Assignment 1 - Part 2
CS 135 Review - RSA Cryptography

Benjamin Zofcin
CS 202 - Fall 2023

1 Introduction

Congratulations on making it past the first part assignment 1. In the previous part you became reacquainted with the following

- basic style of C/C++ syntax
- declaring variables
- structuring logic (if/else)
- constructing loops (for and while)
- invoking functions, parameter passing, and return values
- interacting with command line arguments
- interacting with terminal input
- opening and closing files

The goal for this part of the assignment is to complete the review of CS 135 topics and have you learn something about RSA Encryption along the way.

2 Background

As you may have guessed from the password you decrypted in part 1, stream encryption is fundamentally weak (especially without character mixing) and unable to keep up with modern brute force attacks. For this reason, stream ciphers are hardly (if ever) used in modern day applications. Between the end of WWII and the early 1970's a need for newer and stronger encryption algorithms were necessary.
The goal of any encryption algorithm is to develope a function that can not be easily inverted. Functions like offset and rotation simply do not fit this requirement. After writing the code for the first part, it should be clear as to why. In light of this, several encryption algorithms were conceptualized and formally introduced by Ron Rivest, Adi Shamir, and Leonard Adleman. After many approaches, the research group had determined that (for the time) key generation, key sharing, and block based encryption methods were superior. To implement their RSA encryption algorithm we are going to recall some mathematics from the last part and introduce more mathematics for this part.

3 Purpose

While I do want you get an idea of how the RSA encryption algorithm works, an in depth understanding is rather complicated and unnecessary for the completion of this assignment (this is a review assignment after all). Instead the primary focus of this assignment will be on the following topics from CS 135:

- Looping constructs
- Prompting and re-prompting (on "failure") the user for data with looping constructs
- Writing and reading data to and from files
- Converting mathematical algorithms to code
- Passing variables by reference and by value

4 Relevant Algorithms and Associative Code

Let's start by describing how the RSA algorithm processes the original information differently than stream ciphers.

4.1 Character Processing

In the previous part we saw how the the modified Caesar cipher algorithms would manipulate the characters of a string (or message) on character at a time. In this way, we consider each character to independent from one another and has no bearing on the overall pattern of encryption. The first deviation RSA takes from its predecessor is by grouping characters together and encrypting them in what we call "blocks".

E.g., Consider the message "The orange cat, sleeping quietly, awoke." and a block size of 4. The string would then be partitioned into block of 4 characters and each block would be encrypted.

The	oran	ge c	at,	slee	ping	qui	etly	, aw	oke.

4.1.1 Relevant Function: prompt_user_block(int&)

Before we start partitioning the data into blocks, we need the user to tell us how big each block should be. The only stipulation, is that the block size must be larger than 1. If the user attempts to give us a value other than that, we will display a relevant error message and prompt them for a block size again.

Just like the last part, you should be uncommenting the functions in main one at a time to test your code as you go. Now, consider the following main function:

```
0 int main(int argc, char const* argv[]) {
    int block;
2   prompt_user_block(block);
}
```

The following is an example of the programs behavior.

```
0 [z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
  [z_xps] rsa_encryption <=> ./a.out
2 Enter plaintext block size: 0
  ERROR: block size must be greater than 1!
4 Enter plaintext block size: 1
  ERROR: block size must be greater than 1!
6 Enter plaintext block size: 2
  [z_xps] rsa_encryption <=>
```

The first thing you should note about the majority of the functions in this part, is the parameters are mostly passed by reference. If you don't recall what the difference between pass by value and pass by reference is, you should review that now.

Some of you may be asking yourselves something along the lines of, "What happens if our message length isn't a multiple of our block size?".

We got lucky in the earlier example with a string length of 40 and a block size of 4 to give us 10 total and full blocks of characters. Unfortunately, the world isn't always so kind and the way we would solve this issue is by a method called padding and we aren't going to worry about that here. Just be careful when testing your programs to define a block size that evenly partitions your message.

4.2 Encryption Method

The encryption methods of the Caesear cipher were quite simple.

