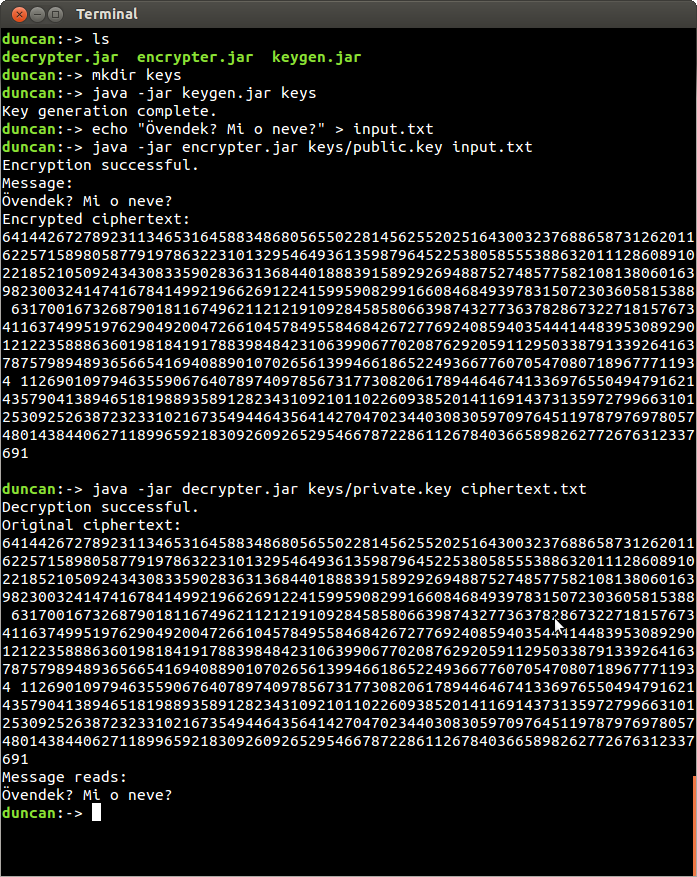
RSA Encryption Implementation

My solution consists of three executable .jar files:

* keygen.jar: generates a private and public key pair. It takes a single command line argument that specifies a path to the directory into which to write the public and private keys. This path is relative to the location of the jar. If no argument is supplied it writes the files to the same location as the jar. The files written are ‘public.key’ and ‘private.key’.
* encrypter.jar: takes an input file and encrypts it using a given public key. It writes the result as ‘ciphertext.txt’ to the same location as the jar file. It takes two arguments specifying where the key is and where the input file is. If no arguments are supplied the program looks for files called ‘public.key’ and ‘input.txt’ in the same location as the jar.
* decrypter.jar: takes a cipher text file and a private key file and prints the decrypted message and ciphertext to the screen. It also writes the decrypted message to a file called ‘output.txt’. It takes two arguments specifying where the private key is and where the ciphertext is. These default to looking for files called ‘ciphertext.txt’ and ‘private.key’ in the same location as the jar.

The operation of the three files from a Linux terminal is documented below:



The .jar files, keys, input and output can be accessed in the ‘final’ directory of this submission.

The classes that make up the system model reflect five responsibilities:

* key generation
* decryption
* encryption
* string parsing
* file handling

My first attempt at the problem was to understand the key generation algorithm I had been asked to implement. Work began with a group discussion with several fellow students in which we worked out the steps the key generator would have to complete to produce the keys:

Set e to 65337

Are e and phi coprime?

Find phi

phi = (p-1)(q-1)

Yes

No

Generate p and q

Multiply p and q to get n

Find d using Euclidean Algorithm

This inventory of the steps involved suggested I create a set of functions, each individually testable, which would combine to produce the keys.

I completed this function-writing portion of my work in a test-driven manner. I began by using the examples we had been given to extract values for p and q as input, along with what values were expected for phi, n, d and e as output. I decided that all my functions would handle the integer primitive type rather than BigInteger objects, and that I would translate the algorithms to use BigIntegers later. This helped me keep my code more readable and intuitive while I was trying to get the algorithm to pass the tests.

It was agreed during our discussion that the easiest function to start with seemed to be Euclid’s algorithm, since we were given a recursive implementation in the UCS lecture notes. I translated Euclid’s algorithm gcd() function into Java, and ran my tests against it until my code was producing the output given in the examples.

**public** **static** BigInteger gcd(BigInteger a, BigInteger b) {

/\*\*

\* Returns the greatest common divisor

\* between two numbers a and b.

