{EPITECH}

MY_PGP_

< A SONG OF CIPHERS AND PRIMES />

MY_PGP

binary name: my_pgp
language: everything working on ”the dump”
compilation: when necessary, via Makefile, including re, clean and fclean rules

- ✓ The totality of your source files, except all useless files (binary, temp files, objfiles,...), must be included in your delivery.
- ✓ All the bonus files (including a potential specific Makefile) should be in a directory named bonus.
- ✓ Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Alice and Bob want to exchange data without Eve to notice and get access to the messages.
Hold up, wait a minute. Something ain't right! Let's start this all over again...

Arya Stark is in a secret mission in the free city of Pentos. She has gathered vital information that she must transmit to Brienne of Tarth back in Westeros. Alas, Euron Greyjoy and his fleet of Ironborn roams the Narrow Sea and they search from top to bottom all the ships crossing the border. If Arya tries to send her message this way, her precious information will be known by the Evil Queen in King's Landing in no time.

Fortunately, the maesters of Pentos have kept alive the old Valyrian art of concealing messages, known as cryptography. The knowledge is all there, but they need you to actually implement these ancient algorithms and finally bring peace to the Seven Kingdoms.

> — What do we say to the God of Death?
> — 4e6f7420746f6461792e

Nomenclature

- ✓ There are two main ways to represent data: it is either a sequence of bytes, or a number.
- ✓ Numbers will always be represented as a **hexadecimal value** in little endian, that is, the lower byte is on the lower address (on the left).
  For instance, the 32-bit integer with the hexadecimal value 0x1234567 is represented in little endian as ”67452301”.
- ✓ Be careful: numbers must also be represented in little endian when displaying the result!
- ✓ The data to be ciphered is always a sequence of bytes.
- ✓ What is a number and what is a sequence of bytes will be explained for each algorithm.

Usage

Terminal

```
$>  ./my_pgp -h
USAGE
    ./my_pgp CRYPTO_SYSTEM MODE [OPTIONS] [key]

DESCRIPTION
Cipher or decipher MESSAGE using a given CRYPTO_SYSTEM. The MESSAGE is read from the
standard input.

  CRYPTO_SYSTEM
    "xor"        computation using XOR algorithm
    "aes"        computation using 128-bit AES algorithm
    "rsa"        computation using RSA algorithm
    "pgp-xor"    computation using both RSA and XOR algorithm
    "pgp-aes"    computation using both RSA and 128-bit AES algorithm

  MODE
    -c           MESSAGE is clear and we want to cipher it
    -d           MESSAGE is ciphered and we want to decipher it
    -g P Q       for RSA only: Don't read a MESSAGE, but instead generate a public and
private key pair from the prime number P and Q

  OPTIONS
    -b           for XOR, AES and PGP, only works on one block. The MESSAGE and the
symmetric key must be the same size

  key        Key used to cipher/decipher MESSAGE (incompatible with -g MODE)
```

To improve debugging, all your symmetric algorithms **MUST** accept the option ”-b”, for **block mode**: only one block (same size as the key) of the message will be treated.

In block mode, all the data to be ciphered/deciphered will always be on a single line!
The potential trailing line feed is not considered as part of the message.

Without the ”-b” modifier, your algorithm must work in **stream mode**: the message to cipher/decipher can be of any length!

Symmetric encryption

Symmetric encryption algorithms use the same key for ciphering and deciphering. You have to implement two of these algorithms: one is very simple but either impractical or easy to crack, the other one is more robust and practical (but harder to implement).

Simple XOR

To hide the message, Arya proposes to XOR the message with some random data called key.

The key and the ciphered output are both hexadecimal **numbers**.

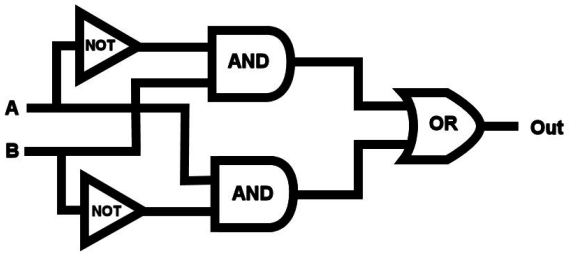
You have to code a program that takes a **message** *m* and a **key** *k*, and returns $c = m \oplus k$.

Terminal

```
$>  echo "You know nothing, Jon Snow" > message
$> ./my_pgp xor -c -b 576861742069732064656164206d6179206e6576657220646965 < message >
ciphered
$> cat -e ciphered
20070f2700071c6a4449060a490515164e4e12190b190011063c$
$> ./my_pgp xor -d -b 576861742069732064656164206d6179206e6576657220646965 < ciphered | cat
-e
You know nothing, Jon Snow$
```

In stream mode, you must encrypt/decrypt data by chunks of blocks, each block being the same size as the KEY. If the last block is smaller than the KEY, you must pad it with zeros.

