CSE 101: Introduction to Computational and Algorithmic Thinking

Unit 11:

Cryptography

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

- 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

- 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

- 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

Stony Brook University – CSE 101

- The encryption algorithm is pretty straightforward:
- 1. First we map each letter to a number in the range 0 through 25: A \rightarrow 0, B \rightarrow 1, ..., Z \rightarrow 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 $E(x) = (x + k) \mod 26$, 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
                                    See caesar cipher.py
   return ciphertext
```

```
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
                                    See caesar cipher.py
   return plaintext
```

- 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 $E(x) = kx \mod 26$

- Suppose the key is 7
- The letter A (0) is mapped to (0×7) mod 26 = 0, which is also A
- The letter J (9) is mapped to $(9 \times 7) \mod 26 = 11$, 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

• Example with k = 7. So, $E(x) = 7x \mod 26$.

Plaintext	X	E(x)	Ciphertext
A	0	0	A
В	1	7	Н
С	2	14	О
D	3	21	V
Е	4	2	С
F	5	9	J
G	6	16	Q
Н	7	23	X
I	8	4	Е
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
0	14	20	U
P	15	1	В
Q	16	8	I
R	17	15	Р
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	Т

```
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
```

- 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
```

- 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 = -1def multiplicative_decrypt(ciphertext, k): global reverse mapping, decrypt_key if k != decrypt key: decrypt key = kencrypted 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 See multiplicative cipher.py

return plaintext

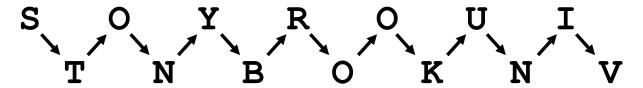
Affine Cipher

- 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
                                     See affine cipher.py
   return ciphertext
```

Rail Fence Cipher

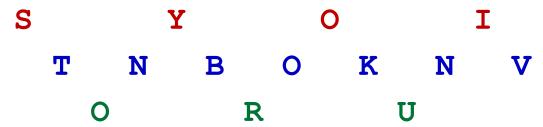
- The rail fence cipher is a type of transposition cipher
 - In a **transposition cipher**, the characters in the original message are rearranged somehow (as opposed to being substituted)
- The rail fence cipher rearranges the characters in a zigzag pattern
 - The key is the number of rows used to create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



Rail Fence Cipher: Encryption



- 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:



• 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):
                                                  Output:
   if row == 0:
      step = 1
                                          row
   elif row == num rows - 1:
                                       increasing
      step = -1
   row += step
   return row, step
                                          row
                                       decreasing
Test code:
row = 0
                                          row
step = 1
                                       increasing
num rows = 4
for i in range(10):
    print(row, step)
    row, step = next row(row, step, num rows)
```

```
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

• Function call: railfence encrypt('STONY', 3)

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

Variable	Value
ch	
row	0
step	1

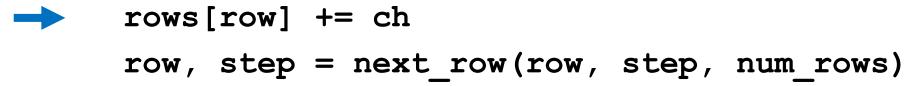
• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']

for ch in plaintext:

```
rows[row] += ch
row, step = next row(row, step, num rows)
```

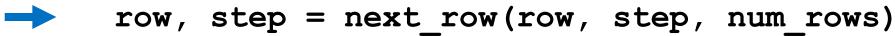
Variable	Value
ch	`S'
row	0
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:



Variable	Value
ch	`S'
row	0
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch



Variable	Value
ch	`S'
row	1
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']

for ch in plaintext:

```
rows[row] += ch
row, step = next row(row, step, num rows)
```

Variable	Value
ch	\T'
row	1
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:



Variable	Value
ch	\T'
row	1
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch



Variable	Value
ch	\T'
row	2
step	1

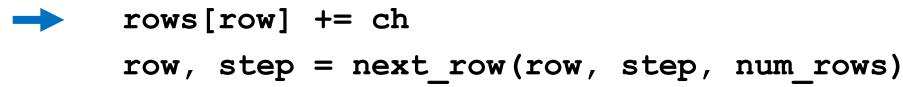
• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']

for ch in plaintext:

```
rows[row] += ch
row, step = next row(row, step, num rows)
```

Variable	Value
ch	'0'
row	2
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:



Variable	Value
ch	'0'
row	2
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch



Variable	Value
ch	'0'
row	1
step	-1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']

for ch in plaintext:

```
rows[row] += ch
row, step = next row(row, step, num rows)
```

Variable	Value
ch	'N'
row	1
step	-1

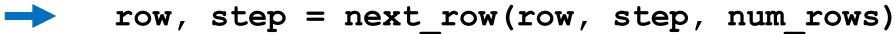
• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:



• Contents of rows list:

Variable	Value
ch	'N'
row	1
step	-1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch



Contents of rows list:

Variable	Value
ch	'N'
row	0
step	-1

• 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:

Variable	Value
ch	'Y'
row	0
step	-1

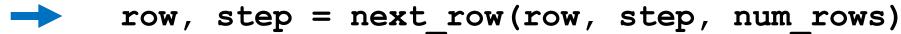
• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:



• Contents of rows list:

Variable	Value
ch	Y'
row	0
step	-1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch



Contents of rows list:

Variable	Value
ch	'Y'
row	1
step	1

• Function call: railfence_encrypt('STONY', 3)
rows = ['', '', '']
for ch in plaintext:
 rows[row] += ch
 row, step = next_row(row, step, num_rows)

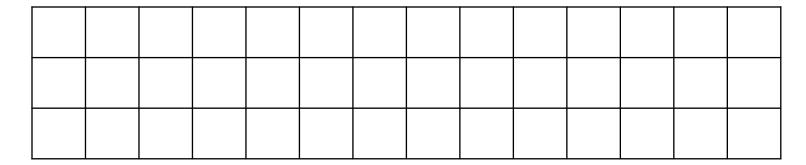
• Contents of rows list:

Variable	Value
ch	'Y'
row	1
step	1

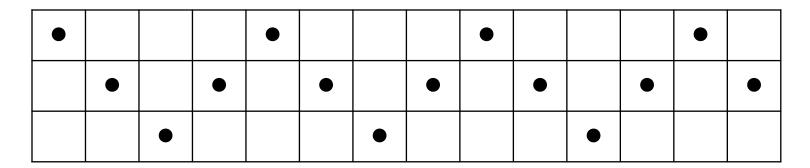
• Then we call ''.join (rows) to generate the final ciphertext: 'SYTNO'

- 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

- 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:



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



- 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				0				I	
	•		•		•		•		•		•		•
		•				•				•			

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

S				Y				0				I	
	T		N		В		0		K		N		٧
		•				•				•			

• Third row completed: 'SYOITNBOKNVORU'

S				Y				0				I	
	T		N		В		0		K		N		V
		0				R				U			

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

S				Y				0				I	
	T		N		В		0		K		N		V
		0				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

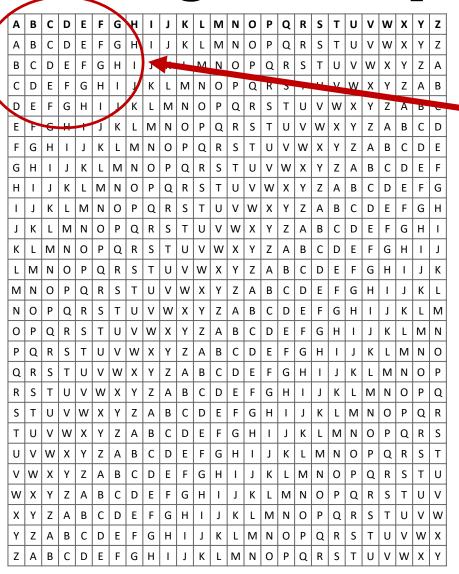
```
railfence decrypt()
# read the characters from the grid in
# zigzag order
plaintext = ''
row = 0
step = 1
for col in range(len(ciphertext)):
   plaintext += grid[row][col]
   row, step = next row(row, step, num rows)
return plaintext
```

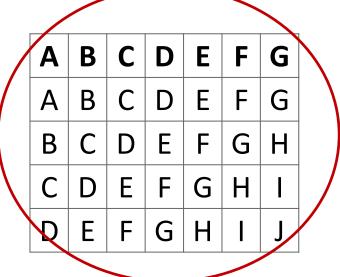
See railfence cipher.py

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

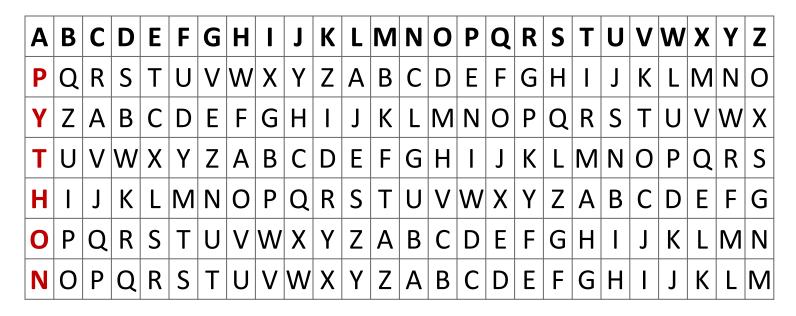
The Vigenère Cipher





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

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



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

Α	В	C	D	Ε	F	G	Н	ı	J	K	L	M	N	0	P	Q	R	S	T	U	V	W	X	Υ	Z
P	Q	R	S	T	U	V	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	ı	J	K	L	M	N	O
Y	Z	Α	В	С	D	Ε	F	G	Н		J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X
T	U	V	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	ı	J	K	L	M	Ν	0	Р	Q	R	S
Н	ı	J	K	L	M	N	0	P	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G
0	Р	Q	R	S	Т	U	V	W	X	Υ	Z	Α	В	С	D	E	F	G	Н	I	J	K	L	M	N
N	O	P	Q	R	S	T	U	V	W	X	Υ	Z	A	В	C	D	E	F	G	Н		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