1. A characters traversal along a number line, provided an offset
2. A characters movement along the domain of a string, provided a rotation

Not only was the encryption to perform, the functional inverse (oppositional direction of movement in the domain) for decryption was equally as simple and not at all difficult to perform (even without the original offset or rotation value) given some time.
The **encryption** method for RSA is, in a similar vein, simple. However, to perform RSA **decryption** without the key would take an incredibly long time to break. To understand why this is let's first talk about prime numbers.

4.2.1 Prime Numbers

Every integer value can be represented by a multiplication of smaller or equivalent integer values. The set of integer multiplications is called a factorization.
E.g., Assume the value 16. 16 can be represented by any of the following multiplications.

$$16 = 1 * 16$$
$$16 = 2 * 8$$
$$16 = 2 * 2 * 4$$
$$16 = 2 * 2 * 2 * 2$$
$$16 = 4 * 4$$

Now, take the value 17. 17 can be represented by any of the following multiplications.

$$17 = 1 * 17$$

It is said that, for any number whose factorization contains only 1 and itself, then that number is a prime value. Alternatively, it can also be stated that for all values (i) greater than 3 and less than the square root of the original number (n), if none of the values in i evenly divide n, then n is prime.

$$\forall_i \; if \; (n \; \textbf{mod} \; i \; == \; 0) \; n \notin \mathbb{P}, \; 3 < i \leq \sqrt{n}$$

Relevant function: **prime_test(long)**

This function will be used to determine if some input value is prime or not.
I've already implemented a part of this function for you to test for and values divisible by 2 and 3, as well as some other checks. The rest is up to you to implement by converting the mathematical equation above into code. Notice,that the function is a boolean returning function and write your code accordingly.

4.2.2 Generate the RSA Keys

Now let's take a look at the RSA key generation, then we will break it down one step at a time. The RSA encryption process is as follows:

1. Pick 2 primes p and q such that $p \neq q$
2. Compute $n = p * q$ such that $n > 27^b$ where b is the block size
3. Compute $\phi(n) = (p-1) * (q-1)$
4. Pick a public key (e) whose value is coprime to $\phi(n)$
5. Compute private key (d) such that $d \equiv e^{-1} \mathbf{mod} \phi(n)$

Selecting Prime numbers: _prompt_user_prime(int&)

There was a reason we've went through the effort of determining what a prime number is. Now you will implement the _prompt_user_prime function in a similar manner to the prompt_user_block function. This function will repeatedly prompt the user for some input and check if that value is a prime number or not by calling the prime_test function. If the value isn't prime, display an error message and prompt the user for another value.
Assume the following main function:

```
0 int main(int argc, char const* argv[]) {
    int p;
2   _prompt_user_prime(p);
}
```

The following is an example of the programs behavior.

```
0 [z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
  [z_xps] rsa_encryption <=> ./a.out
2 Enter sufficiently large prime number: 0
  ERROR: 0 is not prime!
4 Enter sufficiently large prime number: 14
  ERROR: 14 is not prime!
6 Enter sufficiently large prime number: 22
  ERROR: 22 is not prime!
8 Enter sufficiently large prime number: 10000000
  ERROR: 10000000 is not prime!
10 Enter sufficiently large prime number: 17
  [z_xps] rsa_encryption <=>
```

1 & 2. Checking p and q for sufficiency: prompt_user_prime(int&, int&, int)

Now that we can prompt the user for a single prime, we will prompt them for 2 and ensure that their values meet the following conditions:

- $p \neq q$
- $p * q > 27^b$

We're not yet ready to understand why these checks are necessary but they are.

