this link
2. Click on Menu/Copy Project.
3. Modify the HW01.tex document to respond to the following questions.
4. Remember: click the Recompile button to rebuild the document when you have made edits.
5. Remember: Change the author.

Each student must individually upload the following files to the CMDA 3634 Canvas page at https://canvas.vt.edu

1. `firstnameLastnameHW02.tex` LaTeX file.
2. Any figure files to be included by `firstnameLastnameHW02.tex` file.
3. `firstnameLastnameHW02.pdf` PDF file.

In addition, all source code must be submitted to an online git repository as follows:

1. While on the webpage for you git repository, go to Settings → Collaborators.
2. Add nchalmer@vt.edu and zjiaqi@vt.edu as collaborators.
3. Make a folder named HW02 in you repository and store all relevant source files to this assignment in this folder. Ensure your assignment files compile with `make`.

You must complete this assignment on your own.

100 points will be awarded for a successful completion.
Extra credit will be awarded as appropriate.

ElGamal Public-key Cyptography

In the previous assignment we began implementing the ElGamal public-key cryptographic system by first programming several 'first-attempts' of some functions we'll need. In the bonus question, many students identified some serious implementation/performance problems with the codes you created. The most serious ones were:

- Products $ab \bmod p$ and exponentials $a^b \bmod p$ may overflow integer storage for large numbers. Indeed, for some students who used the `pow` library function to compute $a^b \bmod p$ this happens dramatically fast.
- Checking for primality of an integer N by checking if N is divisible by any number smaller than $\sqrt{N}$ becomes very expensive for large numbers N, potentially requiring millions of divisions even for modest size integers.
- The findGenerator function will quickly become very computationally expensive as the input prime p grows since in its most basic implementation you are looping through all powers of a test generator.

In this assignment we will resolve these issues as well as add some more useful functions.

Safe products of integers modulo p: Whenever we program a product of two numbers a and b we need to be aware of storage limits for the resulting value. In the case of 32-bit unsigned integers, this means the product ab must be less that or equal to $2^{32} - 1 = 4,294,967,294$ in order for its value to be properly stored in memory. However, given a modulus $p < 2^{32} - 1$ we know that the product $ab \bmod p$ will be always be less than $2^{32} - 1$. Knowing this, we can properly compute its value if we implement it carefully.

To begin, let's first notice that the $\bmod p$ operation is distributive. That is, it is always true that

$$a(b + c) \bmod p = (ab \bmod p + ac \bmod p) \bmod p,$$

for any a, b, c, and p. Moreover, for a product of three integers a, b, and c is always true that

$$abc \bmod p = a(bc \bmod p) \bmod p,$$

Next, let us recall that we can write any positive integer in binary. Let's write the binary representations of an integer b as

$$b = b_n|b_{n-1}| \cdots |b_1|b_0,$$

where each digit b_i is either 1 or 0 such that

$$b = b_n 2^n + b_{n-1} 2^{n-1} + \ldots b_1 2^1 + b_0 2^0.$$

Replacing b with its binary representation we can write the product $ab \bmod p$ as

$$\begin{aligned} ab \bmod p = (&ab_n 2^n \bmod p \\ &+ ab_{n-1} 2^{n-1} \bmod p \\ &+ \ldots \\ &+ ab_1 2^1 \bmod p \\ &+ ab_0 2^0 \bmod p) \bmod p. \end{aligned}$$

This can be written even more compactly by introducing the values $z_i = a2^i \bmod p$ and noticing that $z_i = 2z_{i-1} \bmod p$. If we separate the evaluation of $ab \bmod p$ into the combination of all these spearate products, we can write a safe version of the modular product $ab \bmod p$ as the following pseudo-code

```
modProd(a,b,p)
    za = a
    ab = 0
    for i=0,...,n {
        if (b_i == 1) ab = (ab + za*b_i) mod p
        za = 2*za mod p
    }
    return ab
```

With this algorithm, we can safely compute any product ab mod p so long as the products `2*za` can be computed in integer storage. For unsigned 32-bit integers this usually means this product is accurate when a, b and p are all less than $2^{31} - 1 = 2,147,483,647$.

We can use this safe modular product algorithm to compute modular exponentials a^b mod p by again writing b using its binary representation to obtain

$$
\begin{aligned}
a^b \bmod p = &(a^{b_n 2^n} \bmod p)\\
&\cdot (a^{b_{n-1} 2^{n-1}} \bmod p)\\
&\cdot \ldots\\
&\cdot (a^{b_1 2^1} \bmod p)\\
&\cdot (a^{b_0 2^0} \bmod p) \bmod p.
\end{aligned}
$$

We then can introduce the values $z_i = a^{2^i}$ and note that $z_i = z_{i-1}^2$ mod p. The pseudo-code for the safe version of modular exponentiation a^b mod p can therefore be written as

```
modExp(a,b,p)
    z = a
    aExpb = 1
    for i=0,...,n {
        if (b_i==1) aExpb = modProd(aExpb,z, p)
        z = modProd(z,z,p)
    }
    return aExpb
```

Q1.1(5 points) Use the procedure described above to write out the details of how to safely compute the modular product 56*74 mod 111.