A simple XOR can be a good cryptosystem, but only if the key is as long as the message and is used only once, which is not very practical.



AES

Arya could use XOR to encrypt her message, but she would need a key that's as long as the message AND a method to secretly share the key with Brienne. If she could securely send a long key, she might as well send the message itself that way. So, they need a better encryption method that uses a shorter key and is more secure than XOR.

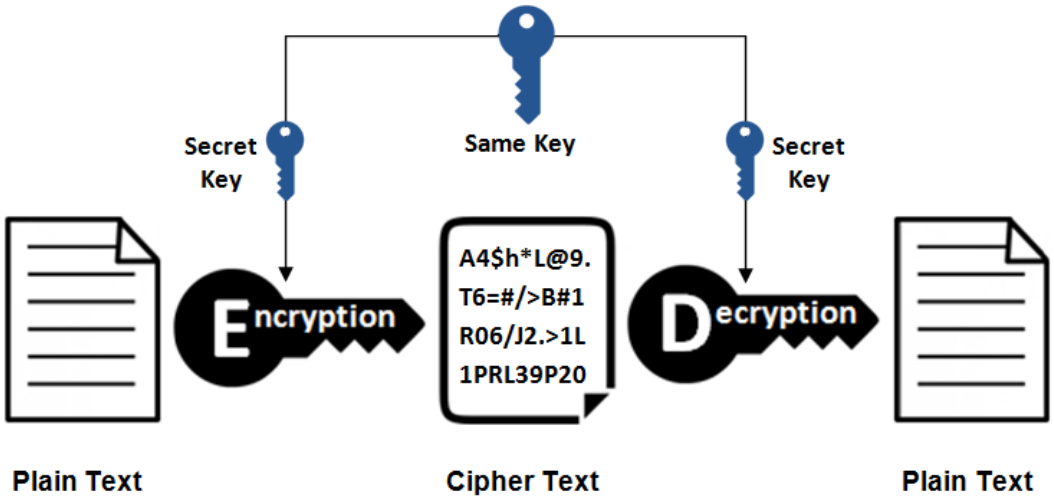
She chooses to use the AES algorithm, with a 128-bit key.

Both the 128-bit key and the ciphered message are four 32-bit numbers.

Here is an example in block mode:

```
Terminal
$>  echo "All men must die" > message
$> ./my_pgp aes -c -b 57696e74657220697320636f6d696e67 < message > ciphered
$> cat -e ciphered
744ce22c385958348f0df26eceb62eef$
$> ./my_pgp aes -d -b 57696e74657220697320636f6d696e67 < ciphered | cat -e
All men must die$
```

In stream mode, the data will be ciphered block by block.
If the last block is smaller than 128 bits, it will be padded with zeros

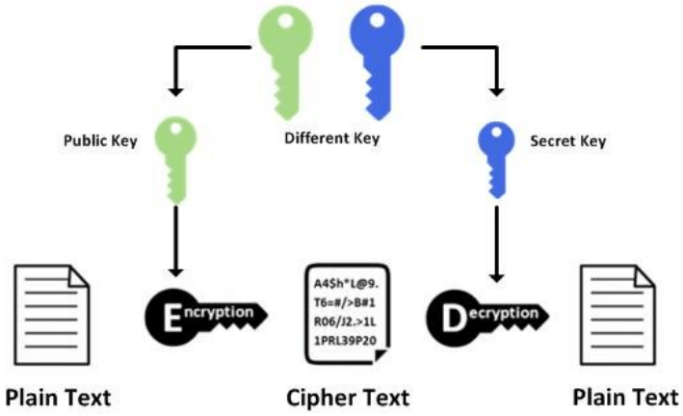


Asymmetric Cryptographic Algorithm

That's great! Arya and Brienne now have a secure way to exchange messages. However, they still need a shared secret key. If one of them selects a key and sends it by boat, Euron will likely intercept it, allowing him to read Arya's secret message when she sends it.

Therefore, they need another method to exchange their key without a prior shared secret, and without Euron being able to interfere.

Here comes to the rescue the asymmetric cryptography, which will allow anybody to cipher a text with Brienne's public key, which only Brienne can decipher using her private key!



RSA

To fully implement the RSA cryptosystem, your program **MUST**:

- ✓ generate a public and a private key, given two prime numbers ;
- ✓ cipher / decipher a message using these keys.

Your public exponent (generally called **e**) **MUST** be chosen as being the biggest Fermat prime that satisfies the algorithm requirements. You **MUST** use Carmichael's totient function to compute both the public and private key exponents (generally called **e** and **d**).