\* Uses a recursive version of the

\* Euclidean Algorithm.

\*/

// base case

**if** (b.equals(BigInteger.*ZERO*)) {

**return** a;

} **else** {

// tail

**return** *gcd*(b, a.mod(b));

}

}

It was agreed in our discussion that the hardest part of the process would be the implementation of the Extended Euclid algorithm, because it involved recursion and returning multiple values. The Extended Euclid function is another recursive function, which given e and phi, produces, amongst other values, the multiplicative inverse of e. This becomes d, one part of the private key. I found a wikibooks article that described the algorithm and provided some example code in Python. I translated this implementation, along with the algorithm supplied in the UCS lecture notes, into Java. I also added the section which translates the multiplicative inverse into d, which is shown below:

When writing this function I hit one of the main problems I encountered during this assignment: returning multiple values from a function in Java. The Extended Euclid function returns three values, which must be passed to the next stage of the recursion. There are two options available: return the values in a collection object, such an Array or HashMap, or create a dedicated class to express a value object made up of those values. In this case I wrote each value into an array and returned the array, then unpacked the values into separate variables. This seemed less cumbersome than creating a class just for returning Extended Euclid results, since all the values were primitive integers. However it is a less readable approach since the variables get put into anonymous positions in an array, rather than being given human-readable labels.

**private** **static** BigInteger[] extendendEuclid(BigInteger a, BigInteger b) {

/\*\*

\* A recursive function that, given a and b,

\* calculates d, s and t from the equation:

\* ax + by = g = gcd(a, b)

\* Adapted from UCS lecture powerpoint

\*/

// base case

**if** (a.equals(BigInteger.*ZERO*)) {

BigInteger[] result = { b, BigInteger.*ZERO*, BigInteger.*ONE* };

**return** result;

}

BigInteger[] result1 = *extendendEuclid*(b.mod(a), a);

// unpacking result array to make code more readable

BigInteger g = result1[0];

BigInteger y = result1[1];

BigInteger x = result1[2];

// packing next stage of recursion into an array,

// ready to be returned

BigInteger[] result = { g, x.subtract(b.divide(a).multiply(y)), y };

**return** result;

}

Once I had completed these functions I immediately started adding Javadoc comments to the top of my functions. These supplemented my function names, so I would remember later what they did, and I also kept a record of the resources I used to arrive at these functions.

I wanted to be sure I had the correct key generation process in place. To do this I needed an integration test, but before I could write this I needed to write some string manipulation functions that would translate characters into integers. This stage was the first time I started using the BigInteger object, since the large size of the integers produced would necessitate using this data type. Again I wrote the tests first, looking up a Unicode character table to determine what numbers should be produced given short input strings:

<Unicode tests>

I now had two functions:

BigInteger stringToUnicodeNumbers(String string)

In Java, casting a char primitive to an int produces the Unicode character code as an integer. stringToUnicodeNumbers()uses this, and adds extra zeros to each integer so that each character is represented by a five digit integer.

String numberToString(BigInteger b)

The only tricky part of numberToString() was remembering to pad with enough leading zeros so that the integer splits separately into five digit chunks. These integer chunks can then be cast back into chars.

At this point I refactored my key generation functions to use the BigInteger type. On reflection I am glad I went through this extra stage since using primitive ints produced much more readable code. For example, this code forms the tail of the Extended Euclid algorithm:

<extended Euclid line as ints>

Compared with the same line written in terms of BigInteger methods:

<extended Euclid line as BigIntegers>

I wrote an integration test that used all my functions to generate a key, encrypt a short message and decrypt it back. The actual encryption and decryption was done using the built-in BigInteger .modPow() function. The test passes if the decrypted message matched the original message.

As well as confirming my code worked as expected, this test highlighted a limitation with the RSA algorithm. The algorithm will only map a finite field of numbers to another finite ciphertext field, because it uses modular arithmetic. Numbers higher than n will wrap around and be mistaken for another number in the field. I mitigated this problem by increasing the bit length of p and q, which increases n and allows longer integers, and so more characters, to be represented. This also increases the computation time, so I added a splitMessage() function which split the message into seven character chunks. Each chunk was encrypted separately, allowing me to use smaller values for p and q, yet still encrypt longer messages.

