Columnar Transposition Cipher: An Introduction

David W. Agler

November 29, 2023

Columnar Transposition Cipher

The columnar transposition cipher is a type of transposition cipher. A transposition cipher is a method of encryption that involves (rather than the substitution of one letter for another) the rearrangement of the plaintext into ciphertext. More plainly, transposition ciphers simply mix up the order, arrangement, or presentation of the letters to get the ciphertext. In what follows, I'll provide a brief introduction to one of the simplest forms of transposition ciphers: the columnar transposition cipher.

Columnar Transposition

Let's start with an example. Take the plaintext "ABCD" and put it in a grid (or table) of two columns. We will write the plaintext in the grid from left to right and top to bottom. Here is the result:

1	2
A	B
C	D

We can say we have a geometrical figure: a rectangle. Next, let's use this rectangle to create our ciphertext. Instead of reading from left to right and top to bottom, let's read from top to bottom and left to right. This gives us "ACBD". This is our ciphertext.

A few things to note. First, we did not substitute any letters in our plaintext. Rather, we simply rearranged or transposed the letters. Second, note that the length of the plaintext and the ciphertext is the same. Third, in our example, we put the plaintext in a grid that has 2 columns. But consider that, with different plaintext, we might have chosen a different number of columns. Let's illustrate this with an example. Suppose we want to send that special someone the following message "I love you". We could, like we did in the previous example, put this plaintext in a grid with 2 columns like this (leaving the cell blank if there is no letter to fill it):

1	2
I	
l	o
v	e
	y
o	u

This would give us the ciphertext "Ilv o oeyu". But, we could have also put the plaintext in a grid with 3 columns like this:

1	2	3
I		l
o	v	e
	y	o
u		

Here our ciphertext is "Io u vyleo". The number of columns we choose is the *key*. In the first example, the key is 2. In the second example, the key is 3.

	1	2	3	4	5	6	7	8	9	10
Plaintext	I		l	o	v	e		y	o	u
Key 2	I	l	v		o	o	e	y	u	
Key 3	I	o		u		v	y	l	e	o

Again, notice that in both cases, the length of the plaintext and the ciphertext is the same.

Now that our special someone has our ciphertext message, how do they decrypt it to get our message? Or, suppose we are given the ciphertext "HlWleoodl r" and a key of 3. How would we decrypt this message? Gaines gives the following instruction: "if the encipherer [. . .] has written the text in rows and taken it off by columns, then the decipherer must do the reverse: write his text by columns and take it off by rows" (*Cryptanalysis*, p.14). Let's put this in a punchier way:

1. Encipherment by rows, then columns.
2. Decipherment by columns, then rows.

Since we know the key is 3, we can create a table with 3 columns and write the ciphertext from *top to bottom* and *left to right*. As the first three letters of "HlWleoodl r" are "HlW", we would start by writing "HlW" from the top to the bottom in the first column.

1	2	3
H	.	.
l	.	.
W	.	.

But as we begin writing from top to bottom, a problem arises. How many rows down do we go? We can figure this out by dividing the length of the ciphertext by the length of the key and rounding up. In our case, the length of the ciphertext "HlWleoodl r" is 11 and the length of the key is 3. So, we have $11/3 = 4$ (rounding up). So, our table should have 3 columns (from the key) and 4 rows. Let's complete our table using "HlWleoodl r":

1	2	3
H	e	l
l	o	
W	o	r
l	d	

With our table created, we get the plaintext by reading the letters from the table from *left to right* and *top to bottom*. In our case, the plaintext is "Hello World".

Thus far, our approach to encryption has been to read the grid from *left to right* and *top to bottom*. But, it is worth pointing out several other variations. First, we could have performed the encryption by reading the grid from *top to bottom* and *right to left*. Or, we could have read the grid using a snake pattern, starting in the first column, reading downward, moving to the right, then *bottom to top*. Another common approach is to use a keyword that is the length of the key and have some feature of the keyword to guide the order in which the columns should be read. For example, suppose we use the keyword "SUNDAY" (a key length of 7) to encrypt the message "Tomorrow is the day we attack." We would start by writing the keyword "SUNDAY" at the top of our grid and write the plaintext in the normal way.

