CSE 101: Introduction to Computational and Algorithmic Thinking

Unit 11: Cryptography

Kevin McDonnell

Cryptography

- The field of **cryptography** (literally, “secret writing”) has a long history
- **Plaintext** refers to unencrypted data that can be intercepted by some means
- Encryption scrambles data in a way that makes it unintelligible to those unauthorized to view it
- The encrypted data is called **ciphertext**
- Modern encryption schemes often use **public-key cryptography**
- In public-key cryptography, each user has two related **keys**, one public and one private
- Each person’s public key is distributed freely

Cryptography

- A person encrypts an outgoing message using the receiver's public key
- Only the receiver's private key can decrypt the message
- The public/private key pairs are generated by a computer program in such a way that the decryption is made possible
  - The keys themselves are very large numbers that are hard to factor
  - The mathematical details are otherwise beyond the scope of the course
- In this Unit we will look at some simpler, but much less secure techniques for encrypting text

Caesar Cipher

- One of the simplest **ciphers** (algorithms for encrypting and decrypting text) is the **substitution cipher**, also known as the **Caesar cipher**
- The Caesar cipher works by replacing each letter of a word with the letter of the alphabet that is k letters later in the alphabet
- k is the **key** of the encryption scheme and provides the *shift amount*: a number in the range 1 through 25, inclusive
- In general, the **key** for a cipher is the secret piece of information that both parties must exchange ahead of time
- Julius Caesar used $k = 3$ in his military communications

Caesar Cipher

- For example, suppose we wanted to encode letters with $k = 3$
  - We would replace “A” with “D”, “B” with “E”, and so on
- For letters at the end of the alphabet, we “wrap-around” to the front of the alphabet
  - For $k = 3$, we would replace “X” with “A”, “Y” with “B”, and “Z” with “C”
- The phrase “Stony Brook” with a shift amount of 2 would be encrypted as “Uvqpa Dtqqm”
- To decrypt a message, we shift each letter of the encrypted message leftward in the alphabet by the shift amount

Caesar Cipher

- Let's consider functions **`caesar_encrypt`** and **`caesar_decrypt`**
- Both functions will take a string and a shift amount
  - For **`caesar_encrypt`**, the string is a plaintext message
  - For **`caesar_decrypt`**, the string is an encrypted message
  - Non-letter characters will be left unencrypted

Caesar Cipher

- The encryption algorithm is pretty straightforward:

1. First we map each letter to a number in the range 0 through 25: A → 0, B → 1, …, Z → 25
2. Next we add k to the number and *mod* by 26
3. Finally, we map the shifted value to a letter from the alphabet

- So, the encryption formula is $E(x) = (x + k) \bmod 26$, where x is the number for the plaintext letter, k is the key, and $E(x)$ gives the number for the ciphertext letter
- To decrypt, we subtract the key from the encrypted value, add 26 (to eliminate any negative differences), and mod by 26 to recover the original number

caesar_encrypt()

```
def caesar_encrypt(plaintext, shift_amt):
   ciphertext = ''
   for ch in plaintext:
      if ch.isupper():
         replacement = (ord(ch) - ord('A') +
                        shift_amt) % 26 + ord('A')
         ciphertext += chr(replacement)
      elif ch.islower():
         replacement = (ord(ch) - ord('a') +
                        shift_amt) % 26 + ord('a')
         ciphertext += chr(replacement)
      else:
         ciphertext += ch
   return ciphertext
```

See **caesar_cipher.py**

caesar_decrypt()

```
def caesar_decrypt(ciphertext, shift_amt):
    plaintext = ''
    for ch in ciphertext:
        if ch.isupper():
            replacement = (ord(ch) - ord('A') -
                        shift_amt + 26) % 26 + ord('A')
            plaintext += chr(replacement)
        elif ch.islower():
            replacement = (ord(ch) - ord('a') -
                        shift_amt + 26) % 26 + ord('a')
            plaintext += chr(replacement)
        else:
            plaintext += ch
    return plaintext
```