I've already implemented the logical structure in the prompt_user_prime function, you need only call the _prompt_user_prime with actual parameters 1 and 2 and then insert the conditional expressions to the if/else block. (Hint: you will need the pow function from the math library)

```
int main(int argc, char const* argv[]) {
  int block, p, q;

  prompt_user_block(block);

  prompt_user_prime(p, q, block);
}
```

The following is an example of the programs behavior.

```
[z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
[z_xps] rsa_encryption <=> ./a.out
Enter plaintext block size: 4
Enter sufficiently large prime number: 113
Enter sufficiently large prime number: 113
ERROR: p cannot equal q!
Enter sufficiently large prime number: 113
Enter sufficiently large prime number: 181
ERROR: p * q must be larger than 27^4!
Enter sufficiently large prime number: 1117
Enter sufficiently large prime number: 3797
[z_xps] rsa_encryption <=>
```

3. Eulers Totient

This is one of those mathematical parts of RSA that goes beyond the scope of "just a review" assignment at the undergraduate level, so I have already structure the main function to compute the Eulers Totient after the prompt_user_prime function successfully completes. Though, I am not discouraging you from leaning about it on your own time. Here's a link to a Wiki article on it.

4. Choosing a public key: prompt_user_relative_prime(long&, long)

Next you will have to prompt the user for a value that is coprime (or relatively prime) to the Eulers Totient value. This value, the public key (e), must satisfy the following:

$$2 < e < \phi(n)$$
$$\&$$
$$gcd(e, \phi(n)) = 1$$

2 values are said to be coprime if the greatest common divisor between the 2 values is 1. I have already implemented the gcd function for you. This function must prompt the user for a value of e until the equations above have been satisfied.
Assume the following main function:

```
int main(int argc, char const* argv[]) {
  int block, p, q;
  long n, phi_n, e, d;

  prompt_user_block(block);

  prompt_user_prime(p, q, block);

  n = p * q;
  phi_n = (p - 1) * (q - 1);

  prompt_user_relative_prime(e, phi_n);
}
```

The following is an example of the programs behavior.

```
[z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
[z_xps] rsa_encryption <=> ./a.out
Enter plaintext block size: 3
Enter sufficiently large prime number: 113
Enter sufficiently large prime number: 181
Enter value that is relative prime to 20160: 20161
ERROR: value must be less than 20160
Enter value that is relative prime to 20160: 1200
ERROR: 1200 is not relatively prime to 20160!
Enter value that is relative prime to 20160: 1769
[z_xps] rsa_encryption <=>
```

If you are unsure of how to use the gcd function I've implemented, look to the `generate_relative_primes` function for an example.

5. Generating the private key: generate_private_key(long, long)

Generating the private key is last piece to the puzzle before we start encrypting and decrypting data. Again, this is one of those processes that goes beyond this assignment. Should you wish to learn more about modular arithmetic and congruency, here is a link to a Wiki article that explains it in detail.

Assume the following main function:

```
int main(int argc, char const* argv[]) {
  int block, p, q;
  long n, phi_n, e, d;

  prompt_user_block(block);

  prompt_user_prime(p, q, block);

  n = p * q;
  phi_n = (p - 1) * (q - 1);

  prompt_user_relative_prime(e, phi_n);

  d = generate_private_key(e, phi_n);

  cout << "Private key = " << d << endl;
}
```

If you have done everything correct up to this point, your output should be as follows

```
[z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
[z_xps] rsa_encryption <=> ./a.out
Enter plaintext block size: 3
Enter sufficiently large prime number: 113
Enter sufficiently large prime number: 181
Enter value that is relative prime to 20160: 1769
Private key = 4889
[z_xps] rsa_encryption <=>
```

4.2.3 Encrypting the block with the public key

This is it, one last function and assignment 1 is done!
The first thing you are going to need to do is drag and drop the following functions from part 1 into this file

- `_open_input_file`
- `open_input_file`
- `_open_output_file`
- `open_output_file`

Now that we have the ability to read and write data to files again, there is a file called input.txt with a single line of text in it. We want to read that line of text, encrypt it using RSA and write the resulting cipher to an output file of our choosing. But, you may be asking, "How do we encrypt the data?"

The mathematical expressions for encryption and decryption are as follows:

$$C = E(P) \equiv P^e \mathbf{mod} n$$
$$P = D(C) \equiv C^d \mathbf{mod} n$$

Or, in English,

- The value of the cipher text is equivalent to the value of the plain text raised to the power of the public key mod n
- The value of the decrypted cipher text is equivalent to the value of the cipher text raised to the power of the private key mod n.

