Assignment 5: RSA Encryption

Aniket Pratap

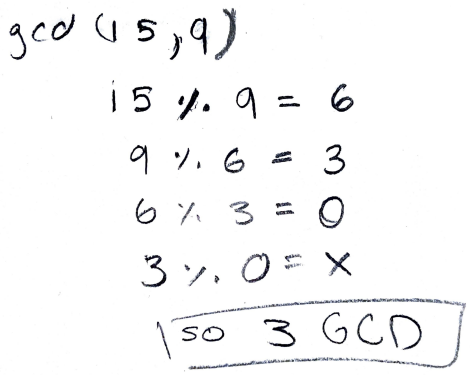
February 2022

1 Introduction

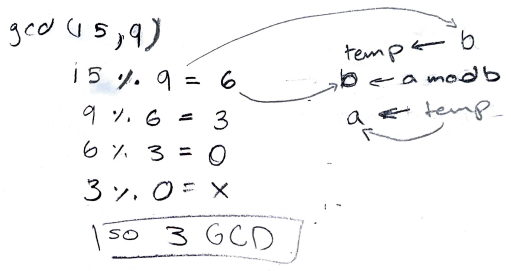
The purpose of this assignment is code our own RSA encryption method using several files and functions. The user inputs a message which gets encrypted using a public key. This public key can be accessed by anyone when talking about the internet, but in this case, we decrypt it after using our private key. The private key is not seen by anyone or even us. In real life, the server would have access to this. After decrptying the message, we obtain our original message, which only we can open due to our private key.

2 GCD

This function is where all the arithmetic for RSA encryption happens. I can begin by using uint32's and later convert them to the GMP format. The first function was GCD. The implementation can be as follows:

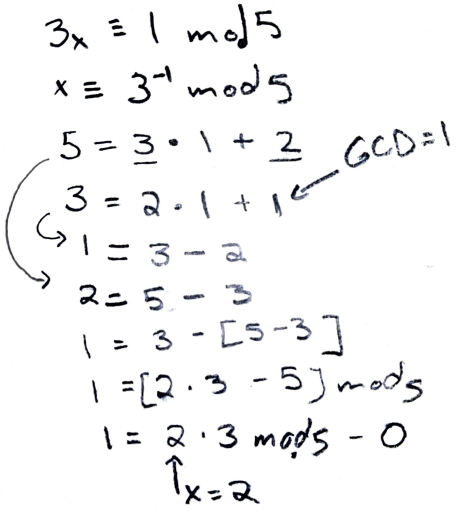


This representation is saying that the GCD of 15 and 9 for example, is the same as the GCD between 9 and 6. This patter continues on until the modulo of 0 is being used–which is impossible. Hence, the GCD would be what 0 is modding–in this case 3. A temporary variable would be needed which takes 9, which we will call be, and set it to a, which the remainder will be modding:



3 Mod Inverse

The mod inverse functions determines if there is an inverse using GCD and the euclidean algorithm. An example can be showed:



the first thing to do could be to write the base (a) in terms of the modulo (n). This can be done by setting a variable q = $\lfloor \frac{n}{a} \rfloor$. Then I need to find the what (l) plus the aq = n. This can be done by doing $n - aq$. I can do this same process for the next iteration in a while loop as well. Once, I have my vales, I have to set everything equal to l for all iterations. l = (simplified equation). I can then substitute the first equation in terms of the next equations until I get an equation in terms of a and n. If I mod the equation I can then find x or the inverse. The GCD is utilized in the process to check if there even is an inverse. If the GCD is not 1, an inverse can't be possible and the function can be halted. This pseudo code can be implemented as given in the assignment documentation:

```
MOD-INVERSE(a, n)
 1  (r, r') ← (n, a)
 2  (t, t') ← (0, 1)
 3  while r' ≠ 0
 4        q ← ⌊r/r'⌋
 5        (r, r') ← (r', r − q × r')
 6        (t, t') ← (t', t − q × t')
 7  if r > 1
 8        return no inverse
 9  if t < 0
10        t ← t + n
11  return t
```