See **caesar_cipher.py**

Multiplicative Cipher

- The Caesar cipher encrypts and decrypts numbers by adding or subtracting the key to a plaintext letter's number (where A $\rightarrow$ 0, B $\rightarrow$ 1, …, Z $\rightarrow$ 25)
- Suppose we use multiplication instead and multiply each number by the key?
  - We then have a **multiplicative cipher**
- Provided that the key is *relatively prime* to 26, no two letters will be encrypted to the same ciphertext letter
  - Two numbers are relatively prime if they have no common factors except 1
- The encryption formula is $E(x) = kx \bmod 26$

Multiplicative Cipher

- Suppose the key is 7
- The letter A (0) is mapped to $(0 \times 7) \bmod 26 = 0$, which is also A
- The letter J (9) is mapped to $(9 \times 7) \bmod 26 = 11$, which is L
- Although this cipher seems to be more complex than the Caesar cipher, it is less secure than the Caesar cipher because the number of possible keys is smaller

Multiplicative Cipher

- Example with $k = 7$. So, $E(x) = 7x \bmod 26$.

Plaintext	x	$E(x)$	Ciphertext
A	0	0	A
B	1	7	H
C	2	14	O
D	3	21	V
E	4	2	C
F	5	9	J
G	6	16	Q
H	7	23	X
I	8	4	E
J	9	11	L
K	10	18	S
L	11	25	Z
M	12	6	G

Plaintext	x	$E(x)$	Ciphertext
N	13	13	N
O	14	20	U
P	15	1	B
Q	16	8	I
R	17	15	P
S	18	22	W
T	19	3	D
U	20	10	K
V	21	17	R
W	22	24	Y
X	23	5	F
Y	24	12	M
Z	25	19	T

multiplicative_encrypt()

```
def multiplicative_encrypt(plaintext, k):
   ciphertext = ''
   for ch in plaintext:
      if ch.isupper():
         replacement = ((ord(ch) - ord('A')) * k)
                       % 26 + ord('A')
         ciphertext += chr(replacement)
      elif ch.islower():
         replacement = ((ord(ch) - ord('a')) * k)
                       % 26 + ord('a')
         ciphertext += chr(replacement)
      else:
         ciphertext += ch
   return ciphertext
```

See **multiplicative_cipher.py**

Multiplicative Cipher

- To decrypt a message encrypted using this scheme we need to do some arithmetic to find the *modular multiplicative inverse of k with respect to 26*
- Going into that much math is a bit out of scope of the course
- So instead, to decrypt we could simply encrypt the entire alphabet to find the 26 mappings, and then perform the reverse mapping for each encrypted letter
  - Remember that the recipient knows the value of *k*
- To help us write this *brute force* algorithm we can use Python's **`zip`** function
- **`zip`** lets us iterate over two or more collections simultaneously

Aside: the `zip()` Function

```
names = ['Adam', 'Chris', 'Mary', 'Frank']
ages = [21, 19, 24, 22]
for name, age in zip(names, ages):
   print(name + ' ' + str(age))
```

- Output:

```
Adam 21
Chris 19
Mary 24
Frank 22
```

Multiplicative Cipher

- Two other Python tricks/features we'll use:
  - a dictionary comprehension, which we explored in an earlier Unit, and
  - the string called **`string.ascii_letters`**, which contains all 26 letters of the Latin alphabet in uppercase and lowercase

multiplicative_decrypt()

```
reverse_mapping = {}
decrypt_key = -1
def multiplicative_decrypt(ciphertext, k):
   global reverse_mapping, decrypt_key
   if k != decrypt_key:
      decrypt_key = k
      encrypted_letters = [multiplicative_encrypt(letter, k)
                           for letter in string.ascii_letters]
      reverse_mapping = {encrypted_letter: letter
                    for letter, encrypted_letter in
                    zip(string.ascii_letters, encrypted_letters)}
   plaintext = ''
   for ch in ciphertext:
      if ch in reverse_mapping:
         plaintext += reverse_mapping[ch]
      else:
         plaintext += ch
   return plaintext
```