To begin the process of computing 56*74 mod111 safely, we must first write b, or in this case, 74 in binary:
74 = 0100 1010 in basic 8-bit integer format.
To calculate 56*74 mod 111 in a safe way, we must use the equations above:

$$56(0)2^7 mod 111+$$
$$56(1)2^6 mod 111+$$
$$56(0)2^5 mod 111+$$
$$56(0)2^4 mod 111+$$
$$56(1)2^3 mod 111+$$
$$56(0)2^2 mod 111+$$
$$56(1)2^1 mod 111+$$
$$56(0)2^0 mod 111$$

Which yields 37.
We then mod 37 with 111 to obtain our final answer:
37 mod 111 = 37.

To confirm, we can obtain 4144 by the operation of 56*74. Which yields 37. Hence this method stated in the notes earlier works.

Q1.2(20 points) You have been given some sample code in the course repository folder HW02. In the file `functions.c` implement the modular product function `modProd` which safely computes the value ab mod p using the method described above. All subsequent modular products throughout your codes should call this function for safety.

Q1.3(20 points) In the file `functions.c` implement a modular exponentiation function `modExp` which safely computes the value a^b mod p using the method described above. All subsequent modular exponentials throughout your code should call this function for safety.

The Miller Rabin Primality Test: As mentioned above, directly testing for primality of an input integer N in your `isPrime` function will quickly become too expensive for large N. An alternative to directly checking if a input number N is prime is to use a *probablistic* primality test. Such tests will return 'true' if the input is *probably* a prime number. By repeatedly performing probabilistic primality tests, we can become quite certain that a number is indeed prime.

A very simple probabilistic primality test to state is the Miller-Rabin primality test. While the rigorous explanation of how this test works is beyond the scope of this assignment, we can write the pseudo-code for this primality test compactly as follows

```
isProbablyPrime(N):
    Make an array, smallPrimeList, of small prime numbers.

    Find r and d such that N-1 = (2^r)*d where d is odd.

    //Miller-Rabin test
    For all k in smallPrimeList:
       x = modExp(a,d,N)
       if x == 1 or x == N-1:
          continue to next k
       for i = 1,...,r-1 {
          x = modProd(x,x,N)
          if x == 1 then
             return false
          if x == N-1 then
             continue to next k
       }
       return false
    return true
```

Q2.1(15 points) In the file `functions.c` we have begun implementing the `isProbablyPrime` function. Use the pseudo-code above to complete the function.

Q2.2(10 points) In the `main` function we have begun our program by inputting a bit-length n from the user. We have also provided a function `randomXbitInt` which inputs a number of bits n and returns a random integer that is at least 2^{n-1} and no greater than 2^n. Use your `isProbablyPrime` function as well as the `randomXbitInt` to generate a prime number p that is at least 2^{n-1} and no greater than 2^n.

Finding a Generator: When we searched for a generator in the last assignment we selected a trial number g and looped through all powers of g, i.e. $g, g^2, g^3, \ldots$ and checked whether we could find an exponent $r \neq p-1$ such that $g^r = 1$. If so, we knew that g could not be a generator and we continued to the next trial number.

Many of you noticed that if we could find an $r \neq p-1$ such that $g^r = 1$ then the sequence of powers of g will repeat, i.e. $g^{r+1} = g, g^{r+2} = g^2$, etc. Since it is also true that $g^{p-1} = 1$ for all g, it must be true that r divides $p-1$. Therefore to check whether a number g is a generator we need only check that $g^r \neq 1$ for all factors r of $p-1$.

This observation doesn't completely solve our problem, as the issue of finding all the factors of $p-1$ is again very computationally expensive for large p. However, notice that if we choose the prime p so that $p = 2q+1$ where q is also prime then we can be sure the only factors of $p-1$ are 2 and q.

Q3.1(5 points) In Q1 of HW01 you showed that 2, 6, 7, and 11 are all generators of $\mathbb{Z}_{13}$. Show for the remaining numbers $a \in \mathbb{Z}_{13}$, i.e. a =3, 4, 5, 8, 9, 10, and 12, there exist a number r such that $a^r = 1$ in $\mathbb{Z}_{13}$

and confirm that r divides $p - 1 = 12$.

Let our prime number p be 13, and $p - 1 = 12$.

a	r	$a^r mod 13$	$12 mod r$
3	3	1	0
4	6	1	0
5	4	1	0
8	4	1	0
9	3	1	0
10	6	1	0
12	2	1	0

The $a^r mod 13$ column shows that there is an r such that $a^r mod 13 = 1$ and $r \neq p-1$, and the column $12 mod r$ verifies that r divides $p - 1 = 12$.

Q3.2(10 points) Modify your `main` function to generate a prime number p that is at least 2^{n-1} and no greater than 2^n and also satisfies $p = 2q + 1$ where q is prime.

Q3.3(15 points) Implement a new `findGenerator` function which assumes the input prime p satisfies $p = 2q + 1$ where q is prime, and outputs a generator of $\mathbb{Z}_p$.

Bonus:(15 points) Now that we can generate a useful prime p and a generator g continue ELGamal cryptographic system setup in the `main` function by picking a random $x \in \mathbb{Z}_p$ and computing $h = g^x$.

Once you've completed the setup, try experimenting with how difficult it would be to find the secret key x provided you only know h and g. That is, code a search for the x which satisfies $h = g^x$ by looping through all possible x. Can your computer do this quickly for large prime numbers?