Rivest–Shamir–Adleman (RSA)
(ICP)

Prof. Subhashis Banerjee
Niranjan Rajesh
Veda, Chavi,Diya, Satyakin, Aditi, Esha

By: Satwik Wats
UG 24

Submission date: May 2, 2022

Introduction

Ron Rivest, Adi Shamir and Leonard Adleman in the year 1977 presented an algorithm that would ensure secure data transmission. They named it as the RSA algorithm, it is a public key cryptosystem. It uses two large prime numbers to encrypt a message and with the help of the same two number it decrypts the message. This whole situation is possible because of one big practical difficulty that we face, the factorization of product of two large prime number. If someone figures out, or factors the product of the prime number then the RSA will break down. So, how this work is in the following way: the sender and reciever both have a public and a private key. The public key is given to everyone publicly, but the private key is just known to these two. And the encoded message will only get revealed by the private key, they both have to be inverse functions.

$$Cipher = P_A(M) \text{ M is the message and } P_A \text{ is the public key}$$

$$M = S_A(Cipher) \ \ S_A \text{ is the private key and M is the message}$$

This idea of encryption and decryption involves many other basic concepts that we will cover in this report.

Developing the Algorithm

The basic idea behind developing RSA involves a certain sequence of steps. We need to have a public and a private key,

- The very first step is to find two large prime numbers, around 1024-2048 bits that is more than 300 digits.
- Next we compute the product of our two prime numbers. Let the prime numbers be p and q and their product as ϕ, so $\phi = p \times q$.

- Now our next step is to generate a public key so we select a small odd integer e that is relatively prime to $\Phi = (p-1) \times (q-1)$
- We have obtained our public key now that is $P = (e, \phi)$
- Now to generate private key we have to look for a value of d that satisfies the condition

$$ed \equiv 1 \ \ mod \ \ \Phi$$

- As both should be inverse functions,so the transformation of a message M associated with the public key $P = (e, n)$ is,

$$P(M) = \Phi^e \ \ mod \ \ n$$

The transformation of the cipher C associated with the private key $S = (d, n)$ is,

$$S(C) = C^d \ \ mod \ \ n$$

Understanding the reason behind the working of RSA

We know the RSA works, but is there rigorous mathematical proof behind its working? No, the RSA works on a very basic concept. We will build on it using few other simpler concepts first the primality testing. For testing that the number we choose are prime or not we use the Fermat's little theorem to show that the number are indeed prime.
The theorem says that if p is a prime then,

$$a^{p-1} \equiv 1 \ \ mod \ \ p \tag{1}$$

The next theorem to look at is a variant of Chinese-remainder theorem, which states that if $n_1, n_2, ..., n_k$ are pair wise relatively prime and $n = n_1 n_2, ..., n_k$ then for all integers x and a,

$$x \equiv a \ \ mod \ \ n_i \tag{2}$$

if and only if,

$$x \equiv a \ \ mod \ \ n \tag{3}$$

Now that we have two of the basic theorems, we will try to understand the working of RSA.

As we know that both of our public P, and private S key are multiplicative inverses modulo Φ then we should get

$$M^{ed} \equiv M \ \ mod \ \ n$$

Proof of this statement has been referred directly from ICP course handout.
As we know that

$$ed = 1 + km = 1 + k(p-1)(q-1)$$

Thus if $M \neq 0 \ \ mod \ \ p$ then we have,

$$M^{ed} \equiv M(M^{(p-1)k(q-1)} \ \ mod \ \ p \tag{4}$$

$$M^{ed} \equiv M(1)^{k(q-1} \ \ mod \ \ p \equiv M \ \ mod \ \ p$$

We get $m^{p-1} \equiv 1 \ \ mod \ \ p$ from equation (1)
Now if $M \equiv 0 \ \ mod \ \ p$ we have,

$$M^{ed} \equiv M \ \ mod \ \ p$$

Also,

$$M^{ed} \equiv M \ \ mod \ \ q$$

So, we get

$$M^{ed} \equiv M \ \ mod \ \ n$$

Steps for coding

We now know few properties of e and d, so we will try to utilize these and get them printed from a function in our computers. I will be using python to implement the code.

We use the concept of gcd to compute e, we can randomly pick e from a range $0 \ \ < e < \ \ \Phi$ then confirm if $gcd(e, \Phi) = 1$, if not then choose again otherwise take that as the value of e. This is known as Euclid's gcd algorithm.

