**Fidelio** is a Python program designed to demonstrate encryption schemes.



**DISCLAIMER: This program should NOT be used for professional security applications. ALL of the encryption schemes implemented by Fidelio should be considered insecure against a knowledgeable attacker.**

To run Fidelio, you must have Python installed. It's available at www.python.org . (Python is free software and Mac and Linux computers may already have it installed.)

No installation is needed; just unzip the file Fidelio.zip and place the unzipped folder somewhere on your hard disk. (All files needed to run Fidelio are in this folder.)

Open a terminal, change to whatever directory contains Fidelio, and type:

```
python Fidelio.py
```

Select from the menu options by typing the appropriate (lower- or upper-case) letter.

Fidelio comes with three default **alphabets** which assign numerical values to text characters. Users can select from alphabets at runtime and/or input their own custom alphabets (up to 99 characters) in the Global Variables of the source code. The **Show Alphabets** command displays the alphabets in the current version of the program.

Fidelio's encryption schemes are called *Substitution, Caesar, Dodgson,* and *RSA* :

**Numerical substitution**

Each character is represented as a 2-digit decimal integer 00-99. (Which number corresponds to each character depends on the selected alphabet.) These integers are then packaged into "packets" of 8-digit long integers and printed to screen. The last packet is **padded** by adding random digits until it is 7 digits long, then one last digit records the number of padding-digits to delete later (including itself). This packaging may seem unnecessarily elaborate, but it will be useful later for RSA encryption.

Any characters not in the chosen alphabet are recorded as the number *N*, where *N* is the length of the alphabet. For example, using the 26-character alphabet A=0, B=1, C=2, ... Z=25, any other character = 26.

HELLO WORLD -> 07041111 14262214 17110392

Here H = 07 , E = 04 , L = 11 , and so on. The blank space is not part of the 26-char alphabet, so it is recorded as 26. Note that after D = 03 , the last digits are 92. The 9 is random and the 2 indicates that the last 2 digits are padding and should be ignored.

This scheme is not at all secure; its purpose is to use Python's string-slicing and type-conversion features to store text as a number so it can later be encrypted by a more "serious" numerical scheme. Also note that unless the alphabet has exactly 99 characters, the resulting output requires much more memory than the original message. Disregarding some obscure system characters, there are 96 ASCII characters, so ASCII encoding is reasonably efficient - though a "serious" encryption program would of course store text and ciphers as carefully-packaged binary data for maximum efficiency.

The functions makenumbers and makepackets are used to form the list of numbers and build a list of 8-digit long integers. The functions unpack and makeletters are used to break the 8-digit longs into a list of 2-digit ints and then recover the original text.

**Caesar encryption**

The classic Caesar encryption scheme creates a cipher by replacing each character in an alphabet by the character 3 places to the right. For a 26-letter alphabet, the result is:

A -> D , B -> E , C -> F , ... , X -> A , Y -> B , Z -> C

Note that the replacement "wraps around" for letters at the end of the alphabet. Representing (A,B,...,Z) as (0,1,...,25), Caesar encryption is equivalent to **modular addition** with 26 as the modulus. The function `caesarshift` converts a string into a list of numbers, then adds the value shift modulo the alphabet size. For the 26-letter alphabet, a shift value of 3 creates the classical Caesar cipher.

Caesar decryption is accomplished by subtracting the shift value using modular arithmetic, so there is no need for a decryption function. Note that a shift of 13 (known as **ROT13**) is encrypted and decrypted by exactly the same process. A shift of 0 (or 26 or any multiple of 26) leaves the original plaintext unchanged.

**Dodgson encryption**

This **polyalphabetic cipher** is more commonly called **Vignère encryption**, though it was actually invented by Giovan Bellaso in 1553. (Charles Dodgson published a version called "The Alphabet Cipher" in a children's magazine; if the 26-character alphabet is selected, Fidelio uses an identical scheme.)

This scheme uses a password as a **shared key** - the same password is used to encrypt and decrypt the message. For an N-character message, the password is repeatedly concatenated to itself to form a string `keystring` which is then converted into a list of numbers called `keylist`. For example, the16-char message MEETMEATMIDNIGHT with password FIDELIO results in the following `keystring` and `keylist`:

FIDELIOFIDELIOFI  5 8 3 4 11 8 14 5 8 3 4 11 8 14 5 8

The function `dodgsonencrypt` represents an input text as a list of numbers, then shifts the *k*-th element of the list forward by the *k*-th element of keylist and adds 1:

$$C_k = M_k + K_k + 1 \pmod{A}$$

where *C* is the resulting cipher, *M* is the message (as a number), *K* is the keylist and *A* is the alphabet size. (The 1 makes this scheme match the one Dodgson described.)

The function `dodgsondecrypt` recovers the original message from the cipher by subtracting the *k*-th key element (then subtracting 1) from the *k*-th cipher element.

Polyalphabetic ciphers make frequency analysis substantially more difficult, but they are not necessarily secure against sophisticated attacks. Variants (including the infamous *Enigma* machine) have been erroneously thought to be uncrackable for centuries.

A **one-time pad** is a polyalphabetic cipher in which the key is the same length as the message, the key characters are chosen completely at random, and the key is used exactly once. Claude Shannon proved in the 1940s that a one-time pad cannot be cracked by any opponent, even one with arbitrarily fast computing power.