See **multiplicative_cipher.py**

Affine Cipher

- An **affine cipher** combines ideas from the Caesar cipher and multiplicative cipher, performing both a multiplication and an addition
- The value x of some letter is encrypted using the formula $(ax + b)$ mod 26, where a is the *multiplier* and b is the *shift amount*
  - a and b together form the encryption key
- In some sense the affine cipher should be stronger than the Caesar and multiplicative ciphers, but it's still inherently weak because it's still a substitution cipher
- The encryption function looks similar to the one for the multiplicative cipher

affine_encrypt()

```
def affine_encrypt(plaintext, a, b):
   ciphertext = ''
   for ch in plaintext:
      if ch.isupper():
         replacement = ((ord(ch) - ord('A')) * a + b)
                          % 26 + ord('A')
         ciphertext += chr(replacement)
      elif ch.islower():
         replacement = ((ord(ch) - ord('a')) * a + b)
                          % 26 + ord('a')
         ciphertext += chr(replacement)
      else:
         ciphertext += ch
   return ciphertext
```

See **affine_cipher.py**

Rail Fence Cipher

- The **rail fence cipher** is a type of **transposition cipher**
  - In a **transposition cipher**, the characters in the original message are rearranged somehow (as opposed to being substituted)
- The rail fence cipher rearranges the characters in a zigzag pattern
  - The key is the number of rows used to create the zigzag
- For example, the message **`STONYBROOKUNIV`** written over two rows would look like this:

```
S   O   Y   R   O   U   I
 ↘ ↗ ↘ ↗ ↘ ↗ ↘ ↗ ↘ ↗ ↘ ↗ ↘
  T   N   B   O   K   N   V
```

Rail Fence Cipher: Encryption

```
S   O   Y   R   O   U   I
  T   N   B   O   K   N   V
```

- To produce the final encrypted message we read off the characters row-by-row: **SOYROUITNBOKNV**
- The same message written over three rows would look like this:

```
S       Y       O       I
  T   N   B   O   K   N   V
    O       R       U
```

- The encrypted message would be: **SYOITNBOKNVORU**

Rail Fence Cipher: Encryption

- To implement the rail fence cipher we will create a list of empty strings, one per row, and append characters one-by-one to each string
- We will use a variable **`row`** (initialized to 0) that will first increase towards **`num_rows`**, then decrease back towards 0, then increase again, etc., until the entire plaintext message has been encrypted
- This computation will be encapsulated in a helper function called **`next_row`**

`next_row()` Helper Function

```
def next_row(row, step, num_rows):
    if row == 0:
        step = 1
    elif row == num_rows - 1:
        step = -1
    row += step
    return row, step
```

- To get a sense of how this function works, let’s pretend that we have 4 rows in the grid and the plaintext message has 10 characters
- See **`railfence_cipher.py`**

`next_row()` Helper Function

```
def next_row(row, step, num_rows):
   if row == 0:
      step = 1
   elif row == num_rows - 1:
      step = -1
   row += step
   return row, step
```

Test code:

```
row = 0
step = 1
num_rows = 4
for i in range(10):
    print(row, step)
    row, step = next_row(row, step, num_rows)
```

Output:

row increasing

```
0 1
1 1
2 1
3 1
```

row decreasing

```
2 -1
1 -1
0 -1
```

row increasing

```
1 1
2 1
3 1
```

railfence_encrypt()

```
def railfence_encrypt(plaintext, num_rows):
   row = 0
   step = 1
   # create num_rows empty strings in a list
   rows = [''] * num_rows
   for ch in plaintext:
      rows[row] += ch
      row, step = next_row(row, step, num_rows)
   return ''.join(rows)
```

- The **`join`** function creates a string by concatenating the elements of a list together
- See **`railfence_cipher.py`**

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['',
        '',
        '']
```

Variable	Value
ch	
row	0
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['',
        '',
        '']
```