S	U	N	D	A	Y
4	5	3	2	1	6
T	o	m	o	r	r
o	w		i	s	
t	h	e		d	a
y		w	e		a
t	t	a	c	k	.

Finally, rather than starting with the upperleft corner, a number (corresponding to the character's position in the alphabet) is used to determine the order in which the columns should be selected to construct the ciphertext. In the example above, we read from top to bottom starting under "A" (1), then "D" (2), then "N" (3) and so on. The ciphertext is "rsd koi ecm ewaTotytowh tr aa."

In this section, we considered a simple case of a transposition cipher: the columnar transposition cipher. We took plaintext and turned it into ciphertext by creating a grid, writing the plaintext horizontally in the grid, then extracting the ciphertext by reading vertically. Our process of decryption was essentially the reverse: we wrote the ciphertext vertically, then extracted the plaintext

by reading the grid horizontally. In the next section, we'll consider how to implement the columnar transposition cipher in Python.

Python Implementation

In this section, we'll focus on how to encrypt plaintext using the columnar transposition cipher in Python before examining how to decrypt the ciphertext. In looking at encryption, we'll consider two different ways to implement the columnar transposition cipher, one involves creating the grid (and closely following how we might create the ciphertext by hand) and the other using a list of strings.

Encryption Method 1: The Intuitive Method

One way to implement the columnar transposition cipher in Python is to create it just like we did by hand. We'll start by creating a function that takes `plaintext` and `key` as arguments. Then, we'll simply create a list filled with strings. The length of the string will be the number of columns and each string in the list will be a row in our grid. What we want is something like this:

```
def grid_create(plaintext, key):
    grid = [] # empty list
    # Start at 0 and get key length num of chars
    # Append chars to grid
    # Skip the length of the key
    # get the next char a key length from 0
    return grid

print(grid_create("Hello World", 3))
# ['Hel', 'lo ', 'Wor', 'ld']
```

What we don't want to do is loop over each item in the `plaintext`. Instead, if our key is 3, we want indices 0 (get 3 chars), move to index 3 (get 3 chars), and so on until we reach the end of the plaintext. To accomplish this, let's use a `for` loop. We'll start at 0 and stop at the end of `plaintext`. We also want to skip the length of the key.

```
def grid_create(plaintext, key):
    grid = []
    for i in range(0, len(plaintext), key):
        # Append chars to grid
    return grid

print(grid_create("Hello World", 3))
# ['Hel', 'lo ', 'Wor', 'ld']
```

So, in the above, if the length of the `plaintext` is 11 and the length of the `key` is 3, then we will loop through the range of 0 to 11 in increments of 3. That is, we will loop through 0, 3, 6, 9. Note that 0, 3, 6, 9 are not our columns. The

columns are specified by the key. The rows are specified by whatever number of loops our range has. So, in the above, we will loop through the range of 0 to 11 in increments of 3. This will give us 4 loops. So, we will have 4 rows.

All this leaves us is to append the characters to the grid on each loop. We can do this by appending the `plaintext` at index `i` to the `plaintext` at index `i` plus the length of the `key` (3).

```
def grid_create(plaintext, key):
    grid = []
    for i in range(0, len(plaintext), key):
        grid.append(plaintext[i:i+key])
    return grid

print(grid_create("Hello World", 3))
# ['Hel', 'lo ', 'Wor', 'ld']
```

Let's describe the first two iterations of the loop:

1. The first iteration of starts with 0. To the `grid`, characters from index 0 to index 3 (indices 0, 1, 2) are appended. The `grid` is now `['Hel']`. This is the first row of the grid. Column 1 is 'H', column 2 is 'e', and column 3 is 'l'.
2. The second iteration of the loop starts with 3. To the `grid`, characters from index 3 to index 6 (indices 3, 4, 5) are appended. The `grid` is now ['Hel', 'lo']. This is the second row of the grid. Column 1 is 'l', column 2 is 'o', and column 3 is an empty space.