4 Is Prime

isPrime is a function that checks if a given number is prime. Since all prime numbers are odd, except 2, we can say that if something is even, then break. The pseudo code can be represented by this from the assignment 5 documentation:

```
MILLER-RABIN(n, k)
 1  write n − 1 = 2^s r such that r is odd
 2  for i ← 1 to k
 3      choose random a ∈ {2, 3, ..., n − 2}
 4      y = POWER-MOD(a, r, n)
 5      if y ≠ 1 and y ≠ n − 1
 6          j ← 1
 7          while j ≤ s − 1 and y ≠ n − 1
 8              y ← POWER-MOD(y, 2, n)
 9              if y == 1
10                  return FALSE
11              j ← j + 1
12          if y ≠ n − 1
13              return FALSE
14  return TRUE
```

The task "write $n-1 = 2^s \times r$" states that any number, assuming it's odd, can be represented by 2 times some power times some odd r value. For example, say you have the value 17. 17 - 1= 16, which is even. This means that it can be represented by $2^s * r$. The best way to do this is to loop until r is an odd number. This loop can be showed as:

$16 = 2^0 \cdot 16$
$16 = 2^1 \cdot 8$
$16 = 2^2 \cdot 4$
$16 = 2^3 \cdot 2$
$16 = 2^4 \cdot 1$

In this case, the counter can be called i starting at 0, and the multiplier can be r, starting at $n-1$. Since we increment the exponent, of 2 by 1, we need to divide r by the corresponding exponent. This can be done by a simple while loop until r is odd:

```
s=0  r=n-1
while (r is not odd)
    s+=1, r/=2
```

The use of powMod is used to check whether, the number is composite or not. If the power mod equals 1 however, the number might be prime. The more times we continue this powMod process, the more time our accuracy increases. A number will be randomized using the function make prime, which generates a prime number. This randomized process will be done in a separate file using GMP.

