JUNE 9TH, SUBSTITUTION CIPHERS

Never trust a computer you can't throw out a window.–Steve Wozniak

We are going to implement substitution ciphers and then make some tools to help us break them.

First we need to think about a way to represent substitution ciphers. Here is one way. A substitution cipher can be represented as a string of 26 distinct letters. The first letter can represent where `a` is sent, the second where `b` is sent, and so on.

For instance `"qwertyuiopasdfghjklzxcvnm"` would mean that $\texttt{a} \mapsto q, \texttt{b} \mapsto w, \ldots, \texttt{z} \mapsto m$.

Let's make a function in Sage which encrypts things. It will take in a string `key` which is a key like the one above. It will also take in a string `plaintext`. Here's the logic.

- Take the letter of plaintext you want to encrypt (we'll run this in a loop).
- Turn it into a number between 0 and 25 using the `ord` function (and a shift). Call this number `c`.
- Look up the corresponding letter in the `key`, ie `key[c]` should be the new letter.
- Store this new letter in your output string.

Here's how my function looked. You have to fill in the ... yourself with several lines of code.

```
def substitutionEncrypt(plaintext, key):
    outString = '' #
    for i in range(0,len(plaintext)):
        ...
    return outString

print substitutionEncrypt("abc","qwertyuiopasdfghjklzxcvbnm")
```

which output `qwe` as expected.

1. Make your own function that does the same thing.

Now we have to think about decryption. We are going to pass in the key the same way, and we'll pass in some ciphertext. Here's our strategy.

- Take the letter of ciphertext you want to decrypt.
- Find where it is in the key string (ie, what position). Call that `loc`. Notice that if you run

  ```
  s = "abc"
  loc = s.find("b")
  print loc
  ```

  Then you will have printed where "b" appears in the text "abc", in position number 1 in this case.
- The decrypted letter will be the letter corresponding to the number `loc`.

2. Make a function that descrypts things made with the substitution cipher.

3. Test your functions by encrypting and decrypting several strings of characters.

On the next page, we will make some functions that help decrypt substitution ciphers even if we don't know the key.

First, let's make a function that returns a list of let's us put in part of a key and spits out some partial plaintext. Here's how the function I made worked.

- I take in the ciphertext as a string `ciphertext`.
- I also receive a string with 26 characters in it. If I think `e` was converted to another letter, I put that other letter in the in the 5th spot in the string (the computer calls the 5th spot the 4th spot). The letters I don't want to do anything to, I put a pound sign `'#'` there.

For instance. If I think the key that the person used to make the substitution cipher turned `e` $\mapsto$ `c`, then I might have this string be

`####c####################`

Ok, now make a function which partially decrypts based on such a string. I find it useful for my decrypted letters to be uppercase while my encrypted letters to be lower case. For example, if I have the ciphertext `abc` and I use the partial key above, then my function would output `abE`. This makes it easy to to try out parts of keys and see if I'm getting close to decrypting. My function

`def substitutionPartialDecrypt(ciphertext, key):`

looks almost exactly like my regular decryption function, except I have some additional logic in the middle.

```
...
if (loc >= 0):
    outString = outString + chr(ord('A')+loc)
else:
    outString = outString + ciphertext[i]
...
```

4. Make your own function that partially decrypts based on a partial key.

5. Next make a function that lists how frequently each letter occurred. You already made a *more* complicated version of this function when we broke Vigenére.

(continued on the next page)

There is one more tool we need in order to crack a random substitution cipher. The ability to identify the most common digraphs[1] in a string. First, for any string `s` you can always take a substring `s[a:b]` where `a` is the position of the first character to take and `b` is the position where you stop taking characters. For example, try running

```
s = "abcde"
print s[1:3]
```

which should return `"bc"`.

Also experiment with the function `count` which tells you how many times a substring occurs.

```
s = "abcdeab"
print s.count("ab")
print s.count("bc")
```

My function is rather naive, it goes through my string, looking at each pair of consecutive letters and counts how many times they occur. Some things are listed more than once. You can make your function smarter if you want (or only print out digraphs that occur at least 5 or 10 times?). My function started like this.

```
def listCommonDigraphs(ciphertext):
    for i in range(0, len(ciphertext)-1):
        ...
```

7. Make your own function. Make sure to test it out and confirm it behaves like it should.

8. Use all your functions to decrypt the ciphertext available at:

`http://www.math.utah.edu/~schwede/Camp2016/Jun9SubstitutionBreakable.txt`

[1]Pairs of two letters.