This is exactly what we want. It looks just like the grid we created by hand.

1	2	3
H	e	l
l	o	
W	o	r
l	d	

The first step in our encryption is complete. We have created the grid. The next step is to read the grid and create the ciphertext. To do this, we will create a variable called `ciphertext` and set it to an empty string. Writing the `plaintext` to the grid involved writing the `plaintext` from left to right and top to bottom. To create the ciphertext, we want to read the grid from top to bottom and left to right. Since our grid is a list of strings, and each character in the string represents a column, we want to go string by string, getting the `i` index character from each string. So, we want to add the first letter of each item in the grid to the `ciphertext`, then the second letter of each item in the grid to the `ciphertext`, and so on. To illustrate, let's say our grid is the following:

```
grid = ['Hel', 'lo ', 'Wor', 'ld']
```

We want to create `ciphertext` string in the following way:

Index	Character
0	H
0	l
0	W
0	l
1	e
1	o
1	o
1	d
2	l
2	
2	r

How do we do this? First, let's create `ciphertext`, which is an empty string to store our ciphertext. Next, let's add a `col` variable. The `col` is used to keep track of what column we are in. We will increment when we need to move to the next column.

```
def columnar_encrypt(plaintext, key):
    ciphertext = ''
    # earlier grid code that constructs the grid
    col = 0
        # go col by col and get the i index char from each string
            # add that char to ciphertext
        col += 1
    return ciphertext
```

Let's add a `while` loop that will loop until `col` is greater than length of the `key`. The idea here is that we want to loop through the index of each column and we want to stop once there are no more columns. Since the `key` is equal to the number of columns, we can stop the `while` function when the `col>key`.

```
def columnar_encrypt(plaintext, key):
    ciphertext = ''
    # early grid code
    col = 0
    while col < key:
        # go col by col and get the i index char from each string
            # add that char to ciphertext
        col += 1
    return ciphertext
```

Next, we want to loop through each string (row) in the grid. We can do this by using `for` loop on the items in the grid. During the loop, we will add the character at the index of the `col` to the `ciphertext`.

```
def columnar_encrypt(plaintext, key):
    ciphertext = ''
    grid = []
    for i in range(0, len(plaintext), key):
        grid.append(plaintext[i:i+key])
    col = 0
    while col < key:
        for row in grid:
            ciphertext = ciphertext + row[col]
        col += 1
    return ciphertext
```

Let's test our function.

```
print(columnar_encrypt("Hello World!", 3))
# HlWleoodl r!
print(columnar_encrypt("ABCDEFGHIJK", 3))
# Error!
```

While our function works for some plaintext, it doesn't work for others. To see why, here is our `grid` for "ABCDEFGHIJK": ['ABC', 'DEF', 'GHI', 'JK']. Putting this in our handy table, we get the following:

0	1	2
A	B	C
D	E	F
G	H	I
J	K	

What we expect then is the following ciphertext "ADGJBEHKCFI". But, when we run our function, we get an error. Why? As the function writes the ciphertext, remember it is writing it from top to bottom and left to right. But notice that when `row=4`, there is no item in `col=2`. So, when the function tries to add `row[4][2]` to the `ciphertext`, it will return an error stating that the string index is out of range. To illustrate, if we print `ciphertext` immediately after we add the character to it, we see it will print the `ciphertext` and the error will occur on the next loop.

Col	Ciphertext
0	A
0	AD
0	ADG
0	ADGJ
1	ADGJB
1	ADGJBE
1	ADGJBEH
1	ADGJBEHK

Col	Ciphertext
2	ADGJBEHKC
2	ADGJBEHKCF
2	ADGJBEHKCFI
2	ERROR!

