**Overview and Goal**

This section of the report aims at explaining the mechanics of the algorithm in simplest terms. We will mostly discuss key generation and the tools that we need to make this cryptosystem work. To provide examples, we will be using a combination of mathematical algorithms taken from the book, pseudocode, and examples that aim at describing how these algorithms might be implemented in a program. I aim to capture the spirit of these mathematical ideas, that we have been discussing, in the implementation. I will provide a structure that reviews the examples and theory behind the mechanics of the RSA cryptosystem alongside the pseudocode. We will first discuss encryption and decryption to have an idea of how the RSA algorithm works in practice. Then, we will discuss the more complex aspect of key generation.

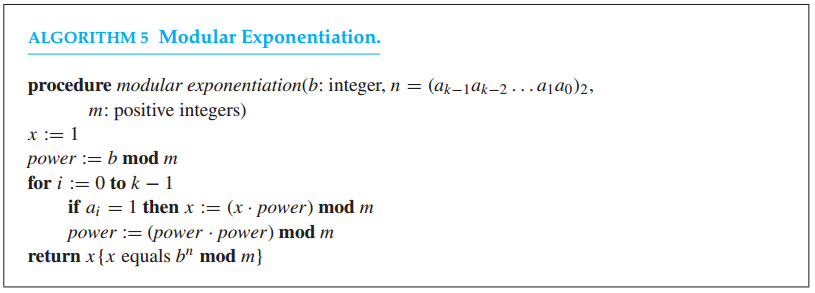
First of all, RSA is known as an asymmetric cryptosystem referencing the way that one set of keys are made public and another are kept private. These keys consist of an ordered pair of integers (a,b). For encrypting, let’s call these numbers “(e,n)” and for decrypting, “(d,n)”. Notice how the second number is the same. We will discuss why this is the case in short, along with how we determine all of these numbers, but for now, we turn our attention to the process of encryption.

**On Modular Exponentiation**

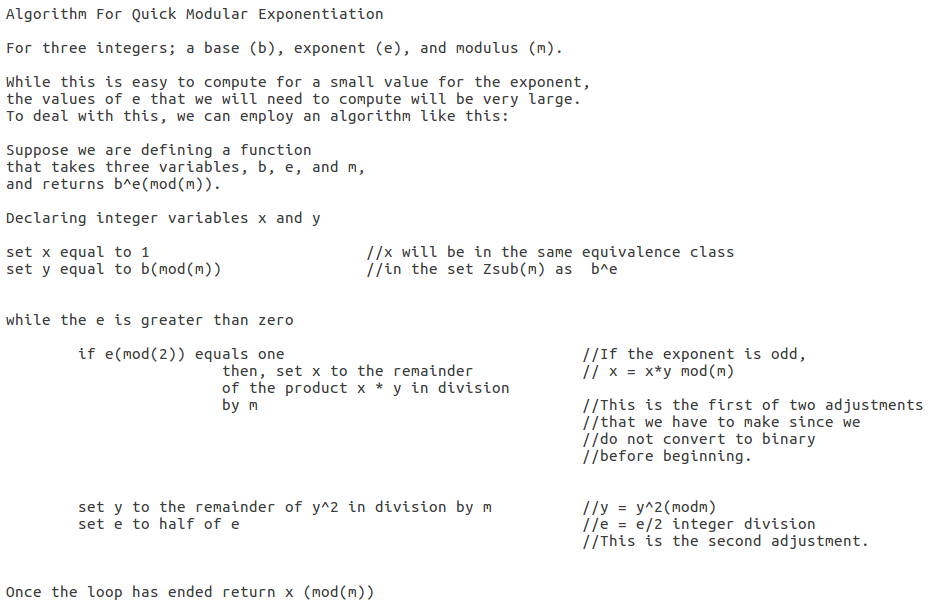
Key to encrypting and decrypting in the RSA cryptosystem is the process of modular exponentiation, or computing the remainder of some base raised to an exponent in division by some integer. In practice, we see that this is a straightforward, simple process when the base and exponent are small numbers:

Given, base = 4 exponent = 5 modulus = 3, we know that and 1024 has remainder 1 in division by 3. So, . However, small numbers like these are not what we work with in this cryptosystem. How do you compute ?

The book\*\*\* gives us an algorithm that lets us deal with this:

****

First, we convert b = 4 into base 2. 4 = . Since the first digit has a “1” and it’s the only “1”, we can quickly compute, . Here is the pseudocode for this algorithm with two small adjustments so that we do not convert to base 2.