<splitMessage>

Up to this point I had approached the problem in a procedural style, with each step given its own function. I had a single Encrypter class with a lot of functions on it, plus some higher level functions that called these step functions to complete the process:

encrypt()

pickPQandPhi()

getD() (Extended Euclid)

splitMessage()

stringToUnicodeNumbers()

decrypt()

partsToBigInts()

decryptArray()

numberToString()

The only higher-level structure that had emerged from my work so far was a value class called encryptedMessagePair. This stored the message, plus d and n, and would be passed from the encrypt process to the decrypt process so that the private key was preserved along with the message. This was far from ideal as in practice the private key was to be stored separately to the message, and I had no provision for storing the public key. Now I had the algorithm implemented and that implementation verified, a refactoring stage was needed.

The worst form of refactoring is one without any stated purpose. I learnt this as my first attempt was to create a Message class, which stored all the key generation values, plus the private key and the public key all in one object. After another look at the problem brief, and a high-level design discussion with a work colleague, I realised that this did not really solve the problem, and that I should split my functions into classes based on the principle of single responsibility.

I looked again at the problem brief and split the functions across the three main responsibilities of the system: key generation, decryption and encryption. I noticed that supporting these domain responsibilities were two other responsibilities: string parsing and file handling. Once these responsibilities had been separated from the domain problem of encryption and decryption, the code became more modular. The string translation functions are arranged in terms of arrays of BigIntegers and arrays of Strings, without any knowledge of keys or other domain concepts. Similarly, the encryption and decryption functions are expressed in terms of BigInteger arrays, and are not concerned with how the information will finally be presented at all.

Refactoring: Classes for each responsibility

KeyGenerator:

Generates a PublicKey and PrivateKey object.

generate()

Encrypter:

Given a PublicKey object, converts a BigInteger array to an encrypted BigInteger array.

Decrypter:

Given a PrivateKey object, converts a cipher BigInteger array to a decrypted BigInteger array.

StringParser Support Class:

For encryption:

Converts a String to a BigInteger array using Unicode values

Converts a BigInteger array into a String by concatenating the integers with spaces.

For decryption:

Converts a ciphertext String into a BigInteger array

Converts a BigInteger array into a string using Unicode values

File Handler Support Class:

Writes a string to a file, replacing the contents

Read a string from a file, handling new lines sensibly

This refactoring yielded three value classes for passing values around the system: The first of these was InitialValues, which encapsulates the fact that the key generator has to pass p, q, n and phi around its functions. This became an inner class of the KeyGenerator class, since it is not needed outside the key generation processes. The other two value objects needed where PrivateKey and PublicKey, which allowed me to label d, n and e sensibly. The Key classes also featured overloaded constructors, which encapsulated the fact that the keys are created by the KeyGenerator from computed values, but are also created from a string by the Decrypter and Encrypter classes.

I also had to refactor my integration tests, which uncovered some implementation bugs, particularly in the StringParser class. This was because the code for string manipulation could now be tested separately, where previously it had been an extra process hanging on to the encrypt() and decrypt() functions.

For example, splitMessage() went through several implementations, as my first attempt was very buggy. It had problems handling zero-length strings and strings that only took up one chunk, producing String arrays with extra null elements, which would then trip up subsequent functions. Because the functions had been refactored out into separate concerns, the problem was easily isolated and unit tests then helped me solve the issue quickly.

Finally, I created the three command line tools that form my complete RSA implementation. This proved to be fairly easy after the refactor, since each process could be expressed in terms of the functions and classes I had created.

The group discussion helped me to understand the stages of the algorithm, and discussion with a more experience developer helped add purpose to the refactoring process. Test Driven Development helped immensely when implementing the algorithm, and helped me uncover bugs in my implementation later in the development process. Each stage could be built up with confidence as the previous stage had proven to be producing the right output.

Wikibooks article: http://en.wikibooks.org/wiki/Algorithm\_Implementation/Mathematics/Extended\_Euclidean\_algorithm

Article concerning finding e: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf

How did I get to this design?

Group discussion

How to understand the key generation algorithm:

1. Generate p and q, then find n and phi.
2. Set e to a fixed, relatively small prime.
3. Keep trying different values for p and q such that phi and e are coprime.
4. Use the recursive Extended Euclid algorithm to find the greatest common divisor.
5. Combines part of the gcd output with phi to get d.
6. Publishes n and e as the public key.
7. Keeps n and d as the private key.

Test driven development

Used the worked examples given to us as expected output.

Particularly challenging to get the right value for d – ended up having to translate an implementation written in python, plus the algorithm written in the lecture slides, to arrive at an algorithm.

Produced single class that got the keys and did the encryption on a given string.