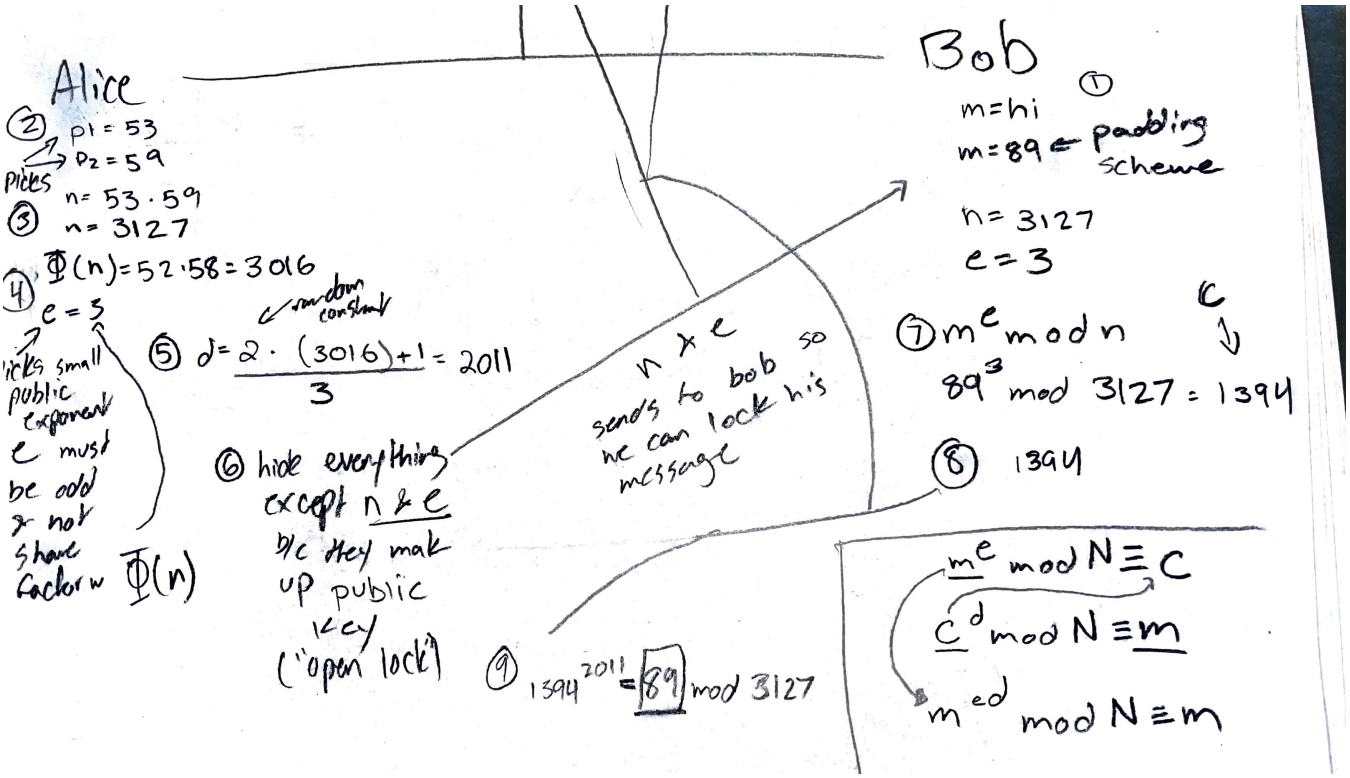
5 Pow Mod

This the function that is utilized in the actual encryption and decryption process. This function checks if the exponent is even or odd. Depending on this result, the function multiples the base accordingly. If even, multiply the base twice in a loop where if odd, multiply it once. This value is then modded with the modulo so that it can repeat the process until the exponent equals zero. The pseudo code can be represented by the following form the assignment, p represents the base and v represents the coefficient to multiply if the exponent is odd:

```
POWER-MOD(a, d, n)
1  v ← 1
2  p ← a
3  while d > 0
4       if ODD(d)
5            v ← (v × p) mod n
6       p ← (p × p) mod n
7       d ← ⌊d/2⌋
8  return v
```

6 RSA Process

The RSA process requires the equations $m^e \mod N \equiv c$ and $c^d \mod N \equiv m$ which together equals $m^e d \mod N \equiv m$. The process starts as follows. Some very large prime values are selected at random which we will call p1 and p2. These numbers are then used to for a number n. The reason this is done is because multiplying p1 and p2 to get n only takes a matter of seconds for the computers, but factoring n requires much, much more time–exponential in fact. Next, the $\phi(n)$ is calculated. However, in this assignment, we will use $\lambda(n)$. For this design however, I will talk about $\phi(n)$ since it's simpler. $\phi(n)$ asks how many numbers are there less than r equal to n that doesn't share any common factors greater than 1. If the number is already prime, then the amount of numbers without a common factor before it would be $n-1$. Since ϕ is multiplicative, we can say that $\phi(n) = \phi(p1) * \phi(p2)$, which in turn becomes: $\phi(n) = (p1-1) \times (p2-1)$. Then we must pick a number e so that it's odd and doesn't share a factor with $\phi(n)$. This e along with the n will serve as our public keys. The m in the first equation will serve as our message. e will serve as our message encryptor in this case. In order to create the decryptor or the private key, we need to create some d so that $d = \frac{k \times \phi(n)+1}{e}$. This undoes the effect of e and allows the message to be decrypted. This private key is not shared with anyone. So, the message is sent encrypted using the equation $m^e \mod N \equiv c$. The c is then sent over where it is deecrypted using the d private key. This result will give the number, which the original message was translated into. In essence, the process should look as follows:



The RSA functions are straight fore ward to appply as they use the numtheory.c library. The difficult part would be encrypting and decrypting the message.

7 Encryption

Since we need to read in chunks of information, I would need to use calloc to allocate memory to a "block" – k * sizeof(uint8) to be precise. The first block should be dedicated to 0xFF so that we can decrypt the data later. Once that is done, we can increment the block by 1, since we already put 0xFF in the 0th block, and read in a certain amount of data. We can then use mpzImport to convert that data into numbers and encrypt it using our encryption function–which we can then print to an outfile.

8 Decryption

This would be the inverse of encryption. We calculate the same k and create the same block. We can then scan in the file and decrypt it using our decryption function. Rather than using mpzImport, we can use mpzExport to to change the data back into words. fwrite could be then used to write the data back in a file. When we use use fwrite however, we need to write stating with the first block because we don't want the 0xFF.