Variable	Value
ch	'S'
row	0
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        '',
        '']
```

Variable	Value
`ch`	`'S'`
`row`	0
`step`	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        '',
        '']
```

Variable	Value
ch	'S'
row	1
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
  rows[row] += ch
  row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        '',
        '']
```

Variable	Value
ch	'T'
row	1
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        '']
```

Variable	Value
ch	'T'
row	1
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        '']
```

Variable	Value
ch	'T'
row	2
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        '']
```

Variable	Value
`ch`	`'O'`
`row`	`2`
`step`	`1`

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        'O']
```

Variable	Value
ch	'O'
row	2
step	1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        'O']
```

Variable	Value
`ch`	`'O'`
`row`	`1`
`step`	`-1`

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'T',
        'O']
```

Variable	Value
ch	'N'
row	1
step	-1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'TN',
        'O']
```

Variable	Value
`ch`	`'N'`
`row`	`1`
`step`	`-1`

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'TN',
        'O']
```

Variable	Value
ch	'N'
row	0
step	-1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['S',
        'TN',
        'O']
```

Variable	Value
ch	'Y'
row	0
step	-1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['SY',
        'TN',
        'O']
```

Variable	Value
ch	'Y'
row	0
step	-1

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
   rows[row] += ch
   row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['SY',
        'TN',
        'O']
```

Variable	Value
`ch`	`'Y'`
`row`	`1`
`step`	`1`

Trace: `railfence_encrypt()`

- Function call: **`railfence_encrypt('STONY', 3)`**

```
rows = ['', '', '']
for ch in plaintext:
    rows[row] += ch
    row, step = next_row(row, step, num_rows)
```

- Contents of rows list:

```
rows = ['SY',
        'TN',
        'O']
```

Variable	Value
`ch`	`'Y'`
`row`	`1`
`step`	`1`

- Then we call **`''.join(rows)`** to generate the final ciphertext: **`'SYTNO'`**

Rail Fence Cipher: Decryption

- The idea for decryption is to first construct a grid using lists of lists of empty strings
- The key tells us how many rows are in the grid
- The length of the message tells us the number of columns
- Using the same zigzag path from the encryption algorithm, we place a **`None`** object (or some other marker) where the characters will go
- Then, we take letters one at a time from the encrypted text and move across the grid row by row, replacing the **`None`** values with characters from the encrypted message
- Finally, we trace out the zigzag pattern once more to read off the plaintext characters

Rail Fence Cipher: Decryption

- Example for ciphertext `'SYOITNBOKNVORU'` with **`num_rows = 3`**
- The input contains 14 letters, so we create a grid with 3 rows and 14 columns by creating a list containing 3 lists of 14 empty strings each:

Rail Fence Cipher: Decryption

- Next we travel in a zigzag pattern, inserting **`None`** objects, which are visualized below as dots:

●				●				●				●	
	●		●		●		●		●		●		●
		●				●				●			

Rail Fence Cipher: Decryption

- Then we travel across each row, inserting characters from the ciphertext whenever we find a **`None`** object
- The ciphertext is **`'SYOITNBOKNVORU'`**
- First row completed:

S				Y				O				I	
	●		●		●		●		●		●		●
		●				●				●			

Rail Fence Cipher: Decryption

- The ciphertext is `'SYOITNBOKNVORU'`
- Second row completed:

S				Y				O				I	
	T		N		B		O		K		N		V
		●				●				●			

- Third row completed: `'SYOITNBOKNVORU'`

S				Y				O				I	
	T		N		B		O		K		N		V
		O				R				U			

Rail Fence Cipher: Decryption

- We can now easily read off the original message by traversing the grid once again in zigzag order

S				Y				O				I	
	T		N		B		O		K		N		V
		O				R				U			

railfence_decrypt()

```
def railfence_decrypt(ciphertext, num_rows):
    grid = []
    for i in range(num_rows):
        grid += [[''] * len(ciphertext)]

    # set up the grid, placing a None value
    # where each letter will go
    row = 0
    step = 1
    for col in range(len(ciphertext)):
        grid[row][col] = None
        row, step = next_row(row, step, num_rows)
```

