CYBERSECURITY PEDAGOGY & PRACTICE JOURNAL

Volume 3, No. 2
October 2024
ISSN: 2832-1006

In this issue:

iscap
Information Systems & Computing Academic Professionals

The **Cybersecurity Pedagogy and Practice Journal** (**CPPJ**) is a double-blind peer-reviewed academic journal published by **ISCAP** (Information Systems and Computing Academic Professionals). Publishing frequency is two times per year. The first year of publication was 2022.

CPPJ is published online (https://cppj.info). Our sister publication, the proceedings of the ISCAP Conference (https://proc.iscap.info) features all papers, panels, workshops, and presentations from the conference.

The journal acceptance review process involves a minimum of three double-blind peer reviews, where both the reviewer is not aware of the identities of the authors and the authors are not aware of the identities of the reviewers. The initial reviews happen before the ISCAP conference. At that point, papers are divided into award papers (top 15%), and other accepted proceedings papers. The other accepted proceedings papers are subjected to a second round of blind peer review to establish whether they will be accepted to the journal or not. Those papers that are deemed of sufficient quality are accepted for publication in the CPPJ journal.

While the primary path to journal publication is through the ISCAP conference, CPPJ does accept direct submissions at https://iscap.us/papers. Direct submissions are subjected to a double-blind peer review process, where reviewers do not know the names and affiliations of paper authors, and paper authors do not know the names and affiliations of reviewers. All submissions (articles, teaching tips, and teaching cases & notes) to the journal will be refereed by a rigorous evaluation process involving at least three blind reviews by qualified academic, industrial, or governmental computing professionals. Submissions will be judged not only on the suitability of the content but also on the readability and clarity of the prose.

Currently, the acceptance rate for the journal is under 35%.

Questions should be addressed to the editor at editorcppj@iscap.us or the publisher at publisher@iscap.us. Special thanks to members of ISCAP who perform the editorial and review processes for CPPJ.

CYBERSECURITY PEDAGOGY AND PRACTICE JOURNAL

Editors

2024 Review Board

Teaching Public Key Cryptography: A Software Approach

David Carlson
david.carlson@stvincent.edu
Computing & Information Systems Department
Saint Vincent College
Latrobe, PA 15650 USA

Abstract

Whether you are just starting to teach cryptography, or you teach it as a stand-alone course for computer science majors or as part of a complete major in cybersecurity, the question of how to provide hands-on experience is an important one. Some software may be too expensive, while other schemes only allow students to use small, toy examples. Here, a solution using a software package called bigint is examined. It can allow students to implement, try out, and try to break C++ implementations of most common public key cryptographic algorithms. Better yet, bigint is free and will run under Linux, which is often free as well. Thus, with this free software, students can implement common cryptographic algorithms, using large numbers instead of tiny ones, time how long the computations take, and investigate where the algorithms fail to work well – the sort of exercises that help students more fully understand this technical and rapidly-changing field.

Keywords: Cryptography, Teaching, Cybersecurity Education, Public Key, Software.

Recommended Citation: Carlson, D.E., (2024). Teaching Public Key Cryptography: A Software Approach *Cybersecurity Pedagogy and Practice Journal,* v3 (n2) pp 12-20. https://doi.org/10.62273/JJIP7451

Teaching Public Key Cryptography: A Software Approach

David Carlson

1. INTRODUCTION

The purpose of this article is not to teach the reader public key encryption and decryption algorithms. This is done in many good textbooks (Elbirt, 2009; Katz & Lindell, 2020; Stallings, 2020; Stinson & Paterson, 2018; Trappe & Washington, 2006). Rather, the purpose is to show that the bigint software (McCutchen, 2010) can be used to produce programs that carry out these cryptographic algorithms well. Note that Mathematica would be a good alternative, but bigint is free.

Trying these algorithms allows students to deepen their understanding beyond what is possible by simply reading about them in a textbook. Also, because the software is free and runs on common Linux systems, this part of a cryptography course can be done cheaply and relatively easily. Understanding the algorithms is a different matter, not addressed in this paper, and requires some background in mathematics, especially abstract algebra and number theory. Many of the available textbooks present the needed background in these areas.

