<u>**Transposition Ciphers**</u>

- As mentioned earlier, Transposition ciphers are rarely used in modern cryptography, however they serve as good starting point for understanding the implementation of cryptographic algorithms and the methods used to crack transposition ciphers.

- They differ from both code systems and substitution ciphers in that a transposition cipher shifts the letters of the plaintext form the ciphertext.

- This can be done in a number of ways and there are some systems where even whole words are transposed, rather than individual letters.

- To encrypt Japanese, Chinese, or Arabic, for instance, one can use a transposition cipher operating on the individual characters of written Japanese (using a substitution cipher for a language like Japanese would be very difficult if not impossible).

- Transposition ciphers rearrange the letters of the plaintext without changing the letters themselves. For example, a very simple transposition cipher is the **rail fence**, in which the plaintext is staggered between two rows and then read off to give the ciphertext.

- In a two row rail fence the message: **lewis carroll** becomes (spaces are omitted):

  l w s a r l
  e i c r o l

- The rail fence is the simplest example of a class of transposition ciphers called route ciphers. These were quite popular in the early history of cryptography.

- Generally, the elements of the plaintext (usually in this case single letters) are written on a pre-arranged route into a matrix agreed upon by the transmitter and receiver.

- The example above has a two row by n-column matrix in which the plaintext is entered sequentially by columns, the encryption route is therefore to read the top row and then the lower.

- To approach any semblance of security, the route would have to be much more complicated than the one in this example.

<u>Single Transposition</u>

- One popular form of transposition uses a key or phrase to transpose each of the characters in the plaintext.

- As a simple case let us assume we have a key size of 12. In order to encrypt, we arrange the plaintext into a table with the number of columns equal to the key length. The plaintext is: "**a wilderness of mirrors**" (entered without spaces):

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
a	<sp>	w	i	l	d	e	r	n	e	s	s
<sp>	o	f	<sp>	m	i	r	r	o	r	s	

- The ciphertext is the letters read from the top left cell going down the column. The cipher text will be: "a", "sp", "sp", "w", "f", "i", "sp", "l", "m", "d", "i", "e", "r", "r", "n", "o", "e", "r", "s", "s", "s". The unused cells are simply ignored.

- The example program provided is a basic implementation of this cipher.

- The implement the decryption program for this cipher, we use the following steps:

    - Determine the number of columns that will be required by dividing the length of the message by the key, and then rounding up.
    - The number of rows will be equal to the key length.
    - Determine the number of unused cells in the list taking the product of the number of rows and columns, and subtracting the length of the ciphertext message. These will be ignored when filling in the list with ciphertext.
    - Start inserting the ciphertext into the list by starting at the top row and going from left to right.
    - The plaintext is extracted by reading from the leftmost column going from top to bottom, and moving over to the top of the next column.

- The essential piece of information in the decryption is the key length. A key length different from the one used to encrypt will result is an incorrect number of rows, and hence incorrect plaintext.

- The code example illustrates the decryption process.

- To make things a bit more difficult to crack, we can scramble the order in which the ciphertext is read, using a key phrase and omitting the spaces. One way is to define the order in which each column is written depending on the alphabetical position of each letter of the key relative to the other letters.

- Consider the following example:

    - The key phrase is" **lewiscarroll**" (note that since we have repeating characters in the phrases, we can simply assign numerical values to those sequentially as they appear).

    - The plaintext is: "**a wilderness of mirrors**" (entered without spaces).

    - The table is generated by first assigning numerical values corresponding to the alphabetic position of the key characters relative to each other.

    - Thus, the first occurrence of the letter **a** is numbered **1**, since there are no **b**'*s,* the next letter to be numbered is **c** followed by **e**, and so on.

    - Next the plaintext is written in rows under the numbered keyword, one letter under each letter of the keyword:

l	**e**	**w**	**i**	**s**	**c**	**a**	**r**	**r**	**o**	**l**	**l**
5	**3**	**12**	**4**	**11**	**2**	**1**	**9**	**10**	**8**	**6**	**7**
a	w	i	l	d	e	r	n	e	s	s	o
f	m	i	r	r	o	r	s				

- To generate the ciphertext, letters of the plaintext are copied down by reading them off column-wise in the order they appear using the enumeration of the keyword.

- The result will be the ciphertext (read in 5-letters groups) as follows:

    rreow mlraf sosns edrii

- Five letter groups is a tradition dating from when messages used to be transmitted by telegraph systems.