See **railfence_cipher.py**

railfence_decrypt()

```
# place characters from the encrypted
# message into the grid
next_char_index = 0
for row in range(num_rows):
    for col in range(len(ciphertext)):
        if grid[row][col] is None:
            grid[row][col] = ciphertext[next_char_index]
            next_char_index += 1
```

See **railfence_cipher.py**

railfence_decrypt()

```
# read the characters from the grid in
# zigzag order
plaintext = ''
row = 0
step = 1
for col in range(len(ciphertext)):
    plaintext += grid[row][col]
    row, step = next_row(row, step, num_rows)
return plaintext
```

See **railfence_cipher.py**

The Vigenère Cipher

- The **Vigenère Cipher** was invented in the 16th century by Frenchman Blaise de Vigenère
  - Uses a series of substitution ciphers to encode a message
  - Took about three centuries before cryptographers figured out a reliable way of cracking this cipher
  - Based on the use of a 26×26 grid of substitution ciphers, each one shifted to the right by one spot
  - We also need to pick a keyword or phrase that determines which rows of this grid to use

The Vigenère Cipher

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

A	B	C	D	E	F	G
A	B	C	D	E	F	G
B	C	D	E	F	G	H
C	D	E	F	G	H	I
D	E	F	G	H	I	J

- Note how each row is a different shifted alphabet of the kind used in the Caesar cipher

The Vigenère Cipher: Example #1

- Suppose our keyword is **`PYTHON`**
- Then we would use this part of the grid:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- If our message is longer than the key, we repeat the key as many times as needed to encode the message

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- To encrypt each plaintext letter, find its column along the top row of the table
- Then find the row for the corresponding letter from the key
- The cell at the intersection of that row and column gives the letter for the encrypted message

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	**R**	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M**

The Vigenère Cipher: Example #1

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	**W**	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	**I**	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I G**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I G T**

The Vigenère Cipher: Example #1

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	**P**	Q	R	S	T	U	V	W	X
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

- Example: encode **COMPUTER**
- Key: **P Y T H O N P Y**
- Plaintext: **C O M P U T E R**
- Ciphertext : **R M F W I G T P**

The Vigenère Cipher: Example #2

- Decryption follows the reverse procedure as for encryption
- Suppose the keyword is **JOKE** and the ciphertext is **OIXGCWYR**
- To decrypt, first arrange the repeated keyword and encrypted message as follows:
- Key: **J O K E J O K E**
- Ciphertext: **O I X G C W Y R**

The Vigenère Cipher: Example #2

- Then, search for each letter from the encrypted message in the row for the corresponding letter from the key
- The column label provides the decrypted letter
- The ciphertext is **O I X G C W Y R**
- For this ciphertext, we look up **O** in the row for **J**. Looking at the top row, we see that we are in column **F**

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	**O**	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D

The Vigenère Cipher: Example #2

- The ciphertext is **O I X G C W Y R**
- The plaintext so far is **F**
- Then we look up the **I** from the encrypted message in row **O**. The **I** is in column **U**. Our decrypted message so far is **FU**.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D

The Vigenère Cipher: Example #2

- The ciphertext is **O I X G C W Y R**
- The plaintext so far is **FU**
- Next we look up the letter **X** in row **K**. We find that **X** is in column **N**. Our decrypted message is now **FUN**.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	**I**	J	K	L	M	N
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D

The Vigenère Cipher: Example #2

- The ciphertext is **O I X G C W Y R**
- The plaintext so far is **FUN**
- Next we look up the letter **G** in row **E**. We find that **G** is in column **C**. Our decrypted message is now **FUNC**.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D