Although bigint is useful for trying out cryptography, it does not do everything a good course should do. For example, it would be useful to show students where a browser displays the digital certificate used on a website. It might also be good to show where digital certificates are installed on a web server, if such a server is accessible to the class.

Cryptography is heavily used. For example, every time someone uses https to connect to a website, cryptography is used to encrypt the data passed between the user and the website. Essentially, cryptography keeps private data safe from prying eyes. Even if the encrypted data is captured and examined, without the cryptographic key, the data cannot be decrypted.

There are two main types of cryptography: public key cryptography and private key cryptography. Both allow data to be encrypted and later decrypted, perhaps after the encrypted data has been sent to some recipient. Public key cryptography, the type being discussed here, uses a public encryption key and a private decryption key. The fact that only someone having the private decryption key can decrypt the message is why this method is so useful.

Unfortunately, public key cryptography is slower than private key cryptography, where both sender and receiver use the same private key to encrypt and decrypt. Generally, public key cryptography is used to securely distribute a shared key for private key cryptography. Once this shared key has been shared with the recipient, messages encoded with the shared key can be safely sent to the recipient and decrypted. Other parties cannot decrypt the messages since they don't have the shared key.

This article is about helping students to learn public key cryptography by trying it out using the bigint software package. This can be done in two distinct ways: Students with good programming backgrounds can write some of the code for doing this type of cryptography. Students who need to know some of the characteristics of public key cryptography but who do not have sufficient programming backgrounds can run prewritten programs that demonstrate some of the features of this type of cryptography.

Bigint is a free package that allows programs to be written that work with arbitrarily large integers. Much of public key cryptography uses extremely large integers, the kind that do not fit into an ordinary integer variable. Thus, bigint is an excellent package for trying out the algorithms of public key cryptography. Bigint's creator, Matt McCutchen, has put it in the public domain. Of course, the larger the integers are, the slower any computations will run. Still, it is often possible to work with a hundred or more decimal digits in a reasonable amount of time. Although bigint is no longer maintained, it works well for the example algorithms discussed here. The author of this paper has successfully used bigint on a Linux server for over a decade as a significant part of a cryptography course taught to cybersecurity and computer science majors.

2. SHORT BIGINT EXAMPLES

The bigint package uses .cc files containing C++ code and filenames with a .hh extension for its header files. It provides a test program that tries out and illustrates the functionality that is provided for these big integers. Most of the constructs are familiar items from C++ and C. What is new is using these items on arbitrarily large integers as well as some of the operations that can be done on these integers.

There are two data types, `BigInteger` and `BigUnsigned`, for large integer variables. You can simply assign an ordinary `int` into a variable having one of these new types. You can also calculate a huge value having one of these two types and assign it into a `bigint` variable. To copy a value from a `BigInteger` to an ordinary `int` variable, a `toInt()` conversion function must be used. An exception is thrown if the value is too large to fit into an ordinary `int`. You can also convert a string of digits into a `BigInteger` by using the `stringToBigInteger` conversion function. Here are some examples of putting a number into a `BigInteger` variable:

```
BigInteger a;  // a contains 0 by default
int b = 2047;
a = b;  // Converts int into a BigInteger
b = a.toInt();  // Converts BigInteger to
// ordinary int (if it will fit)
BigInteger c(a); // Copy a to BigInteger c
// Put int literal into BigInteger d:
BigInteger d(-314159265);
```

There is no `BigInteger` literal, but you can convert a string of digits to a `BigInteger` as follows:

```
string s("31415926535897932384626433832");
BigInteger e = stringToBigInteger(s);
cout << e << endl;
```

You can also work in hexadecimal, which is sometimes convenient. Note how this example switches to hexadecimal, does some bitwise Boolean operations and shifts, and then returns to decimal for future output:

```
BigUnsigned i(0xFF0000FF), j(0x0000FFFF);
cout.flags(ios::hex | ios::showbase);
// The << operator uses these flags.
cout << (i & j) <<  // Boolean AND is &
   << (i | j) <<    // Boolean OR is |
   << (i ^ j) <<    // Exclusive OR is ^
   // Shift distances are unsigned ints
   << (j << 5) <<   // Shift left 5 bits
   << (i >> 3) <<   // Shift right 3 bits
   cout.flags(ios::dec);// Now to decimal
```

In the last lines of the above example, note that `j = 0000FFFF = 1111111111111111` and that we then shift this number left by 5 bits.