We can extend this algorithm to compute d, this is called Euclid's extended algorithm. Here calculate x and y such that

$$gcd(e, \Phi) = 1 = \ \ ex + \Phi y$$

then we take d as x This is how we define values for our public and private key. From our steps we have the prime numbers and the keys now we just need to change the string values we enter as message into ASCII characters so we can encrypt it. For that we use the simple python function of ord to encrypt from strings to numbers and chr to decrypt from numbers to string.

Using my own packages for RSA

Implementing python code for RSA using my own multiplication, division, exponent, power function.
Multiplication
To make a function that can multiply is very simple, we just have to do repetitive additions. But, this algorithm will take a long time to encrypt even a 3 digit number. It is an order n^2 algorithm in terms of time complexity. To get this out we use the Karatsuba's multiplication algorithm, which reduces the number of multiplication and therefore reducing the time complexity.
We know how high school multiplication works, it is just repetitive addition with shift in places. So, to make this more efficient we reduce the multiplication. Take two numbers x and y and both of them are in some base B. So what the algorithm says is that if there exists an positive integer m less than n, then we can write the numbers as,

$$x = x_1 B^m + x_0$$

and

$$y = y_1 B^m + y_0$$

So, now the product of the numbers are

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0)$$

We assume

$$x_1 y_1 = z_2$$

$$x_1 y_0 + x_0 y_1 = z_1$$

and

$$z_0 = x_0 y_0$$

So, this gives us the function as

$$z_2 B^{2m} + z_1 B^m + z_0$$

These functions require four separate multiplication, but Karatsuba reduced them to 3 by applying just few extra addition.

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

So, now instead of initial value of z_1 where we had to multiply four times we only do it three times.

This doesn't reduce the efficiency from $O(n^2)$ to $O(n)$, but surely reduces it to $O(n^{1.58})$ which is much better than our previous high school method. Now we implement the algorithm using python.

```
#multi is my defined function that will perform karatsuba multiplication


def multi(x,y):


    n1=len(str(x))  ## determining length of x and y

    n2=len(str(y))

    if n1==1 or n2==1: #Assert: If n1 or n2 are equal to
    zero then perform simple multiplication
        return x*y

    else:
        n=max(n1,n2)//2 #We split them in half

        a=x//10**n # 10**n becomes my new base

        b=x%10**n # mod of this plus above value will return my entire function.
        ## Example: 12345= 12*10**3+345
```

```
c=y//10**n ## repeat the same steps
d=y%10**n

ac=multi(a,c) #Now we again recursively compute ac and bd then adbc

bd=multi(b,d)

adbc=multi(a+b,c+d)-ac-bd #reducing 4 multiplication to 3 multiplication

return ac*10**(2*n)+adbc*10**n+bd ## return function in its normal form
```

Modular Exponentiation function

The next important algorithm in RSA structure is the fast exponentiation algorithm. We need to calculate power mod values in order to have fast encryption and decryption. For this algorithm we use the idea of odd and even separation, here we see if the power value is even we perform simple one step multiplication and take mod of that function and half the power value.

If the power value continues to be even we repeat the steps, but if it is odd we multiply it with a carry value of $i = 0$ and take new i value as

$$i = (x \times i) \ \ mod \ \ n$$

and then reduce the value of n by 1. The algorithm takes time complexity of $O(\log_2 y)$ as the loop runs through $y/2$ every time.

```
def expo(x, y, n):

if (x == 0):
    return 0
if (y == 0):
    return 1


i = 0
if (y % 2 == 0):
    i = expo(x, y // 2, n)
    i = (i*i) % n

else:
    i = x % n
    i = (i * expo(x, y - 1,n) % n) % n
return ((i + n) % n)
```

Power function

To compute exponent function we will divide the algorithm in different steps, first we can divide the power in half for even numbers and for odd we can still divide it in half but we will have to do an extra multiplication.

$$2^{10} = (2^5)^2$$

and as,

$$2^5 = ((2^2)^2 * 2)$$

So we use the same logic in our algorithm. The algorithm takes time complexity of $O(\log_2 n)$ as the loop runs through $n/2$ every time and goes through the number .

```
    def powe(x, n):
##Computing the power of a number  with base x and power n

    if n == 0: ## if n=0 we return 1, as 20=1
        return 1


    p = powe(x, n // 2) # WE use the dividing technique

    if n & 1:    #if the number is odd we do one extra step
        return x * p * p

    #if the number is odd we continue with our even mulitplication steps
    return p * p
```