- In order to decrypt the ciphertext, we must first calculate the number of letters present in the ciphertext message. This will provide us with the number of letters that were originally present in the last row.

- As can be seen in the table, the last 4 columns - numbered **6**, **7**, **8**, and **10** - only contain single letters and this is important.

- The ciphertext contains 20 letters and the length of the key, which is 12 characters, is known to both parties. Therefore the last row must consist of eight letters (20 – 12), the last 4 final positions being empty.

- During the decryption process the 4 final positions of row 2 will be marked as empty. Then we number the keyword letters (similar to encryption) and then start by writing the first two characters of the ciphertext under keyword number one:

l	**e**	**w**	**i**	**s**	**c**	**a**	**r**	**r**	**o**	**l**	**l**
5	**3**	**12**	**4**	**11**	**2**	**1**	**9**	**10**	**8**	**6**	**7**
						r					
						r		*	*	*	*

- Next the ciphertext under keyword 2 gets filled, and so on. The table up to keyword 5 will be as follows:

l	**e**	**w**	**i**	**s**	**c**	**a**	**r**	**r**	**o**	**l**	**l**
5	**3**	**12**	**4**	**11**	**2**	**1**	**9**	**10**	**8**	**6**	**7**
a	w		l		e	r					
f	m		r		o	r		*	*	*	*

- Since the last position in columns 6, 7, and 8 are marked as unused, we only write a single character into those positions and continue on to obtain the complete table, from which we can read the plaintext:

l	**e**	**w**	**i**	**s**	**c**	**a**	**r**	**r**	**o**	**l**	**l**
5	**3**	**12**	**4**	**11**	**2**	**1**	**9**	**10**	**8**	**6**	**7**
a	w	i	l	d	e	r	n	e	s	s	o
f	m	i	r	r	o	r	s	*	*	*	*

- In implementing Transposition ciphers like the one above, it is common practice to add dummy characters to ensure that final 5-letter is always full if it is not.

- It is important to do this before transposing the letters, otherwise the receiver will not be able to determine which of the columns do not have a full set of characters if the last row is not complete.

- The code example provided illustrates this model.

Double transposition

- Double transposition is similar to single transposition, except that the process is repeated twice with two different keys.
- For example we will double transpose our previous example.
- Recall that after the first transposition we had the following result and ciphertext:

l	**e**	**w**	**i**	**s**	**c**	**a**	**r**	**r**	**o**	**l**	**l**
5	**3**	**12**	**4**	**11**	**2**	**1**	**9**	**10**	**8**	**6**	**7**
a	w	i	l	d	e	r	n	e	s	s	o
f	m	i	r	r	o	r	s				

- The resulting ciphertext (in 5-letters groups) is as follows:

  rreow mlraf sosns edrii

- Now we carry out yet another single transposition on the ciphertext using a different key. The key phrase this time is ”**charlesdodgson**”, and the text to be encrypted is “**rreow mlraf sosns edrii**”:

c	**h**	**a**	**r**	**l**	**e**	**s**	**d**	**o**	**d**	**g**	**s**	**o**	**n**
2	**7**	**1**	**12**	**8**	**5**	**13**	**3**	**10**	**4**	**6**	**14**	**11**	**9**
r	r	e	o	w	m	l	r	a	f	s	o	s	n
s	e	d	r	i	i								

- The resulting ciphertext will be:

  edrsr fmisr ewina sorlo

- This time in order to decrypt the ciphertext, we repeat the decryption process described earlier twice, with two different keys to obtain the original plaintext.

Breaking Transposition Ciphers

- In a message encrypted with the transposition cipher the number of possible keys is potentially too large to break unless we use large amounts of computing power to brute force that many keys.

- Once we have the outputs from all possible key permutations, we then face the equally large problem of examining the decrypted plaintext to find the correct plaintext, i.e., the one that linguistically makes sense.

- What all this implies is that we need a software algorithm that will recognize whether of or the plaintext produced by the cryptanalysis process is linguistically correct.

- Basically all this algorithm would have to do is pause when it has detected a recognizable word (from the language alphabet) and alert the analyst who will then either accept that as a key or allow the algorithm to continue to work through more keys.

- Another technique is to exploit the fact that transposition does not affect the frequency of individual symbols. This means that simple transposition can be easily detected by the by doing a frequency count.

- If the ciphertext exhibits a frequency distribution very similar to plaintext, it is most likely a transposition. This can then often be attacked by anagramming, i.e., sliding pieces of ciphertext around, then looking for sections that look like anagrams of English words, and solving the anagrams.