You may have noticed that we are seeing some of the math that is needed to do public key cryptography. In particular, we need operations on huge integers, operations such as powers, greatest common divisor (gcd), modular exponentiation (modexp), and modular inverse (modinv). Bigint has all of these.

We say that numbers a and b are congruent mod m if their difference is divisible by m. Thus $a \equiv b \pmod{m}$ means that $a - b = km$, for some integer k. Congruence mod m is heavily used in public key cryptography.

Now try an example where we calculate powers of 274 by repeated multiplication:

```
int max = 10;
BigUnsigned x(1), big274(274);
for (int power = 0; power <= max; power++)
   {
   cout << "274^" << power << " = " << x
      << endl;
   x *= big274;  // A bigint assignment
   }
```

In public key cryptography, it is very common to calculate a number to a power, but done mod some third integer. That's called modular exponentiation. Bigint has a built-in `modexp` function for this. Here is some BigInteger code to calculate a greatest common divisor, a modular inverse, and a modular exponentiation, though the numbers could be much larger than what is used here:

```
cout << gcd(BigUnsigned(60), 72) << '\n'
   << modinv(BigUnsigned(7), 11) << '\n'
   << modexp(BigUnsigned(314), 159, 2653)
   << endl;
```

Notice that `modinv(y, n)` finds a value, which when multiplied by y, produces 1 (mod n). Thus, the example with 7 and 11 should produce 8 as the inverse, since 7 * 8 = 56 and $56 \equiv 1 \pmod{11}$, where $\equiv$ indicates congruence. If we convert that congruence to an equation, it would say that $56 = 1 + 11k$ for some integer k. (In fact, it is clear that k = 5 makes this statement to be true.) Note that `modexp(r, s, n)` finds r to the s power, with the result reduced mod n.

3. USING BIGINT

Installing bigint on a Linux system involves copying folders of files to each user's directory, including files whose names fit the patterns BigInteger* or BigUnsigned*, as well as ones named sample.cc, Makefile, and a few others. Makefile uses your Linux installation's g++ compiler for C++. (If you don't have C++ installed, you need to install it for bigint to work.)

The simplest way to start is to edit the sample.cc file, which contains a test program, and replace the test code with your own code. Then enter make at the command line to compile your sample.cc program. If there are error messages, edit sample.cc to make corrections. Otherwise, run your program by entering ./filename but with filename replaced by the name of your compiled program. By default this will be the name of your .cc file with the .cc omitted.

You can also name your C++ file something other than sample.cc as long as you make two small changes to Makefile. In that file, find the comment "Components of the program". On the next two lines, change sample and sample.o to use the actual name of your C++ file. For example, you might use program4 and program4.o instead of the original names. Once your program compiles, you can run it by entering ./program4 at the command line. Use the actual name of your compiled program, of course. When you are ready to create another bigint program, just make a copy of the entire folder containing the current program, change the code in the new folder, run make, and if make is successful, use ./filename to run your program.

4. BASIC NUMBER THEORY FUNCTIONS

Let's consider again these three functions: `gcd`, `modinv`, and `modexp`. Public key cryptography uses them extensively. This gives us hope that we can implement cryptographic routines such as those in RSA and elliptic curve cryptography. However, we need a few other key items, such as the ability to generate large primes. Here are a few of the routines. They use the C++ `rand()` function which generates a random integer between 0 and the constant `RAND_MAX`. The reader can refer to cryptography textbooks for explanations of these and other related functions. The purpose of including these functions here is not to teach public key cryptography but simply to show that we have all of the machinery needed to do public key cryptography. Students can then run cryptography exercises on a computer and not simply read the descriptions of public key cryptography in a text. Hands-on work is quite possible! Let's try some here.