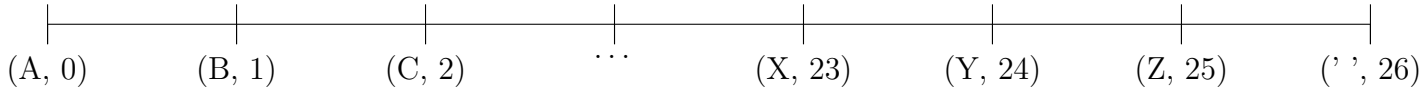
At this point I'm thinking there are 2 questions or concerns going on in your mind

1. What is the "value" of text?
2. Isn't something raised to the powers of e and d going to be enormous?

Well, don't worry too much about the second one; as there is some mathemagical trickery happening in the apply_key function to implement the function without doing the math.

As for the first question, however; that's the last thing you need to for this assignment and it's a bit involved.

Let's think back to Caesar and how in offset encryption, we associate characters with values. In the English language, we have a set of letters {A,B,C, ... , X, Y, Z, ' '}. Assume that each one of these letters can be related to a value on a subset of the integer number line.



Now let's recall the equation for series composition and alter it for RSA

Figure 1: Series Composition

$$\sum_{i=0}^{b} M[i] * B^i,$$

$b :=$ block size

$M :=$ character value at index i in sub-message M

$B :=$ Base system

Only this time, instead of evaluating numbers in our base 10 system, we are evaluating characters in base 27.

Let's take a look at an example of what this would look like for the first set of 3 characters.
E.g.,
message = CONGRATULATIONS, b = 3, B = 27 p = 113, q = 181, n = 20160, e = 1769,

First, recall that RSA will analyze P, 3 characters at a time so the first set of characters to be analyzed are CON.

Then convert the characters to their numerical equivalent.

$$message[0,2] \implies M = \{N, O, C\} = \{13, 14, 2\}$$

$$\begin{aligned} P &= \sum_{i=0}^{3} P[i] * 27^i \\ &= 13 * 27^0 + 14 * 27^1 + 2 * 27^2 \\ &= 13 * 1 + 14 * 27 + 2 * 729 \\ &= 13 + 378 + 1458 \\ &= 13 + 378 + 1458 \\ &= 1849 \end{aligned}$$

$$\begin{aligned} C &= E(P) \equiv P^e \mathbf{mod} n \\ &= 1849^{1769} \mathbf{mod} 20160 \\ &= 10722 \end{aligned}$$

The value of 10722 is stored away in an output file (you'll see it's the first number in cipher.txt) and the process is repeated for the remainder of the substrings in the message.

encrypt_message(long, long, int)

It is now your job to implement and apply the algorithm above to every block of characters in a message. Much like the rotate algorithm, there are many different ways to implement this algorithm. Once you are confident that your algorithm works your program should perform as follows.

Assume the following main function:

```
int main(int argc, char const* argv[]) {
  int block, p, q;
  long n, phi_n, e, d;

  prompt_user_block(block);

  prompt_user_prime(p, q, block);

  n = p * q;
  phi_n = (p - 1) * (q - 1);

  prompt_user_relative_prime(e, phi_n);

  d = generate_private_key(e, phi_n);

  cout << "Private key = " << d << endl;

  cout << "Encrypting Message" << endl;
  encrypt_message(e, n, block);

  cout << "Decrypting Cipher" << endl;
  decrypt_message(d, n, block);
}
```

If you have done everything correct up to this point, your output should be as follows

```
[z_xps] rsa_encryption <=> g++ rsa_crypto.cpp
[z_xps] rsa_encryption <=> ./a.out
Enter plaintext block size: 3
Enter sufficiently large prime number: 113
Enter sufficiently large prime number: 181
Enter value that is relative prime to 20160: 1769
Private key = 4889
Encrypting Message
Please enter the name of your input file: input.txt
Please enter the name of your output file: student_cipher.txt
Decrypting Cipher
Please enter the name of your input file: student_cipher.txt
CONGRATULATIONS
[z_xps] rsa_encryption <=>
```

If your student_cipher.txt file is identical to my cipher.txt file, then you implemented the series composition algorithm correctly.