Black Problem 3: Exceptional Encryption

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/ExceptionalEncryptionBlack on 3/15/2017

Starter file: hw9pr3.py.

In class, we have briefly mentioned the RSA algorithm, which is the most commonly used approach to *public-key encryption*. The feature of public-key encryption that makes it so widely used is that is *asymmetric*: the key that encrypts is distinct from the one that decrypts. That allows one of the keys to be published widely, while the other is kept secret. The RSA approach (named after its inventors, Ron Rivest, Adi Shamir, and Len Adleman) is also two-way: either key can encrypt while the other decrypts. The latter property allows us to support *digital signatures* in addition to secret messages.

In this assignment, you will implement a simple version of RSA encryption yourself. The implementation won't be "industrial strength"; in fact, you'll also develop a way to crack it. (Getting encryption exactly right is **really** difficult, and for real applications you should always use software written by a pro rather than trying to do it yourself. But this assignment will help you understand both the RSA algorithm and how to attack it.)

Outline of the RSA Algorithm

The RSA method is based on *modular exponentiation*—i.e., taking a number to a power modulo some other number. For example, you might encrypt the message M by raising it to the power of 23, modulo 143:

```
encrypted = M**23 % 143
```

If M = 7, then 7**23 = 27368747340080916343, and 7**23 % 143 = 2.
Then we can decrypt it by raising 2 to the 47th power modulo 143:

```
In [1]: 2**47
Out[1]: 140737488355328

In [2]: 2**47 % 143
Out[2]: 7
```

As you might suspect, the trick to making this work lies in the selection of the exponents and the modulus. Once you have those, Python will do the hard part of the calculation for you, as above. (There are *much* faster ways to do this exponentiation, but we won't attack those in this lab.)

Part I: Helper Functions

To make this lab work, you're going to need some helper functions. You'll find it useful to write the following:

- `isPrime(n)` returns True if `n` is prime.
- `nextPrime(n)` returns the first prime that is greater than `n`. An easy way to do that is to first ensure that `n` is odd, then keep adding 2 to it until it becomes prime.
- `gcd(a, b)` returns the greatest common divisor of `a` and `b`, using Euclid's algorithm: if `a` equals `b`, return a. Otherwise, if `a` is greater than `b`, return `gcd(a - b, b)`. Otherwise, return `gcd(b, a)`.

Try these functions on the following test cases:

```
In [1]: isPrime(2)
Out[1]: True

In [2]: isPrime(17)
Out[2]: True

In [3]: isPrime(49)
Out[3]: False

In [4]: isPrime(101)
Out[4]: True

In [5]: nextPrime(2)
Out[5]: 3

In [6]: nextPrime(3)
Out[6]: 5

In [7]: nextPrime(121)
Out[7]: 127

In [8]: gcd(48, 60)
Out[8]: 12

In [9]: gcd(100, 49)
Out[9]: 1
```

You will also need a function to compute the multiplicative inverse of a number in a modular field—that is, the number that, when multiplied by `n`, gives 1 mod `m`. We have provided that function to you in the file hw9pr3.py. Try it out:

```
In [1]: modularMultiplicativeInverse(2, 51)
Out[1]: 26

In [2]: modularMultiplicativeInverse(26, 51)
Out[2]: 2
```

Part II: `chooseKey`

We're now ready to write the guts of RSA encryption. The function `chooseKey` will select three numbers: a decryption exponent `d`, an encryption exponent `e`, and a modulus `m`. The encryption exponent and the modulus will be published as the *public* key; the decryption exponent and the modulus will be your *private* key. (Strictly speaking, either exponent can be chosen to be the private key, and either can be used to encrypt, with the other then serving to decrypt.)