First, we have some functions that produce random digits:

```
char RandomDigit(void)
   {
   int digit;
   digit = (9.999 * rand()) / RAND_MAX;
   return digit + '0';
   }


char RandomNonzeroDigit(void)
   {
   int digit;
   digit = (8.999 * rand())/RAND_MAX + 1;
   return digit + '0';
   }


char RandomOddDigit(void)
   {
   int digit;
   digit = (4.999 * rand()) / RAND_MAX;
   return 2 * digit + 1 + '0';
   }
```

Next is a function that generates a positive integer:

```
void GeneratePosIntPlain(BigUnsigned &
PosInt, int NumDigits)
   {
   int k;
   string s;

   s = RandomNonzeroDigit();
   for (k = 1; k < NumDigits; k++)
      s = s + RandomDigit();
   PosInt = stringToBigUnsigned(s);
   }
```

Then we have a function to generate a random string of digits. The string that is produced has a set number of digits.

```
string GenerateString(int NumDigits)
   {
   int k;
   char nonzero[2];
   nonzero[0] = RandomNonzeroDigit();
   nonzero[1] = '\0';
   string s(nonzero);
   for (k = 2; k < NumDigits; k++)
      s = s + RandomDigit();
   return s + RandomOddDigit();
   }
```

The strategy for getting a prime number of a

desired length is the same one that is used elsewhere in public key cryptography: generate strings of digits and then use appropriate primality tests as many times as needed to find a string that represents a prime number with probability as close to 1 (that is, 100%) as desired. The Fermat primality test (Trappe & Wahington, 2006) is shown here:

```
// Global constants for speed:
const BigUnsigned BigOne = BigUnsigned(1);
const BigUnsigned BigTwo = BigUnsigned(2);
const BigUnsigned BigThree =
   BigUnsigned(3);
const BigUnsigned BigFive =
   BigUnsigned(5);
const BigUnsigned BigSeven =
   BigUnsigned(7);
```

The following is a function to generate a prime number with a set number of digits:

```
// Assumes srand(time(NULL)) was done to
// seed the random number generator.
BigUnsigned GeneratePrime(int NumDigits)
   {
   BigUnsigned candidate;
   string s = GenerateString(NumDigits);
   candidate = stringToBigUnsigned(s);
   // Fermat primality tests:
   while(! PassPrimalityTests(candidate))
      {
      string s=GenerateString(NumDigits);
      candidate = stringToBigUnsigned(s);
      }
   return candidate;
   }
```

Next is a function to run tests to see if integer m is probably prime:

```
bool PassPrimalityTests(const BigUnsigned
& m)
   {
   if (! FermatPrimalityTest(BigTwo, m))
   // Uses global constant defined above.
      return false;
   if (! FermatPrimalityTest(BigThree, m))
      return false;
   if (! FermatPrimalityTest(BigFive, m))
      return false;
   if (! FermatPrimalityTest(BigSeven, m))
      return false;
   return true; // Hope it really is prime!
   }
```

The previous function uses the following function that performs the Fermat primality test.

```
// Base a to see if m is probably prime.
bool FermatPrimalityTest(const
   BigUnsigned & a, const BigUnsigned & m)
   {
   if (modexp(a, m - 1, m) == BigOne)
   // Check if a^(m-1) gives 1, mod m.
      return true;
   else
      return false;
   }
```

Miller-Rabin primality testing can be implemented (Trappe & Washington, 2006) in a similar way. By applying this test for n values of variable a, we can guarantee that the probability that a number is not prime when it passes n of the checks described in the Miller-Rabin algorithm is less than (1/4)^n. Thus, we can make this probability arbitrarily small.

5. PUBLIC KEY CRYPTOGRAPHY

We can now try out cryptographic algorithms such as RSA, elliptic curve encryption and decryption, etc. Here is a typical way in which a textbook might describe RSA encryption and decryption:

Bob's computer picks large secret primes p and q.

It calculates phi = (p – 1) * (q – 1).

The computer calculates n = p * q.
It further chooses an integer encryption exponent e so that gcd(e, phi) = 1.

It goes on to find an integer decryption exponent d with $d * e \equiv 1 \pmod{phi}$.

The sender's computer makes n and e public, but keeps p, q, phi, and d private.

Alice encrypts a numeric message m as $c = m^e \pmod{n}$ and sends it, c, to Bob.

Bob decrypts c by using $m = c^d \pmod{n}$.