Unexpected Limitations

Discovered that only values up to a certain number may be encrypted accurately. The maximum is dependent on n, because the encryption/decryption processes involve mod n. Any larger value just wraps round. To mitigate this the program splits the message into separate chunks, which are encrypted and decrypted separately from each other. This allows me to be in control of the maximum number that might be sent into the encryption/decryption process, and so avoid wrap-around errors.

Extensive refactoring into classes

First had a Message value object being passed around

Refactored into a class for each responsibility within the overall process.

1. I began by writing the actions the program must perform:
   1. Key generation
   2. Encryption
   3. Decryption
2. I wrote out each of the stages within each action in more detail. For example, KeyGenerator must:
   1. Generate p, q and n, with the correct relationship between phi and e
   2. Find d.
   3. Write the keys to a file.

The encryption process:

Read in the public key:

e n

65537 112261115257780114381657619942310817957

Read in a message: ‘abcdef’

Split message into parts, for example 3 character parts: [‘abc’, ‘def’]

‘a’ ‘b’ ‘c’ ‘d’ ‘e’ ‘f’

Translate into BigIntegers using the characters’ Unicode values: [**97**00098**00099**, 100**00101**00102]

Encrypt these integers:

[31495632388743691742356620824474429649, 77990186094313540699241866795966817542]

Write to a file.

The decryption process:

Read in the private key:

d n

108403568657225911123489228638924770753 112261115257780114381657619942310817957

Read in the cipher text:

“31495632388743691742356620824474429649 77990186094313540699241866795966817542”

Split into parts separated by whitespace:

[31495632388743691742356620824474429649, 77990186094313540699241866795966817542]

Decrypt these integers:

[970009800099, 1000010100102]

Translate each section into Unicode characters (remembering to add zeros to the start before splitting)

[‘abc’, ‘def’]

Join the parts together and print the result to the screen:

‘abcdef’

Process (according to github):

|  |  |  |
| --- | --- | --- |
| 849373d2 | April 7th | First dummy test |
| 8dc361bb | April 11th | Function to find d complete following group session.  Tried several d-finding functions including an iterative one.  Everything else fairly easy.  All code uses integers. |
| bcff10d1 | April 15th | First test for stringToUnicodeNumbers() |
| eec18337 | April 16th | Completed first draft of stringToUnicodeNumbers()  Used Unicode conversion from:  http://stackoverflow.com/questions/2220366/get-unicode-value-of-a-character |
| a54273ee | May 8th | Completed numberToString(), which converts a BigInteger into a Unicode string. |
| 01d06458 | May 9th | Converted encryption functions to use BigInteger.  Wrote first integration test which checked that string parsing and encryption/decryption worked together. |
| 44c018f8 | May 9th | Added function to split string into smaller strings to speed up encryption/decryption time.  Encrypt and keygen currently done by one large function. Decrypt the same, taking d and n as arguments.  Introduced encryptedMessagePair class to move d and n (the private key) around alongside the encrypted message.  Introduced loop during key generation process to make sure e is coprime with phi. |
| 600b5da0 | May 9th | Introduced a value object Message, which has p, q, e, phi, n and d along with the message. |
| d37e9d30 | May 14th | Set encoding properly in source code. |
| 8e8611ad | May 23rd | Pulled string parsing methods into their own classes.  KeyGenerator gets its own class.  InitialValues class replaced Message and EncryptedMessagePair for KeyGenerator.  Concept of a private key class and a public key class. |
| b8b0559a | May 23rd | File class added.  Other classes refactored into StringParser, KeyGenerator, SimpleFile, Decrypter and Encrypter.  StringParser takes an argument in its constructor to specify how big each chunk fed to the encryption/decryption process will be.  Added tests for top level StringParser functions. |
| 966c592e | May 23rd | Added new integration test for entire file operations to string parsing to encryption/decryption process.  PublicKey and PrivateKey handle conversion to and from a string for public and private keys. |
| 1ab99465 | May 24th | Added guards against empty strings at the parser level.  Updated tests to match – this bug was originally found using the integration tests |
| 32ad02c5 | May 24th | Created a class for each procedure that calls the other classes. Each takes arguments from the command line. Output final jar files. |

Refactoring Stage 3: move StringParser into Encrypter/Decrypter?

Emergent Classes

An initial value object emerged that is used inside the KeyGenerator to pass values (p, q, n, etc.) from function to function.

Classes for the public and private key allowed the keys to be passed around and handled converting to and from strings.

How the private key is found

How e is found

How d is found

How the prime numbers are found