The fundamental problem is that the `col` index is greater than the number of characters in the string. So, let's simply add an `if` statement that checks to see if the `col` index number is less than the length of the string (`row`). If it is, then we'll add the character at `row[col]` to the `ciphertext`. If it isn't, then we'll break out of the loop. Alternatively, we can simply add a dummy character to our ciphertext.

```
def columnar_encrypt(plaintext, key):
    ciphertext = ''
    grid = []
    for i in range(0, len(plaintext), key):
        grid.append(plaintext[i:i+key])
    col = 0
    while col < key:
        for row in grid:
            if col < len(row):
                ciphertext = ciphertext + row[col]
            else:
                break # alt: ciphertext += '*'
        col += 1
    return ciphertext
```

Let's test this function with a few examples.

```
print(columnar_encrypt("Run, they are coming", 2))
# Rn hyaecmnu,te r oig
print(columnar_encrypt("ABCDEFG", 3))
# Hore llWdlo
print(columnar_encrypt("ABCDEFGHIJK", 3))
# ADGJBEHKCFI
```

In this section, we implemented the columnar transposition cipher by taking plaintext and writing it to a grid (columns and rows). We wrote the plaintext in a grid *left to right* and *top to bottom* by appending to a list strings from the plaintext the size of the key (which stand for the number of columns of the grid). In doing so, we let each character in the string represent a column, and each string in the list represent a row. The next step was to create a column variable `col` that we used as an index to extract characters from each column. In doing this, we were able to read the strings from *top to bottom* and *left to right*.

Encryption Method 2: Using a List of Strings

Let's consider another way to implement the encryption. This time, we will create a list of strings called `ciphertext` that will represent the columns of the grid. First, we will make the number of columns equal to the length of the key.

```
def columnar_encrypt2(plaintext, key):
    ciphertext = [''] * key
```

If our key is 3, then printing the `ciphertext` above will return a list of 3 empty strings `['', '', '']`. Next, the plan is to loop through each of these columns, taking text from `plaintext` and adding it to each of the items (columns) in the list. To do this, let's create a `for` loop that will loop through the range of the key. If our key is 3, then the range will be 0, 1, 2. We will use the variable `col` to represent each column.

```
def columnar_encrypt2(plaintext, key):
    ciphertext = [''] * key
    for col in range(key):
```

Next, what we want to add text to each item (column) in the list by looping through `plaintext`. If our `plaintext` is `Hello World` and our key is 3, then we want to add the first letter of `plaintext` to the `ciphertext` at index 0. After that, we want to move down `plaintext` the length of the key (3) and add the next letter to the `ciphertext` at index 3, and so on. We want to do this until we get to the end of the `plaintext`.

0	1	2	3	4	5	6	7	8	9	10
H	e	l	l	o		W	o	r	l	d

Taking every third letter, we get `HlWl`.

0	1	2	3	4	5	6	7	8	9	10
H			l			W			l	

So, how do we do this? One thought is to declare a variable `i`, set it to 0 (`i=0`), then loop through `plaintext`. We can then just add the character at index `i` to the `ciphertext`. We'll then increment it by the length of the key (3) to `i` until we get to the end of the `plaintext`. The problem with this is that (1) it will only loop once and (2) each loop will start at 0 return the same string `HlWl`.

So, instead, let's set `i` to the column number `i = col`. This way, we will start at 0, then 1, then 2, and so on. In addition, we'll say that `while i < len(plaintext)`, we want to add the character at index `i` to the `ciphertext` at index `col`. Since `col` starts at 0, the first column, we'll complete the entire first column before moving to the second column.

```
def columnar_encryptB(plaintext, key):
    ciphertext = [''] * key
```

```
    for col in range(key):
        i = col
        while i < len(plaintext):
            ciphertext[col] += plaintext[i]
            i = i + key
    return # ciphertext
```

Let's say our `plaintext` is 'ABCDEFGHIJK' and our key is 3. Here is how the function should build the `ciphertext`:

Col (index)	Ciphertext
0	['A', ' ', ' ']
0	['AD', ' ', ' ']
0	['ADG', ' ', ' ']
0	['ADGJ', ' ', ' ']
1	['ADGJ', 'B', ' ']
1	['ADGJ', 'BE', ' ']
1	['ADGJ', 'BEH', ' ']
1	['ADGJ', 'BEHK', ' ']
2	['ADGJ', 'BEHK', 'C']
2	['ADGJ', 'BEHK', 'CF']
2	['ADGJ', 'BEHK', 'CFI']