It would seem to be difficult to translate the above into code that would run on a computer, but with bigint, it is fairly easy. That's why it is useful for examples in the classroom and for student homework.

Here is what we saw above in outline, now written using C++ and bigint code that students can run. In this example, the one program is both the sender and receiver, but the two pieces of functionality could be split out and given to a pair of students, one of whom sends the encrypted message to the other, who then decrypts it. Some minor details have been omitted. The initialization stage looks like this:

```
int PrimeLength;
// Seed the random number generator:
srand(time(NULL));
BigUnsigned m, c, e, d, decrypted;
cout << "Enter number of decimal digits"
   "for primes p and q (e.g. 120): ";
cin >> PrimeLength;
BigUnsigned p =
   GeneratePrime(PrimeLength);
BigUnsigned q =
   GeneratePrime(PrimeLength);
BigUnsigned n = p * q;
BigUnsigned phi = (p - 1) * (q - 1);
```

Next, the program generates a 4-digit prime and the encryption exponent e and decryption exponent d:

```
e = GeneratePrime(4);

// We need to have gcd(e, phi) = 1.
// Try until you we get one that works.
while (gcd(e, phi) != 1)
   e = GeneratePrime(4);

d = modinv(e, phi);
```

The next step is to ask the user (Alice) for a short message to be encrypted. It is written as a number.

```
string data;
cout << "Enter a message string: " << endl;
// For example, 010203 to represent ABC
cin >> data;
m = stringToBigUnsigned(data);
```

Now we encrypt the message m to produce the ciphertext c and then decrypt it so that Bob can read it:

```
c = modexp(m, e, n);
decrypted = modexp(c, d, n); // Decrypt c.
cout << "Decrypted  message m = " <<
   decrypted << endl << endl;
```

Finally, we check to see if the decrypted message matches the original plaintext message and report on the results:

```
cout << "original m & decrypted message "
if (m == decrypted)
   << cout << "match" << endl;
else
   cout << "do NOT match" << endl;
```

Similarly, bigint can be used to write programs that do elliptic curve cryptography, ElGamal encryption and decryption, etc.

6. CRYPTOGRAPHIC EXERCISES

Many types of cryptographic exercises can be tried with bigint. For students who have a good background in programming and math, it is quite possible to have them write programs in bigint that encrypt and decrypt messages using RSA, ElGamal, and elliptic curve cryptography, much like the brief RSA example just presented here. Textbooks are available that provide the details of the algorithms and the math on which they are based. Students would have to translate those algorithms into bigint code. They might also write bigint programs to try to break some encrypted message and perhaps time how long it takes. It would be useful to have as a parameter the minimum number of bits the primes have in them. That way, the difficulty of cracking the code can be easily varied. If the number of bits is too large, there is little chance of extracting the message, while too small of a number of bits would result in a very fast calculation of the plaintext message.

In some cryptography courses, the students might not have the math or programming background to do the type of exercises just discussed. There are other types of exercises that would be more appropriate. Some of these are timing exercises where you find out how long it takes to do something. For example, students could use a program that finds how long it takes to encrypt a message (supplied as input to the program). Then the same type of timing exercise could be used for the process of decrypting the ciphertext.

To be more specific, consider a timing exercise that begins by calculating n = p * q as in RSA. By printing a message just before and after this calculation, the program can show that this multiplication happens quickly. (An alternate approach is to have the program access the system time right before and after the calculation. Then the difference of those two times gives the approximate time for that calculation.

In contrast, if n is large enough, the factoring of n as p times q can take much longer, with a very noticeable or even prohibitive delay. This factoring could be done by trying successive values in a large table of primes to see if any of them divide n. The difference of two squares method could also be tried. In the latter, you need a large table of squares. You then add a square to n to see if you get a square. If that

works, you have $n + b^2 = a^2$, so that $n = a^2 - b^2 = (a + b)(a - b)$. In both methods, the table might be too small to factor n. However, even if the table is large enough, the checking of many table values may mean that the factoring of n takes a large amount of time. If RSA is implemented well and n is large enough, none of these factoring methods will succeed unless you can devote a large amount of time (perhaps years) to the factoring attempts. Students can try RSA with moderate values of n to see how long it takes to break the scheme by factoring n. They might also graph how this time increases as they slowly raise the number of digits in n.