```
G H I
                                          |\mathbf{Q}|\mathbf{R}|\mathbf{S}|
                        K L M N O P
                                                       UVWXYZ
            VWX
                        Z |A|B|
                                        Ε
                                              G H
                                     \mathsf{D}
                              \mathsf{E} \mid \mathsf{F} \mid
                  G H
            Z \mid A \mid B \mid
                     E
                                  G H
                                              K L M N O
V|W|X|Y|
                              F
      L M N O P Q R S T
                                 U \mid V \mid W \mid X \mid Y \mid Z \mid
                                                    A \mid B \mid
   R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
                                                 G H I
                                              F
            T|U|V|W|X|Y|Z|A|B|
                                              E
                                           |\mathsf{D}|
                                                    G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R

```
GHI
                                     |\mathbf{Q}|\mathbf{R}|\mathbf{S}|
                    K L M N O P
                                                 UVWXYZ
                    Z |A|B|
                                   Ε
                                        G H
        V|W|X|
                             L M NOPQRSTUVX
              G H
           |A|B|
                 C|D|E|
                             G H
                                           LMNO
   L M N O P Q R S T
                             U \mid V \mid W \mid X \mid Y \mid Z \mid
                                              A \mid B \mid
R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
                                           G H I
                                        F
           |U|V|W|X|Y|Z|A|B|
                                        E
                                     |\mathsf{D}|
                                              G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M

```
GHI
                                      |\mathbf{Q}|\mathbf{R}|\mathbf{S}|
                    K L <mark>M</mark> N O P
                                                  UVWXYZ
        V W X Y
                    ZAB
                                   E
                                         G H
                             L MNOPQRSTUVWX
              G H
           AB
                    D
                        Ε
                             G H
                                         K L M N O
   L M N O P Q R S T
                             U \mid V \mid W \mid X \mid Y \mid Z \mid
                                               A \mid B \mid
R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
                                            G H I
                                         F
           |U|V|W|X|Y|Z|A|B|
                                         E
                                      |\mathsf{D}|
                                               G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F

```
GHI
               K L M N O P Q R S
                                   UVWXYZ
       V W X Y
               Z |A|B|
                         Ε
                             G H
                       D
                   G H
       Z \mid A \mid B \mid
             E
                     G H
                               LMNO
V |W| X |Y|
                   F
             QR
                 ST
                     UVWXXYZ
                                 A \mid B \mid
    L M N O P
  R|S|T|U|V|W|X|Y|Z|A|B|C|D|E|
                               G H I
                             F
         |U|V|W|X|Y|Z|A|B|
                             E
                           |\mathsf{D}|
                                 G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F W