Your program may assume that the numbers given with -g are prime without further checking.
Due to the way the RSA algorithm works, you can always assume the data than needs to be ciphered/deciphered to be read on a single line, no matter if the flag -b is set or not.

Every component of both the public and the private keys are numbers.
The ciphered text is also a number.
The bytes of the message to be ciphered must be concatenated to make a little-endian number (for example: ”WF” is 0x5746, which is the decimal number **18,007**)

Here is an example with small keys:

Terminal

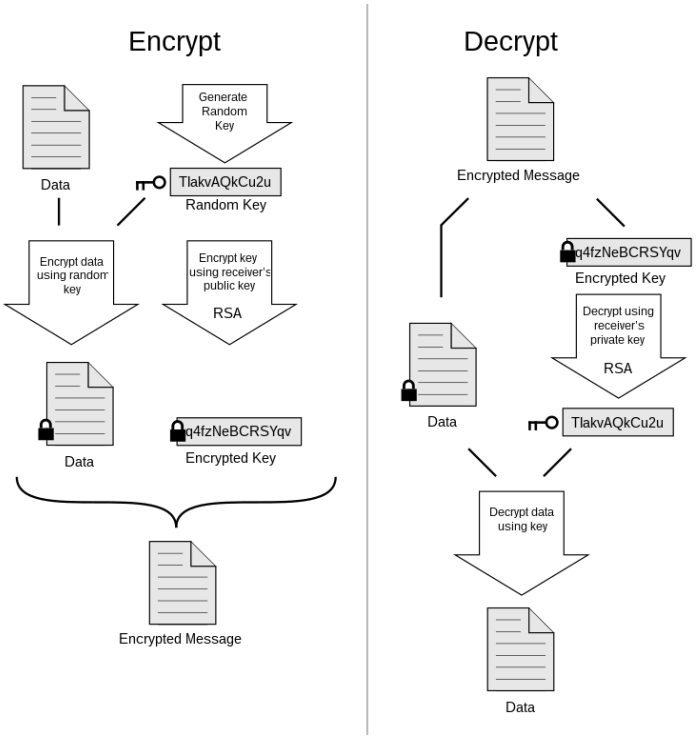
```
$>  ./my_pgp rsa -g d3 e3 | cat -e
public key: 0101-19bb$
private key: 9d5b-19bb$
$> echo "WF" | ./my_pgp rsa -c 0101-19bb | cat -e
8f84$
$> echo "8f84" | ./my_pgp rsa -d 9d5b-19bb | cat -e
WF$
```

And here is another one with larger keys:

Terminal

```
$>  ./my_pgp rsa -g 4b1da73924978f2e9c1f04170e46820d648edbee12ccf4d4462af89b\
080c86e1 bb3ca1e126f7c8751bd81bc8daa226494efb3d128f72ed9f6cacbe96e14166cb
public key: 010001-c9f91a9ff3bd6d84005b9cc8448296330bd23480f8cf8b36fd4edd0a8cd925de13\
9a0076b962f4d57f50d6f9e64e7c41587784488f923dd60136c763fd602fb3
private key: 81b08f4eb6dd8a4dd21728e5194dfc4e349829c9991c8b5e44b31e6ceee1e56a11d66ef2\
3389be92ef7a4178470693f509c90b86d4a1e1831056ca0757f3e209-c9f91a9ff3bd6d84005b9cc84482\
96330bd23480f8cf8b36fd4edd0a8cd925de139a0076b962f4d57f50d6f9e64e7c41587784488f923dd60\
136c763fd602fb3
$> echo "The night is dark and full of terrors" | ./my_pgp rsa -c 010001-c9\
f91a9ff3bd6d84005b9cc8448296330bd23480f8cf8b36fd4edd0a8cd925de139a0076b962f4d57f50d6f\
9e64e7c41587784488f923dd60136c763fd602fb3 > ciphered
$> cat ciphered
445b349e7318ad6af16b0bbb718be88ba1c41751f95751cd58857f88fe31f970405c6ec3f16d79172543b\
f4e571b5596d212f3e79cd08ef14abd244e325b80
$> cat ciphered | ./my_pgp rsa -d 81b08f4eb6dd8a4dd21728e5194dfc4e349829c99\
91c8b5e44b31e6ceee1e56a11d66ef23389be92ef7a4178470693f509c90b86d4a1e1831056ca0757f3e2\
09-c9f91a9ff3bd6d84005b9cc8448296330bd23480f8cf8b36fd4edd0a8cd925de139a0076b962f4d57f\
50d6f9e64e7c41587784488f923dd60136c763fd602fb3
The night is dark and full of terrors
```

Pretty Good Privacy



RSA solves the issue of secret exchange but is too slow and inconvenient for encrypting long messages. To build a practical application for sharing secret messages, **asymmetric** and **symmetric** cryptography must be combined:

- ✓ pick a random symmetric key ;
- ✓ cipher your message with the symmetric key ;
- ✓ cipher the symmetric key with the recipient public key ;
- ✓ display both the ciphered symmetric key and the ciphered message ;

To facilitate debugging, the ”random” symmetric key will be part of the key given in parameter to your program, both for encryption and decryption, in this format: ”SYMMETRIC_KEY:RSA_KEY”. The ciphered symmetric key must be displayed on the first line of your output, followed by the ciphered message.