Other exercises might have students encrypt a message, decrypt it, and verify that the original message is obtained. Still others might report how many collisions some cryptographic hash function (or even a non-cryptographic hash function) produces with a certain set of data. (A collision is when two data values hash to the same result.) As a general rule, the fewer collisions there are, the better the hash function is. Another type of exercise is to see how many hash function values you have to check to get a collision. Collisions are inevitable if you hash enough values, but they should be rare.

One strategy is to use a test program based on the birthday paradox. It has been shown that if the values fit the range [0, n-1], and some other technical conditions are met, having about the square root of n as the number of data items to be hashed is enough to have better than a 50% chance of a collision. A variation is to make two tables of hash values and look for a value in one table that also occurs in the other. A related exercise is to use the Baby Step, Giant Step algorithm (Trappe & Washington, 2006) that uses two lists to try to break ElGamal encryption, which is based on discrete logs. If students lack the math and programming background to create the software for this, they could simply be given the software and asked to see if they can use the software to get one or more collisions.

We next look at some of the specifics of the testing of a hash function using the two list approach. Below is a partial listing of a program that looks for collisions when testing a simple non-cryptographic hash function. Unless you have really strong students, you would probably want to give students the program and simply ask them to use it to find hash function collisions.

This program makes two lists of hash values for somewhat random inputs and looks for matches between the lists. The details of the matches (such as what input hashed to what value) and the total number of matches are printed. Both lists are 45000 items long. The hash values are 30 bits long. Using the analysis by Trappe and Washington (Trappe & Washington, 2006), N = 2^30 = 1073741824 possible hash values. Then sqrt(N) = 2^15 = 32768. Since each list is almost 50% longer than sqrt(N), namely length 45000, we expect a very good chance of a collision, a match. Note that lambda = (45000 ^ 2) / (2 ^ 30) = 1.886 roughly. Then the chance of a match is approximately 1 - e ^ (-lambda) = 0.848 = 84.8% Thus, a match is quite likely.

We begin with some initialization so that we can create the first table of hash values.

```
NumDigits = 61;
GeneratePosIntPlain(FirstRunStart,
  NumDigits);
cout << "FirstRunStart is " <<
   FirstRunStart << endl;


cout << "Creating a table of hash values"
   << endl << endl;
BigUnsigned Largest =
   stringToBigUnsigned("100truncated");
```

That last line should use 10 ^ 61, which is 1 followed by 61 zeros. It can't fit here, but a string containing a 1 followed by 61 zeros will work in the code.

Next, we produce the table of hash values, all of which have 61 bits:

```
for (k = 0; k < MAX; k++)
   {
   Increment = modexp(Two, k, Largest);
   k3 = FirstRunStart + Increment;
   k3 = k3 % Largest;
   table[k] = hash(k3);
   }
```

The next step is to set up to create the second table of hash values. These will all have 63 bits.

```
NumDigits = 63;
GeneratePosIntPlain(SecondRunStart,
   NumDigits);
num = SecondRunStart;
cout << "SecondRunStart is " <<
   SecondRunStart << endl << endl;
```

The last section generates the new hash values and checks if any of them match a hash value in the first table.

```
for (m = 0; m < MAX; m++)
   {
   GeneratePosIntPlain(num, NumDigits);
   value = hash(num);
   // Sequential search table for value
   index = SequentialSearch(value);
   if (index >= 0)
      {
      matches++;
      bigindex = BigUnsigned(index);
      index3 = FirstRunStart +
         modexp(Two, bigindex, Largest);
      index3 = index3 % Largest;
      cout << "Matching values found at "
      << index3 << " and " << num << endl;
      cout << "hash of first one: " <<
         hash(index3) << endl;
      cout << "hash of second one: " <<
         hash(num) << endl << endl;
      }
   num++;
  }
cout << "Number matches: " << matches
   << endl << endl;
```

For those who teach ElGamal encryption, a useful exercise is to have students write, or simply use, a small program that prints the powers of the numbers 1 through n-1 for some positive number n, where the powers are reduced modulo n (Stallings, 2020). It makes it easy to see which of the numbers 1 through n-1 is a primitive root of n, since in such a row all of the computed values are distinct. Note that a primitive root might be better named a "multiplicative generator". Students could have their program print an asterisk after each row in which the starting number is a primitive root of n. The following is the output of this program when 11 is chosen as the prime. It flags 2, 6, 7, and 8 as the primitive roots mod 11.