- As a proof of concept we will implement an algorithm that will recognize English words by matching decrypted words in a string against an English language dictionary. We will then apply it to crack a simple transposition cipher.

- In our simple transposition algorithm presented earlier, the ciphertext maintained all the spaces present in the plaintext. This makes it relatively easy to implement an algorithm that will extract individual words, attempt to crack them using different keys, and then match the results against dictionary words to find a match.

- There are many dictionaries for a variety of languages available online. We will use one such dictionary which contains upper case English words.

- There are two very useful Python features that we can use to implement this algorithm. One is the ***spilt()*** method, which returns a list of all the words in the string, using whitespace (be default) as a separator:

```
>>> string = "a wilderness of mirrors"
>>> string.split()
['a', 'wilderness', 'of', 'mirrors']
>>> string = "a;wilderness;of;mirrors"
>>> string.split(";")
['a', 'wilderness', 'of', 'mirrors']
>>>
```

- The other useful feature is the built-in **dictionary** type called ***dict*** which you can use to create dictionaries with arbitrary definitions for character strings. It can be used for the common usage, as in a simple English-French dictionary:

```
>>> french = dict()
>>> french['hello'] = 'bonjour'
>>> french['yes'] = 'oui'
>>> french['no'] = 'non'
>>> print(french['hello'])
bonjour
>>> print(french['no'])
non
>>>
```

- The dictionary data type has values which can contain multiple other values, just like lists do. To retrieve items from a list we use integer index values. In list values, you use an integer index value to retrieve items in the list, like **english[2]**.

- For each **item** in the dictionary contains a value and an associated key, which can used to retrieve it. The key can be an integer or a string value, like **french['hello']** or **french[2].**

- Items in a dictionary are stored as key:value pair. Multiple key-value pairs are separated by commas. To retrieve values from a dictionary, we use square brackets with the key in between them:

```
language = {'key1': value1, ‘key2’:value2}
```

- The following examples illustrate this:

```
>>> french = {'1':'bonjour', '2':'oui'}
>>> french['1']
'bonjour'
>>> french['2']
'oui'
```

- Note that keys can be integers or a strings:

```
>>> french = {'bonjour':'1', 'oui':'2'}
>>> french['bonjour']
'1'
>>> french['oui']
'2'
```

- Dictionaries seem similar to lists but it is important to keep some important differences in mind:

    - A list is just that - a list of values, each one of them is indexed using integers starting from zero – to the length of the list minus 1.
    - Values can be removed from the list (mutable), and new values added to the end of the list.
    - In a conventional dictionary, we have an 'index' of words; for each of them there is a definition. In python, the word is called a 'key', and the definition a 'value'.
    - The values in a dictionary aren't numbered and they are not in any specific order either - the key does the same thing. You can add, remove, and modify the values in dictionaries.
    - Dictionaries do not have concatenation with the + operator. To add a new item to a dictionary, we use indexing with a new key.

- In order to determine whether or not the words in a string are in the dictionary we need to establish some criteria as to the percentage of words that should be English words, and the percentage of characters in the string that are letters as opposed to other special characters.

- A loose threshold might be 25% and 85%. A tighter threshold might be 60% and 90%. This would all depend on the type of dictionary we are using.

- Another important parameter is the actual percentage of English words in a string or message. Then we can establish a threshold to establish the likelihood that the message is in fact predominantly English.

- The calculation is very simple: divide the number of English words by the total number of words and multiply by 100.

- The above parameters are important considerations in the software implementation of the language detection algorithms.

- The code example provided illustrates an implementation of an English language detection application.

Breaking Transposition Ciphers

- It is relatively simple task to break transposition ciphers using a brute-force approach. The basic process is to carry out the decryption process using every possible combination until the algorithm begins to detect recognizable English words.

- We will use the English language detection program to implement the brute-force cryptanalysis application.

- The application will also use the transposition decryption program presented earlier. The decryption function will be invoked with a range of possible keys. The range will be integers between 1 and the length of the ciphertext.

- The process of iterating through the key lengths will pause once the algorithm matches words in the dictionary and wait for user input to either confirm the correct key or continue with the brute-force iterations.

- In the next assignment you will implement the code to break the simple transposition cipher.

- With a few modifications, this program can also be used to break the scrambled transposition cipher as well. Since we will now have to brute-force different key combinations and not just key lengths, the computing effort will be a huge order of magnitude greater.

- A better approach would be to use frequency analysis and English language detection.