Finally, we want to return the `ciphertext`. If we were to return it as is, then we would return a list of strings. Since we would rather return a string, we can use the `join` method to join the strings in the list.

```
def columnar_encryptB(plaintext, key):
    ciphertext = [''] * key
    for col in range(key):
        i = col
        while i < len(plaintext):
            ciphertext[col] += plaintext[i]
            i = i + key
    return ''.join(ciphertext)
```

Let's test our function.

```
print(columnar_encryptB("ABCDEFGHIJK", 3))
# ADGJBEHKCFI
print(columnar_encryptB("Hello World", 4))
# Hore llWdlo
print(columnar_encryptB("Run, they are coming", 2))
# Rn hyaecmnu,te r oig
```

Decryption Using Python

Now that we have two functions to encrypt plaintext using the columnar transposition cipher, let's consider how to decrypt the ciphertext. To do this, we will create a function called `columnar_decrypt` that takes `ciphertext` and `key` as arguments and returns the plaintext.

```
def columnar_decrypt(ciphertext, key)
    plaintext = ""
    return # plaintext
```

Recall the general strategy noted by Gaines on how to decrypt the columnar cipher: "if the encipherer [...] has written the text in rows and taken it off by columns, then the decipherer must do the reverse: write his text by columns and take it off by rows" (*Cryptanalysis*, p.14). Let's try this approach. First, we are faced with some ciphertext "ADGJBEHKCFI" and given a key 3. Second, we are to write the ciphertext from *top to bottom* and *left to right*.

0	1	2
A		
D		
.		
.		

As we begin writing, recall that we need to know the number of rows. To get the number of rows, we can divide the length of the ciphertext by the length of the key and round up. At this point, we could fill out the grid by hand as follows:

0	1	2
A	B	C
D	E	F
G	H	I
J	K	

In Python, we can import that `math` module and use the `ceil` function to round up.

```
import math
def columnar_decrypt(ciphertext, key):
    plaintext = ""
    rows = math.ceil(len(ciphertext)/key)
    return #plaintext
```

Next, since we know the number of rows, we can iterate through the range of the number of rows. Each time we iterate, we want to start at the index of the row, get every `rows` number of characters, and then write it to the `plaintext`.

```
import math
def columnar_decrypt(ciphertext, key):
```

```
    plaintext = ""
    rows = math.ceil(len(ciphertext)/key)
    for row in range(rows):
        plaintext += ciphertext[row:len(ciphertext):rows]
    return plaintext
```

The reason we want every `rows` number of characters is because each character between that number will be written in the same column. Let's test our function.

```
print(columnar_decrypt("ADGJBEHKCFI", 3))
# ABCDEFGHIJK
print(columnar_decrypt("HlWleoodl r", 3))
# Hello World
print(columnar_decrypt("Rn hyaecmnu,te r oig", 2))
# Run, they are coming
```

References and Further Reading

1. Gaines, Helen Fouché. *Cryptanalysis: A Study of Ciphers and Their Solution*. New York: Dover Publications, 1956.
2. Sweigart, Al. *Cracking Codes with Python: An Introduction to Building and Breaking Ciphers*. San Francisco: No Starch Press, 2018.
3. Christensen, Chris. 2015. Columnar Transposition.
4. Neso Academy. 2021. Row Column Transposition Ciphering Technique
5. George Lasry, Nils Kopal & Arno Wacker (2014) Solving the Double Transposition Challenge with a Divide-and-Conquer Approach, Cryptologia, 38:3, 197-214, DOI: 10.1080/01611194.2014.915269
6. George Lasry, Nils Kopal & Arno Wacker (2016) Cryptanalysis of columnar transposition cipher with long keys, Cryptologia, 40:4, 374-398, DOI: 10.1080/01611194.2015.1087074
7. Goodin, Dan. 2020. Zociac Killer cipher is cracked after eluding sleuths for 51 years. Ars Technica
8. For a more powerful Python implementation, see CrypTool-Online. Simple Column Transposition. https://www.cryptool.org/en/cto/transposition