Implement a cryptosystem using either XOR or AES as the symmetric cryptosystem, and RSA for the asym-

metric cryptosystem.

```
Terminal
$>  echo "All men must die" | ./my_pgp pgp-aes -c -b 57696e74657220697320636\
f6d696e67:010001-c9f91a9ff3bd6d84005b9cc8448296330bd23480f8cf8b36fd4edd0a8cd925de139a\
0076b962f4d57f50d6f9e64e7c41587784488f923dd60136c763fd602fb3 | cat -e
97f2af4c1b712008c1e46935f446756443a8a700f20581d138e4e6916afe5c5f9b9d6eaa0a870374b686f\
1a024f9bbb88c23c766654579339caf55afd149d41d$
744ce22c385958348f0df26eceb62eef$
$> echo "744ce22c385958348f0df26eceb62eef" | ./my_pgp pgp-aes -d -b 97f2af4\
c1b712008c1e46935f446756443a8a700f20581d138e4e6916afe5c5f9b9d6eaa0a870374b686f1a024f9\
bbb88c23c766654579339caf55afd149d41d:81b08f4eb6dd8a4dd21728e5194dfc4e349829c9991c8b5e\
44b31e6ceee1e56a11d66ef23389be92ef7a4178470693f509c90b86d4a1e1831056ca0757f3e209-c9f9\
1a9ff3bd6d84005b9cc8448296330bd23480f8cf8b36fd4edd0a8cd925de139a0076b962f4d57f50d6f9e\
64e7c41587784488f923dd60136c763fd602fb3 | cat -e
All men must die$
```

What you have built gets close to the real-life PGP cryptosystem.

With this, Arya could ask Brienne to generate a pair of keys and send back the public one. Arya can use this public key to cipher her long message with this cryptosystem, and send Brienne the message only she can read. Even if Euron intercepts it, he wouldn't be able to read it.

Bonus

As bonuses, you could:

- ✓ Implement 192-bit and 256-bit key size for AES.
- ✓ Add an option to generate the RSA keys using your own random prime number generator.
- ✓ Add **padding schemes** to secure even more your cryptosystems.
- ✓ Implement a way to **sign** the messages to guarantee who wrote them.
- ✓ Any other (interesting) cryptosystem.

Appendix

Find below some prime numbers for different key sizes:

- ✓ Four 8-bit primes
  - d3
  - e3
  - e9
  - f1
- ✓ Four 32-bit primes
  - 13d3c01d
  - 0b1e8a1e
  - d3dd082f
  - 0bfe6c0d
- ✓ Four 256-bit primes
  - 4b1da73924978f2e9c1f04170e46820d648edbee12ccf4d4462af89b080c86e1
  - bb3ca1e126f7c8751bd81bc8daa226494efb3d128f72ed9f6cacbe96e14166cb
  - d5da1a8d443812956185f9fe2d8696ea4959d3415967c7a8c5fec34bf7dd53e1
  - 4fbd36c46bd3fa40ebcef3447a4e2f2f16a6cee884bf889fd58ec5bca6024fe1
- ✓ Two 1024-bit primes
  - d7c9d58c694fe7ad5d77d888c98d71e7f6a58b4f4dc90582668fd28c0bc20f51
    c667ba7b70cb94842006eb5b223065346f7a6bb307ef572fee882c8ef420410b
    8b8fa8278ae6a300e63123b28ba1d47259bc308827fcd509585bcee1d98b461f
    9eff9c20559540b8c0d6036eff7caf0107d935ecefd5faab87b802bf74c041c8
  - e9dfe6b53248c4dc0f391fdda5694bd9f68e111dac5e921a942a157ec92431dc
    4833e1a327d36cebcc4ac9b76f2ac2a643db8a04c14963759a800f75915de3ff
    beccf86118923b388b352f7e3d20edfd5bb6609fd0b6416da6a56050b000ae9e
    14065f06f46ccfaa84c755689e8d95525bb1de8b8a43d380f4db68baac92e0bf

Larger primes

If you want more or larger primes, you can use `openssl genrsa 2048 | openssl rsa -text`.

OpenSSL prints it's values in a **big endian** notation. You'll have to convert them according to our nomenclature described at the beginning of this document.

v 2.1