**RSA encryption**

The keys in the Dodgson cipher and one-time pad are examples of **private keys** which share a common inconvenience: the key itself must itself be somehow secretly shared between Alice and Bob before any message is sent. **Public-key encryption** schemes such as RSA were developed to avoid this nuisance.

As an analogy, imagine that Alice wants to send a message that must not be read by anyone but Bob. Bob buys a padlock and keeps the only key for himself, then sends the (unlocked) padlock by mail to Alice. Alice put her message in an armored box, locks it with the padlock, then mails the box to Bob. Even if an eavesdropper (conventionally called Eve) intercepts the message, Eve cannot read it. In this case, the padlock is Bob's public key and the key that opens the padlock is his private key.

If Eve wants to **spoof** messages by pretending she is Alice, she must intercept Bob's padlock, use it to lock away a fake message, and mail that to Bob. To prevent this, Alice

stamps all her messages with an elaborate wax seal that can be easily recognized by Bob but is extremely difficult to create without Alice's special stamping-tool. Here the wax seal is Alice's public signature and the stamping-tool is her private (signature) key. So long as both private keys are always kept private, Eve cannot interfere.

Digital public-key encryption schemes send, instead of padlocks and wax seals, difficult math problems that are much easier to solve (for sending) or create (for signing) by someone who knows a certain hint. That hint then becomes a private key and the math problems are used as "padlocks." In the case of RSA encryption, the problems are:

**1. The discrete logarithm problem:**

$$M = \log_E[C] \ (\mod N)$$

where M is the original message, E is the public key (or "public exponent"), C is the encrypted ciphertext, and N is an RSA number sent along with the public key.

C ,E, and N are all large integers. (Fidelio uses 5- and 6-digit decimal integers for E and 9-digit decimal integers for N, while a military scheme might use 1024-digit binary integers.) The "log" here means "take the logarithm base E with respect to modular multiplication (mod N )," or equivalently, "find M such that $(M)^E \pmod N = C$ ."

**2. Semi-prime factorization:**

Given that N is the product of two prime numbers p and q, find p and q. (Such numbers are called semi-prime.) RSA private keys are generated using p and q, so if Eve can factor Alice's RSA number N , she can create her own copy of Alice's private key.

The security of RSA encryption depends on these problems being impractically slow to solve on present-day computers, while their inverse problems (modular exponentiation, multiplying two large integers) can be done quickly and reliably.

RSA encryption requires three numbers for each party involved in communication:

- an **RSA number** (denoted here as **N**) which is the product of two primes.
- a **public key** or public exponent which is coprime to the *totient* of N.
- a **private key** which is the multiplicative inverse of the public key ( mod totient ).

To send a secret message to Bob, Alice needs to know his RSA number and public key. The message can then only be read using Bob's RSA number and private key. Fidelio includes an option for automatically generating all three numbers, though users should note that **the numbers used by Fidelio are NOT large enough to ensure security.**

The RSA key-generation algorithm is described below:

0. Choose two primes p and q. Multiply them together to find N. Fidelio chooses p and q randomly from the second half of a list of 10,000 prime numbers in a separate file. This ensures large-but-manageable numbers.

1. Find **Euler's totient** of N, which is the number of elements in (mod N) arithmetic which have a multiplicative inverse. (These elements are called **units**.) For a semi-prime number like N, the totient equals (p-1) * (q-1) .

2. Chose a public key E that is large but less than the totient. E must be **coprime** to the totient (that is, they share no factors). Fidelio selects E by choosing another random prime from the same range as p and q, then checking that it is less than the totient and choosing again if needed.

3. The private key is the multiplicative inverse (mod totient) of the public key E. This can be found quickly using the extended Euclidean algorithm, which Fidelio does. Note that *anyone* can find a private key this way if they know the totient (p-1) * (q-1) of their victim's RSA number; this is why it is important that RSA numbers are too large to be easily factored. (Fidelio only chooses from 5000 primes, so it is not hard to factor Fidelio's generated RSA numbers by force.)

Once keys have been generated, Alice sends a message M to Bob by first encrypting it:

$$C = M^E \ ( \bmod N \ )$$

The result C is called the **ciphertext**, though it is actually a number. The message itself must be a number less than N. For messages larger than the RSA number N, the message is broken into smaller **packets,** each of which is then encrypted. (Fidelio uses 8-digit decimal long integers as packets and 9-digit decimal RSA numbers.)

To decrypt the message, Bob uses his private key to find:

$$C^\delta = (M^E)^\delta = M^{E\delta} = M^1 \ (\bmod N \ )$$

which, according to a result from number theory called the *Chinese remainder theorem*, will be true only if the private key δ and the public key E are multiplicative inverses ( mod totient ). This exponentation is much easier than the discrete logarithm problem.

Professional RSA encryption schemes also involve **digital signatures** in which a separate public key, private key, and RSA number are used along with a *hash function* to "sign" messages to prove that the sender is who he/she claims. Fidelio does not sign messages and is thus vulnerable to spoofing. In addition, more sophisticated schemes for packaging and padding a message are needed to guarantee true security. Lastly, remember that the prime numbers used by Fidelio are small enough to allow an unauthorized attacker to find the private key corresponding to an RSA number and public key! For these reasons, Fidelio should be used for educational purposes only.