Division Algorithm

We will use a simple algorithm to determine the remainder and quotient of x/y. The function will take $T(x) = T(x \ \ div \ \ 2) + O(1); T(0) = 0$, which gives us $T(x) = O(log_2 x)$.

```
    def divide(x,y):
    if x == 0: ##if x=0 return 0
        return (0,0)

    elif y==0: ##if y=0 return 'Error'
        print('Error')


    elif x==0 and y==0: ##if both x and y are 0 return 0
        print('Error')

    #For 0¡=n¡x divide (n,y) to return (q,r) such that n=qy+r
    else:
        (q,r) = divide(x // 2, y)
        q1 = 2*q
        r1 = 2*r
        ## If x is odd then divide(x//2, y) and return r=2*q+1
        if (x%2 == 1):
            r2 = r1+1
            if (r2¡y): ##if r2 is less than y then return (2*q,2*q+1)
                return (q1,r2)
            else: #if r2¿y then return (q1+1,r2 - y = 2r + 1 - y)
                return (q1+1,r2-y)
        ##if the x is odd
```

```
        #then algorithm returns x = (q1 + 1)y + (r1 - y)
        else:
            if (r1¡y):
                return (q1,r1)
            else:
                return(q1+1,r1-y)
def rem(d) : #This function finds the mod value of (x,y)
    return d[1]
rem(divide(5,2))
```

Getting started with the code

The first step is to confirm the numbers we use are prime or not.

```
    def isPrime(a):
    if a==2:
        return True
    i=0
    while i¡5:
        p=np.random.randint(2,20)
        if expo(p,a-1,a)!=1:
            return False
        else:
            return True
        i+=1
```

After doing the primality testing, we will develop the functions for gcd, extended gcd and multiplicative inverse.
GCD

```
    def gcd(e, phi):

    while phi != 0:
        e, phi = phi, e%phi
    return e

    Time complexity: O(log N)
    Space complexity: O(1)
```

Extended GCD

```
    def euclidextalgo(a,b): #x = y1 - b/a * x1  ##y = x1
    de,re=divide(a,b) #de=div and re=mod
    if(a%b==0):
        return(b,0,1)
    else:
        g,x,t = euclidextalgo(b,re) #recursive call of  gcd(b,a%b)
        f=multi((de),t)
        x =x-f
```

```
        return(g,t,x)
        Time complexity: O(log N)
        Space complexity: O(1)
```

Multiplicative Inverse

```
    def mulinv(e,phi):
    g,x,=euclidextalgo(e,phi)
    if(g!=1):
        return None
    else:
        if(x¡0):
            (x,x,rem(divide(x,phi)))
        elif(x¿0):
            print(x)
        return rem(divide(x,phi))
```

Developing the private and public key

```
    def priv“pub(p,q):
    n=multi(p,q)
    phi=multi((p-1),(q-1))
    #We randomly generate e value and check its gcd if it is true we
    ##take it and continue further calculation ow repeat the step
    e = random.randrange(1, 1000000)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, 1000000)
        g = gcd(e, phi)
    d=mulinv(e,phi)
    return (d,n),(e,n)
```

Generating encryption and decryption algorithm

```
    def encrypt(pub“key, message):
    d, n = pub“key
    secret = [expo(ord(char), d,n) for char in message]
    return secret


def decrypt(pri“key, secret):
    e, n = pri“key
    message = [chr(expo(char,e,n)) for char in secret]

    return ’’. join(message)
```

This method of RSA using my own package reduces the efficiency of the algorithm. The major issue is using a better division algorithm to calculate mod fast. The above function can produce p and q value of 13 digits, but If we use the inbuilt mod function we can use p an q value of 200 digits each. And, we can make it much better if we use multiplication, division, and power from python and implement them then we get p and q values of more than 400 digits. This gives RSA an overall time complexity for encryption as $O(b^2)$ and for decryption $O(nb^2)$, where the public and private modulus n has b bits

Implementing RSA using python big-num package

GCD

```
def gcd(e, phi):
while phi != 0:
    e, phi = phi, e%phi
return e
```

Extended GCD

```
def euclidextalgo(a,b): #x = y1 - b/a * x1  ##y = x1
if(a%b==0):
    return(b,0,1)
else:
    gcd,x,t = euclidextalgo(b,a%b) #recursive call of  gcd(b,a%b)


    x =x-((a//b)*t)
    return(gcd,t,x)
```