`chooseKey` will accept a number indicating the approximate size of the desired key, and will return a triple `(d, e, m)` where `d` is the decryption exponent, `e` is the encryption exponent, and `m` is the modulus. For example:

```
In [1]: chooseKey(1000)
Out[1]: (237, 101, 391)

In [2]: chooseKey(1000)
Out[2]: (869, 101, 1739)
```

The steps for generating a key are as follows:

1. Choose two distinct random primes `p` and `q`, of approximately equal size. Each should be selected from the range `sqrt(n)/2`, `3*sqrt(n)/2`, where `n` is the argument to `chooseKey`. This will ensure that the eventual modulus (which is `p*q`) will be very roughly equal to `n`. (You will find the `randint` function, provided by the `random`module, useful for this purpose.)

   IMPORTANT DETAIL: Be sure `p` and `q` are different. If they're the same, choose the next larger prime for one of them.
2. Calculate the modulus, `m`, as `p*q`.

3. Calculate Euler's totient for `m`. Because `p` and `q` are prime, the totient will be `(p - 1) * (q - 1)`.
4. Choose a random number in the range (1, totient), such that gcd(e, totient) = 1. This will be your encryption exponent, `e`. (Note that the range excludes both 1 and the totient.)
5. Select `d` as the modular multiplicative inverse of `e` with respect to the totient (i.e., `d*e % totient` should equal 1).
6. Return the triple `(d, e, m)`.

IMPORTANT NOTE: when using `chooseKey`, always make sure that the returned modulus is at least 256. Otherwise you won't be able to tell all the ASCII characters apart, and you'll get garbled messages when you decrypt them.

Also, be careful not to pass huge numbers to `chooseKey`, or you can overload your computer so badly that you might need to reboot it.

Part III: Encryption and Decryption

Now it's time to write `encrypt(message, key)`. This function accepts a string (`message`) and an RSA encryption key consisting of an exponent and a modulus (as a tuple). It will return a list of integers that represents the encrypted version of the message. For example:

```
In [1]: encrypt('Hello, world', (101, 323))
[174, 169, 211, 211, 161, 74, 2, 272, 161, 190, 211, 291]
In [2]: encrypt('What is RSA**42?', (101, 253))
[186, 236, 251, 116, 219, 215, 115, 219, 192, 149, 76, 53, 53,
151, 39, 217]
```

It will be helpful to remember that the `ord` function will convert a character into the corresponding integer.

Decryption is essentially identical: `decrypt(L, key)` accepts a list of encrypted numbers and returns the decryption of same. For example:

```
In [1]: decrypt([174, 169, 211, 211, 161, 74, 2, 272, 161, 190,
211, 291], (77, 323))
Out[1]: 'Hello, world'

In [2]: decrypt([149, 244, 251, 10], (61, 253))
Out[2]: 'Spam'
```

Remember that the `chr` function will convert an integer back into a character.

Test your encryption and decryption functions with several different strings and different keys. Here's an example of an easy way to test:

```
In [1]: key = chooseKey(256)

In [2]: key
Out[2]: (237, 101, 391)

In [3]: secret = encrypt('This is a test', (key[1], key[2]))

In [4]: decrypt(secret, (key[0], key[2]))
Out[4]: 'This is a test'
```

Attacking an Encryption Algorithm

Sometimes, it's useful to be able to uncover the key of an encrypted message—usually one sent by one of your adversaries. There are two common attacks on cryptosystems: *known plaintext* and *unknown plaintext*. In the first attack, you have somehow acquired a copy of an unencrypted message (the "plaintext") and its encrypted equivalent. The problem is to recover the key that was used to encrypt the message (or, in a public-key cryptosystem, whichever key is unknown). Once you have that, you can read all future messages as well.

In the second attack, on the other hand, all you have is the secret message. If you can recover the key(s), you'll be able to read that message as well as future ones. Usually, an unknown-plaintext attack is more difficult.

The simplest way (in terms of programming) to figure out the key to a cryptosystem is "brute force": try all possible keys until you find one that works. A cryptosystem is considered strong if brute force is the best known way to attack it.