```
G H I
                UVWXYZ
        VWX
                Z |A|B|
                          Ε
                              G H
                        \mathsf{D}
                    G H
        Z \mid A \mid B \mid
              E
                      G H
                              K L M N O
V|W|X|Y|
                    F
    L M N O P Q R S T
                      U \mid V \mid W \mid X \mid Y \mid Z
                                  A B
    STUVWXYZAB
                        Ε
                                G H
                                           L |M| N
                              F
          |U|V|W|X|Y|Z|
                      |\mathsf{A}|\mathsf{B}|
                              E
                                  G H I
                            D
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F W I

```
GHI
              J K L M N O P Q R S
                                    UVWXYZ
        V W X Y
                Z |A|B|
                          Ε
                              G H
                    K L M N O P Q
            G H
        Z \mid A \mid B \mid
              E
                      G H
                              K L M N O
V |W| X |Y|
                    F
    L|M|N|O|P|Q|R|S|T|
                      U|V|W|X|Y|Z|
                                  A B
  R|S|T|U|V|W|X|Y|Z|A|B|C|D|E|
                                  HII
                                G
                              F
          UVWXYZAB
                              E
                                  G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F W I G

```
|\mathbf{Q}|\mathbf{R}|\mathbf{S}|
           G H I
                      K L M N O P
                                                   UVWXYZ
        UVWX
                      Z |A|B|
                                     Ε
                                          G H
                                  \mathsf{D}
                            G H
           Z \mid A \mid B \mid
                   E
                               G H
                                          K L M N O
V|W|X|Y|
                            F
      L M N O P Q R S T
                               U \mid V \mid W \mid X \mid Y \mid Z \mid
                                                A \mid B \mid
  R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
                                             G H I
                                          F
           T|U|V|W|X|Y|Z|A|B|
                                          E
                                       |\mathsf{D}|
                                                G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F W I G T

```
GHI
                    K| L |M| N |O | P |Q <mark>R</mark> S |
                                                 UVWXYZ
        V|W|X|
                    Z |A|B|
                                   Ε
                                         G H
                             L M N O P Q R S T U V W X
              G H
        ZAB
                 C|D|E|
                             G H
                                         K L M N O
   L|M|N|O|P|Q|R|S|T|
                             U \mid V \mid W \mid X \mid Y \mid Z \mid
                                               A \mid B \mid
R | S | T | U | V | W | X | Y | Z | A | B | C | D | E |
                                            G H I
                                         F
           |U|V|W|X|Y|Z|A|B|
                                         E
                                      |\mathsf{D}|
                                               G H I
```

• Example: encode **COMPUTER**

• Key: PYTHONPY

• Plaintext: C O M P U T E R

• Ciphertext: R M F W I G T P

- 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

- 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

Α	В	С	D	Ε	F	G	Н		J	K	L	M	N	0	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	
0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	С	D	Ε	F	G	Н	1	J	K	L	M	N
K	L	M	N	0	Р	Q	R	S	T	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	ı	J
E	F	G	Н	ı	J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D

- 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.

Α	В	С	D	Ε	F	G	Н	I	J	K	L	M	Ν	0	Р	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	
0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	1	J	K	L	M	N
K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J
E	F	G	Н		J	K	L	M	N	0	Р	Q	R	S	Т	U	V	W	X	Υ	Z	Α	В	С	D

- 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.

Α	В	С	D	Ε	F	G	Н	I	J	K	L	M	N	0	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	С	D	Ε	F	G	Н	
0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J	K	L	M	N
K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J
E	F	G	Н	I	J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D

- 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	В	С	D	Ε	F	G	Н	I	J	K	L	M	N	0	P	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	
0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J	K	L	M	N
K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	Χ	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J
Ε	F	G	Н	ı	J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	С	D

Α	В	С	D	Ε	F	G	Н		J	K	L	M	N	0	Р	Q	R	S	T	U	V	W	X	Y	Z
J	K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Υ	Z	Α	В	C	D	Ε	F	G	Н	
0	Р	Q	R	S	T	U	٧	W	X	Υ	Z	Α	В	С	D	Ε	F	G	Н	I	J	K	L	M	N
K	L	M	N	0	Р	Q	R	S	Т	U	٧	W	X	Y	Z	Α	В	С	D	Ε	F	G	Н	1	J
E	F	G	Н		J	K	L	M	N	0	Р	Q	R	S	T	U	V	W	X	Υ	Z	Α	В	С	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: FUNCTION

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 \rightarrow 0, B \rightarrow 1, ..., Z \rightarrow 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: $C \to 2 \to (2 + 15) \mod 26 = 17$

The Vigenère Cipher

- 3. Convert the sum (mod 26) to its corresponding letter of the alphabet (with $0 \rightarrow A$, $1 \rightarrow B$, ..., $25 \rightarrow Z$).
- 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 $(0 \rightarrow A, 1 \rightarrow B, ..., 25 \rightarrow Z)$.

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