Multiplicative Inverse

```
def mulinv(e,phi):
gcd,x,=euclidextalgo(e,phi)
if(gcd!=1):
    return None
else:
    if(x¡0):
        (x,x,x%phi)
    elif(x¿0):
        print(x)
    return x%phi
```

Developing private and public key

```
def privpub(p,q):
n=(p*q)
phi=((p-1)*(q-1))
e = random.randrange(1, phi)

g = gcd(e, phi)
while g != 1:
    e = random.randrange(1, phi)
```

```
        g = gcd(e, phi)
    d=mulinv(e,phi)
    return (d,n),(e,n)
```

Generating encryption and decryption algorithm

```
    def encrypt(pubkey, message):
    d, n = pubkey
    secret = [pow(ord(char), d,n) for char in message]
    return secret


def decrypt(prikey, secret):
    e, n = prikey
    message = [chr(pow(char,e,n)) for char in secret]

    return ''. join(message)
```

Snippet of the code

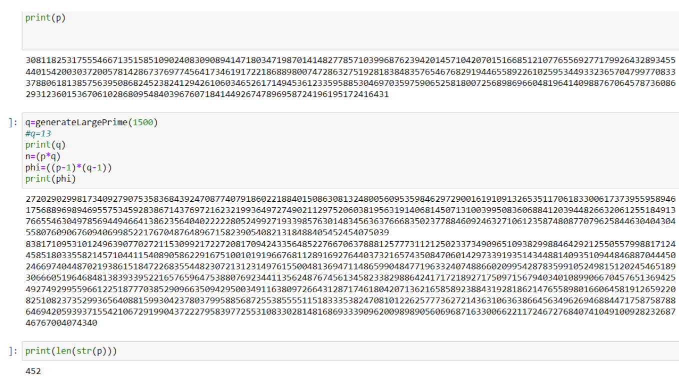

Figure 1: Generating very large prime number around 400 digits.

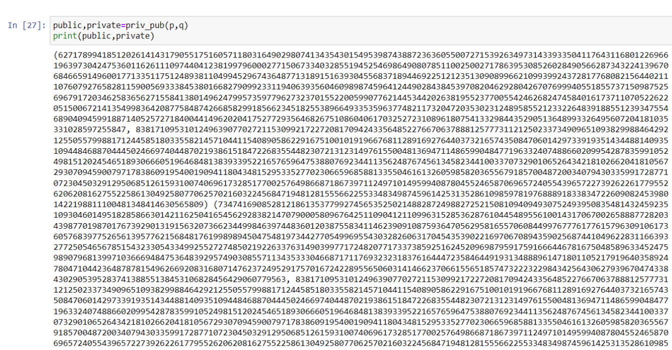

Figure 2: Generating very large public and private keys for encrypting and decrypting.

In [13]:
```
msg='Is this okay?' ##Enter your message!!
value=encrypt(public,msg)
print(value)
```

```
[116874290779447766230147, 60086293322882751421929, 144681021746757688711077, 29907140276024712988260, 24267340204038441297205,
97357695785429658837675, 60086293322882751421929, 144681021746757688711077, 132054475280072536111682, 96787653506413354936773,
148539308807553617047051, 70578380251355897928934, 11983186138701784980973]
```

In [14]:
```
f=(input('Do you wanna decrypt your message? If yes type "yes" otherwise "no": '))
if f=='yes':
    print(decrypt(private,value))
elif f=='no':
    print('Your loss!!')
else:
    print('Wrong option bud!')
```

```
Do you wanna decrypt your message? If yes type "yes" otherwise "no": yes
Is this okay?
```

Figure 3: Encrypting and decrypting a message using 13 digit prime number RSA code.

References

- https://en.wikipedia.org/wiki/Fermat_primality_test: :text=adds
- https://www.codespeedy.com/fast-exponentiation-python/
- https://pythonandr.com/2015/10/13/karatsuba-multiplication-algorithm-python-code/
- https://www.codespeedy.com/modular-exponentiation-in-cpp-power-in-modular/
- https://en.wikipedia.org/wiki/RSA_(cryptosystem)
- https://www.rsa.com/
- FinalProject_CS:1101 handout
- https://crypto.stackexchange.com/questions/6164/how-do-i-derive-the-time-complexity-of-encryption-and-decryption-based-on-modula/61946194
- https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
- https://cp-algorithms.com/algebra/module-inverse.html