We'll write a brute-force known-plaintext decrypter, cleverly named `bruteForceKnownPlaintext`. It will take three arguments: the plaintext string, the encrypted message (as a list), and the publicly known modulus. It will try all possible keys in the range [2, modulus) and return one that produces the given plaintext, or `None` if there is no such key.

There is one tricky detail we'll have to deal with: when you try to decrypt a message with the wrong key, you may get a nonstandard character returned by the `chr`function. For example:

```
In [1]: chr(61)
Out[1]: '='

In [2]: chr(259)
Out[2]: ă
```

Notice that the return is a nonstandard character, and that you can reasonably assume that the plaintext *only* includes normal characters. You'll need to use a `try...except` clause to catch this and deal with it appropriately (we suggest returning `None`, which is the Python convention for "This is not a valid result.").

You will find that finding a key by brute force is only practical for relatively small key sizes (i.e., small arguments to `chooseKey`). That's the whole point: if you use a big enough key, it will be impossible to find it by brute force even with an unimaginably powerful supercomputer. Of course, there are other forms of brute force…

Double Bonus: Unknown-Plaintext Decryption

For 10 further bonus points, write another function named `bruteForce`, which will return a list of all keys that can correctly decrypt a message, given the modulus. Here's an example:

```
In [1]: bruteForce([149, 244, 251, 10], 253)
Out[1]: [15, 27, 61, 125, 137, 171, 235, 247]
```

Note that not all of the keys above are correct; for example, decrypting with 15 gives garbage. That's because we used such a short message; brute-forcing a longer one would give a shorter list.

How do you know you've successfully decrypted an unknown message? One way is to assume that it's plain text—i.e., that it only contains valid ASCII characters. You can test a given character for ASCII by including this code at the top of your file:

```
import string

def isprint(c):
    """ returns True if c is printable;
        False otherwise
    """
    return c in string.printable
```

With this code in place, `isprint(c)` will return True if `c` is a valid printing character (including spaces and newlines). If your entire decrypted string is valid, you might have a good decryption!

One last thing that it's good to know is that you can test a variable to see whether it is (or isn't) `None`. Although normal comparison for equality works just fine, Python offers another way that is preferred in this case:

```
    if x is None:
      print("x is None")
    if x is not None:
      print("x isn't None")
```

Why You Shouldn't Use This to Encrypt Your Banking Transactions

As mentioned earlier, the code you've written here is not of industrial strength. Hopefully not, if you can use a brute force algorithm to reliably decrypt it! One of the problems that an enterprising student may notice in the above examples is that the same letters map to the same numbers, making this a *monoalphabetic cipher*. For example:

```
encrypt('Hello, world', (101, 323))
[174, 169, 211, 211, 161, 74, 2, 272, 161, 190, 211, 291]
```

Note that both "l"s are represented by 211. Monoalphabetic ciphers are among the weakest forms of encryption, as they can be easily broken with frequency analysis. Polyalphabetic ciphers, in which the same letter can and will map to different outputs, are much more secure. Both polyalphabetic and monoalphabetic ciphers, however, are substitution ciphers. (The Enigma Device was also a substitution device! Albeit a very complex one...) RSA, however, is not.

From Ask A Mathematician:

"If your messages were 'Hello A', 'Hello B', and 'Hello C', then a substitution cypher might produce 'Tjvvw L', 'Tjvvw C', and 'Tjvvw S' while RSA (the most common modern encryption) might produce 'idkrn7shd', '62hmcpgue', and 'nchhd8pdq'. In the first case you can tell that the messages are nearly the same, but in the second you got nothing."

The real implementation of RSA encryption first converts your text (a string) into a bunch of bits (1's and 0's) and then converts those bits into a (usually somewhat large) number. This number (an integer, not a list of numbers like what you've implemented) is then taken to a power and then modulo some number, and effectively avoids the substitution problem. If this sounds interesting to you, there are some great explanations of the topic here and here (Warning: heavy math ahead!) If simulations are more your thing, you might like an Enigma Device simulator or an RSA simulator.

-- GeoffKuenning - 04 Nov 2013