**Encryption**

Given public encryption keys, (e, n), we can encrypt a message to a recipient using modular exponentiation. As an example, we will encode the message, “Hello World”. First, we need an integer representation of the characters such as is provided in ASCII or Unicode. In Unicode, the characters that appear in the string, “Hello World” have the following values:

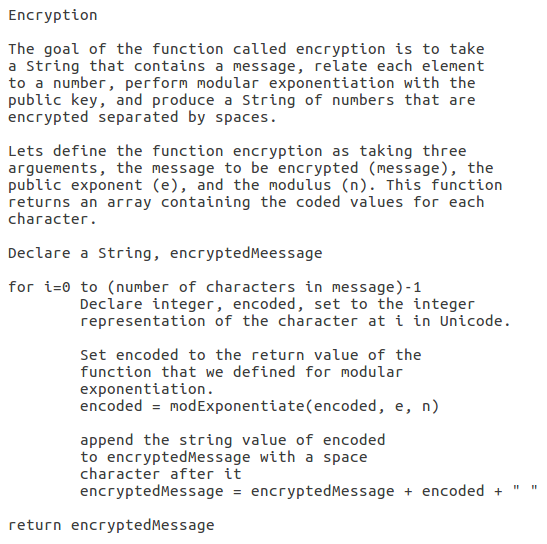
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H | e | l | l | o |  | W | o | r | l | d |
| 72 | 101 | 108 | 108 | 111 | 32 | 87 | 111 | 114 | 108 | 100 |

To encrypt a single-letter message, M, for the person whose public encryption key is (e,n) one would compute: . So for our message, we perform this operation on each of the characters in the string. For this example, let e = 139 and n = 143, then:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| H | e | l | l | o |  |
|  |  |  |  |  |  |
| 123 | 127 | 82 | 82 | 45 | 98 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| W | o | r | l | d |
|  |  |  |  |  |
| 87 | 45 | 36 | 82 | 100 |

Hence, our encrypted string would read “123 127 82 82 45 98 87 45 36 82 100”.



**Decryption**

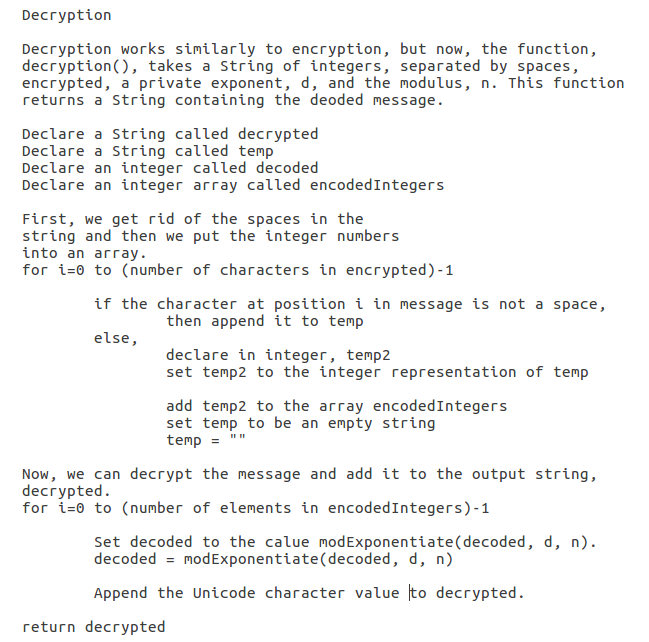
Decryption works similarly to encryption. If presented with a character, we find its numerical representative. Then we use the decryption keys (d,n) and the same process of modular exponentiation as is used for encrypting. This time, we take in the string, “123 127 82 82 45 98 87 45 36 82 100”.

Given the decryption key (19,143):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 123 | 127 | 82 | 82 | 45 | 98 |
|  |  |  |  |  |  |
| 72 | 101 | 108 | 108 | 111 | 32 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 87 | 45 | 36 | 82 | 100 |
|  |  |  |  |  |
| 87 | 111 | 114 | 108 | 100 |

Looking up these values in the Unicode tables, we find that we are back to the original message:



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H | e | l | l | o |  | W | o | r | l | d |
| 72 | 101 | 108 | 108 | 111 | 32 | 87 | 111 | 114 | 108 | 100 |

“Hello World”

**Key Generation**

As discussed above, RSA is an asymmetric cryptosystem. So one of the keys will be publicly shared and the other kept private. Such that one could send a message to anyone using their public key, (e,n) to encrypt the message and they can use their private key, (d,n) to decrypt the message. Here, we will discuss how the numbers d, e, and n are generated. However, in order to learn about e, d, and n, we must also talk about the two prime numbers, p and q, Euler’s Totient function, , and their role in the process.

We begin with two large prime numbers. Let’s call them p and q. The number n that we have seen as the modulus for encryption and decryption, is simply the product of p and q.

As the book, Discrete Mathematics, mentions on page 299, these prime numbers are oftentimes 200 digits long. It is here where the strength of the cryptosystem can be found. For such large prime numbers p and q, it is estimated to take a very long time to factor their product, given that there is no known easy way to factor very large numbers into primes.

In the case of our example, p is 11 and q is 13. Now we can calculate n.