Enter a prime number less than 44: 11
Table of powers of a modulo 11 where the powers for a are on the next line:

```
 1  2  3  4  5  6  7  8  9 10

 1  1  1  1  1  1  1  1  1  1
 2  4  8  5 10 9  7  3  6  1*
 3  9  5  4  1  3  9  5  4  1
 4  5  9  3  1  4  5  9  3  1
 5  3  4  9  1  5  3  4  9  1
 6  3  7  9 10 5  8  4  2  1*
 7  5  2  3 10 4  6  9  8  1*
 8  9  6  4 10 3  2  5  7  1*
 9  4  3  5  1  9  4  3  5  1
10 1 10 1 10  1 10 1 10 1
```

This table can also be read backwards to find discrete logarithms. For example, if we look in the row starting with 2 for the item in column 4, we get a 5. That indicates that 2^4 gives 5 when working modulo 11. That's correct since 2^4 = 16, which is congruent to 5 when we are working mod 11. We can also read this calculation backwards to say that the discrete log of 5 (when using base 2 and modulus 11) is 4.

7. CONCLUSIONS

This article has made the case that the bigint software package is a very useful tool in the teaching of cryptography. It can easily deal with integers having one hundred or so decimal digits, which is much more realistic than using ordinary numbers of type int. It is also free, which is a big help when budgets are tight. Of course, other types of exercises are likely to be needed in addition to the ones that use bigint.

The presentation here does not try to cover all areas of public key cryptography. Rather, it simply shows some representative examples. Professors who teach cryptography are welcome to contact the author for bigint examples and homeworks. These would be best placed on a Linux system where students could copy these items to their personal folders. Some of the examples that have students run a program several times and record what was produced can be done in less than an hour; ones that involve coding might take several hours. Students almost always succeed at that first type of problem, but some find those involving coding to be challenging. Still, the majority of students succeed on this. Since the author's class is typically small, any statistics on this beyond this general trend are probably meaningless.

It should be noted that public key cryptography is expected to be phased out around 2030. By then, it is likely that quantum computers will be approaching the power needed to break a lot of public key cryptography. The U.S. National Institute of Standards and Technology (NIST) has already published its first group of algorithms for quantum-resistant cryptography (NIST, n.d.; NIST, 2023). Many are based on the closest vector problem or the shortest vector problem in a high-dimensional space. That would be a quite different approach to cryptography. Public key cryptography is expected to still be taught but will likely be done more for historical interest. The main emphasis will instead be on private key methods (thought to be quantum-resistant) and new types of quantum-resistant cryptography.

8. REFERENCES

Elbirt, A. (2009). Understanding and Applying Cryptography and Data Security. Taylor & Francis Group, LLC.

Katz, J., & Lindell, Y. (2020). Introduction to Modern Cryptography, 3rd ed. CRC Press.

McCutchen M. (2010), C++ Big Integer Library. Retrieved August 21, 2023 from https://mattmccutchen.net/bigint/.

National Institute of Standards and Technology (n.d.). National Cybersecurity Center of Excellence. Migration to Post-Quantum Cryptography. Retrieved August 21, 2023 from https://www.nccoe.nist.gov/crypto-agility-considerations-migrating-post-quantum-cryptographic-algorithms.

National Institute of Standards and Technology. Computer Security Resource Center. Selected Algorithms 2022. Last modified August 14, 2023. Retrieved August 21, 2023 from https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

Stallings, W. (2020). Cryptography and Network Security: Principles and Practice, 8th ed. Pearson Education, USA.

Stinson, D., & Paterson, M. (2018). Cryptography: Theory and Practice, 4th ed. CRC Press.

Trappe, W., & Washington, L. (2006). Introduction to Cryptography with Coding Theory, 2nd ed. Pearson Prentice Hall.