The Vigenère Cipher: Example #2

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	**O**	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	**C**	D	E	F	G	H	I
O	P	Q	R	S	T	U	V	**W**	X	Y	Z	A	B	C	D	E	F	G	H	**I**	J	K	L	M	N
K	L	M	N	O	P	Q	R	S	T	U	V	W	**X**	**Y**	Z	A	B	C	D	E	F	G	H	I	J
E	F	**G**	H	I	J	K	L	M	N	O	P	Q	**R**	S	T	U	V	W	X	Y	Z	A	B	C	D

- Continue in this fashion until the last letter and we finally decrypt the entire message
- Key: **J O K E J O K E**
- Ciphertext: **O I X G C W Y R**
- Plaintext: **F U N C T I O N**

The Vigenère Cipher

- To implement the Vigenère Cipher we do not need to represent the table in the computer's memory
- Instead, we can use the algorithm below, which computes the table entries "on the fly":

1. Map each letter from the plaintext to a number in the range 0 to 25, as we did with the other ciphers. (A → 0, B → 1, … , Z → 25)
2. Add this number to the number corresponding to the keyword's letter (and then mod by 26).
    - Example: for plaintext **`COMPUTER`** and keyword **`PYTHON`**
    - 2 is the number for C and 15 is the number for P
    - To encode C: C → 2→ (2 + 15) mod 26 = 17

The Vigenère Cipher

3. Convert the sum (mod 26) to its corresponding letter of the alphabet (with 0 → A, 1 → B, …, 25 → Z).

- The decryption algorithm performs a similar series steps, but in reverse order:

1. Map each letter from the encrypted message to a number in the range 0 to 25.
2. Subtract from this number the number corresponding to the keyword's letter.
3. Add 26 in case the subtraction resulted in a negative difference, and then compute the remainder mod 26.
4. Convert the resulting number to its corresponding letter of the alphabet (0 → A, 1 → B, …, 25 → Z).

vigenere_encrypt()

```
def vigenere_encrypt(plaintext, keyword):
    # duplicate the keyword as many times as needed
    keyword = keyword * (len(plaintext) // len(keyword) + 1)

    # convert plaintext letters to numbers
    plaintext_nums = [ord(ch) - ord('A') for ch in plaintext]

    # convert keyword letters to numbers
    keyword_nums = [ord(ch) - ord('A') for ch in keyword]

    # generate ciphertext
    ciphertext = ''
    for i in range(len(plaintext)):
        # add the two numerical codes and map sum (mod 26)
        # back to a letter
        ciphertext += chr((plaintext_nums[i]+keyword_nums[i])
                          % 26 + ord('A'))
    return ciphertext
```

vigenere_decrypt()

```
def vigenere_decrypt(ciphertext, keyword):
   # duplicate the keyword as many times as needed
   keyword = keyword * (len(ciphertext) // len(keyword) + 1)
   # convert ciphertext letters to numbers
   ciphertext_nums = [ord(ch) - ord('A') for ch in ciphertext]
   # convert keyword letters to numbers
   keyword_nums = [ord(ch) - ord('A') for ch in keyword]
   # generate plaintext
   plaintext = ''
   for i in range(len(ciphertext)):
      # subtract keyword num from ciphertext num, add 26
      # and map difference (mod 26) back to a letter
      plaintext += chr((ciphertext_nums[i]-keyword_nums[i]
                     + 26) % 26 + ord('A'))
   return plaintext
```

Encryption Algorithms

- A major drawback of the Caesar, multiplicative, affine, transposition and Vigenère ciphers (beside the obvious drawback that they can be broken) is the use of a *shared* private key
- The private key must first be exchanged, presumably in a face-to-face manner or some other “secure” way
- Public-key cryptography does not have this shortcoming: each person has a private key that is never shared and a public key that is shared
- The only known way at the moment to crack the hardest public-key encryption algorithms is to try virtually all the possible keys, which is an *intractable* problem

Cryptography Website

- www.counton.org/explorer/codebreaking/
- This is an excellent website that covers the basics of encryption.
- It includes programs you can use to test your knowledge of the ciphers we studied in this Unit