I would like to point out the trivial restriction that n must be greater than or equal to the highest integer character representation. For example, if we are using Unicode decimal encoding, n >= 122 would ensure that we could encode and decode characters: ABCD…abcd…z. I designate it trivial as we are trying to work with large values for p and q, not the small ones that would cause this sort of trouble.

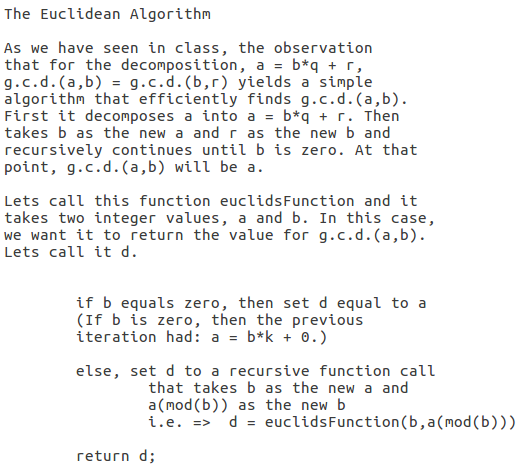
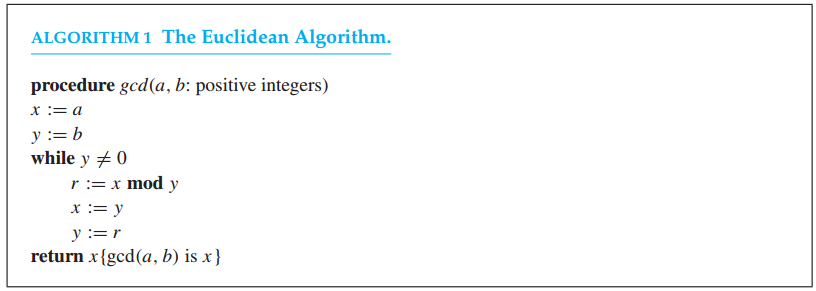
Euler’s Totient function, , is a function that returns the number of integers that are less than or equal to n and do not share any common factors with n. That is, the number of integers less than n that are coprime with n. If we look at any prime number, we see that it has no common factors with any number less than itself. Therefore, if n is prime.

Noticing that if p and q are prime:

In the case of our example,

At this point, most implementations of the algorithm proceed to choose a value for e. This can be done by storing a list of candidates and just confirming that the selected candidate is coprime with .

and in the range: 1 < e < n. Our implementation uses the Euclidean Algorithm to select the largest value for e.



Following our algorithm to get e,

(142, 120) are multiples of 2

(141,120) are multiples of 3

(140,120) are multiples of 2

(139,120) are coprime

So that is why the value for our encryption exponent is 139.

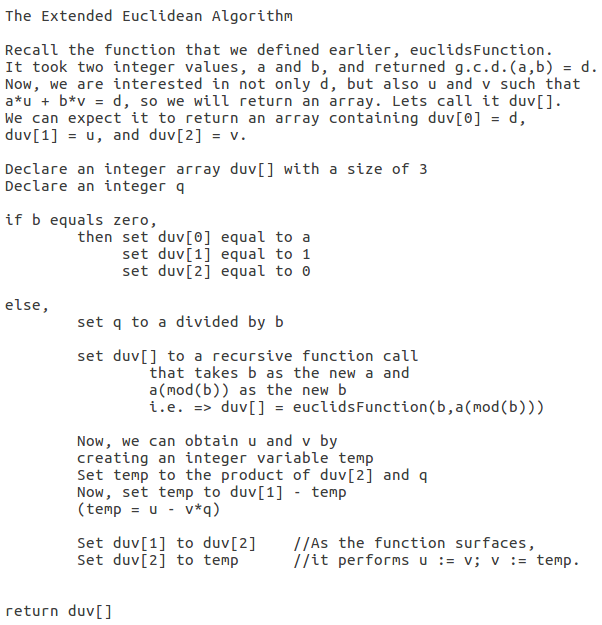
It is important that e be coprime with because later we want to pick d such that:

e will not have an inverse if it is not coprime with .

Now, how do we choose d such that ?

Well we chose e to be coprime with and that the Bezout Identity tells us that for any two integers (a and b), not both zero, there exist two integers (u and v) such that . Where d is a common divisor of a and b. In lowest positive form, d = g.c.d.(a,b).

Therefore, we know that is the lowest possible positive way of expressing this equation since g.c.d.(e,) = 1. In particular, we see that u = d. So, if we have a function that performs the extended Euclidean algorithm on two integers, a and b, and returns not only the greatest common divisor, but also integers u and v such that , then we can compute d.



Computing d for our example:

e = 139,

139 = 1 =

120 = 1 =

19 = 1 =

6 = 1 =

1 = Hence, d = 19.

With everything that we have discussed in place, generating the keys becomes simple:

