Assignment 1

Security and Cryptography (IN4191)

Francisco Manuel Colmenar Lamas
5196361

Enschede, 2019

Contents

List of Figures

List of Tables

List of Equations

1. CLASSICAL SYSTEMS

1.1. Section A

In order to know which method has been used to encrypt this ciphertext, the first step is to calculate the *Index of Coincidence*.

The *Index of Coincidence* indicates the similarity of the ciphertext to a plaintext. The closer the *Index of Coincidence* of the ciphertext is to the value of an English plaintext, which its value is 0.0667, the more probable that a transposition or a mono-alphabetic substitution cipher has been used in order to encrypt the plaintext.

$$IC = \sum_{i=A}^{Z} \frac{n_i(n_i - 1)}{N(N-1)} = 0.06121 \tag{1.1}$$

As it can be seen, the result of the Index of Coincidence for the ciphertext is very similar to the one of a plaintext written in English, $0.06121 \approx 0.0667$. As a consequence, it is probable that the plaintext has been ciphered with a transposition or mono-alphabetic substitution.

In order to obtain more information about the ciphertext, a *Frequency Analysis* is going to be applied to the ciphertext. The objective of this analysis is to study the distribution of the letters in the ciphertext.

Thanks to the *Frequency Analysis* useful information about the most used letters and bigrams in the ciphertext could be obtain in order to compare it the expected values of an English's plaintext.

In order to facilitate the understanding of the performed Frequency Analysis, the table representing the analysis to the ciphertext and the table with the frequencies of the English language are going to be displayed.

Letter Frequency			
Ciphertext		English language	
Letter	Frequency	Letter	Frequency
H	12.65%	E	12.7%
L	10.34%	T	9.1%
U	9.2%	A	8.2%
Z	8.05%	O	7.5%

Letter Frequency			
Ciphertext		English language	
Letter	Frequency	Letter	Frequency
P	6.9%	I	7.0%
V	6.9%	N	6.7%
S	5.75%	S	6.3%
Y	5.75%	H	6.1%
O	5.75%	R	6.0%
K	5.75%	D	4.3%
A	4.6%	L	4.0%
C	4.6%	C	2.8%
B	3.45%	U	2.8%
J	3.45%	M	2.4%
N	2.3%	W	2.4%
M	1.15%	F	2.2%
D	1.15%	G	2.0%
F	1.15%	Y	2.0%
T	1.15%	P	1.9%
E	0%	B	1.5%
G	0%	V	1.0%
I	0%	K	0.8%
Q	0%	J	0.15%
R	0%	X	0.15%
W	0%	Q	0.10%
X	0%	Z	0.07%

Table 1.1: Frequency Analysis performed to the ciphertext of Section 1 A

In order to compare both analysis the most frequent letters are being taking into consideration. If the most frequent letter of the ciphertext, *H*, is compared with the frequency of the most frequent letters of English the most probable is that *H* corresponds to *E* or *T*.

If *E* is considered as the XXXXX of *H*, the shift displacement of the ciphertext in relation to its original plaintext is *-3*. If the shift is applied to the ciphertext in order to decrypted the result is the following one.

"GLSGSPEXI AEW MRZIRXIH JSYV XLSYWERH CIEVW EKS MR E WQEPP ZMPPEKI MR LSRHYVEW ERH LEW XLVMZIH IZIV WMRGI"

As it is seen, this is not the plaintext belonging to the ciphertext. However, if the second most frequent letter of English is tried instead, *T* with frequency of 9.1%, the shift displacement should be *-7*. The result in this case is the following one.

"CHOCOLATE WAS INVENTED FOUR THOUSAND YEARS AGO IN A SMALL VILLAGE IN HONDURAS AND HAS THRIVED EVER SINCE"

Consequently, the plaintext has been encrypted with a Shift Cipher with a displacement of 7 letters.

In the case that the shift displacement was not found using this approach, an frequency analysis for the bigrams could be performed as well as calculating the statistical distance. However, as its displacement has already been found there is no need to calculate them in this case.

Furthermore, an implementation of the decryption process excluding the frequency analysis has been implemented in Python in order to gain a better understanding of its core calculations and processes. The source code of this implementation is annexed in the appendix of this report, page 25.

1.2. Section B

Firstly, in order to get some information about the ciphering method used in order to encrypt the message the *Index of Coincidence* is calculated.

$$IC = \sum_{i=A}^{Z} \frac{n_i(n_i - 1)}{N(N-1)} = 0.06782 \tag{1.2}$$

As it can be seen, the *Index of Coincidence* of the ciphertext is similar to the one of a plaintext written in English. As a consequence, it is probable that the encryption scheme used is a transposition cipher or a mono-alphabetic substitution.

In order to know if a transposition cipher, such as the one used in the previous section, or a mono-alphabetic substitution encryption schema has been used the *Frequency Analysis* needs to be performed.

If a *Frequency Analysis* is performed to the given ciphertext the result is the following one.

Letter Frequency			
Ciphertext		English language	
Letter	Frequency	Letter	Frequency
E	14.7%	E	12.7%
T	9.48%	T	9.1%
A	7.83%	A	8.2%
O	7.01%	O	7.5%
S	6.73%	I	7.0%

Letter Frequency			
Ciphertext		English language	
Letter	Frequency	Letter	Frequency
N	6.46%	N	6.7%
R	6.46%	S	6.3%
I	6.04%	H	6.1%
H	5.49%	R	6.0%
L	4.53%	D	4.3%
C	3.57%	L	4.0%
D	3.16%	C	2.8%
M	2.75%	U	2.8%
U	2.61%	M	2.4%
W	2.34%	W	2.4%
P	2.2%	F	2.2%
Y	1.79%	G	2.0%
F	1.65%	Y	2.0%
G	1.24%	P	1.9%
B	1.24%	B	1.5%
V	1.24%	V	1.0%
K	0.69%	K	0.8%
X	0.69%	J	0.15%
Q	0.14%	X	0.15%
J	0%	Q	0.10%
Z	0%	Z	0.07%

Table 1.2: Frequency Analysis performed to the ciphertext of Section 1 B

As it can be observed in the previous table, the letter frequency of both, the ciphertext and the English language is very similar. Consequently, it is highly probable that the encryption scheme used is a transposition cipher.

Furthermore, in order to get more information about the transposition cipher used in this ciphertext the end of it is going to be analyzed. At the end of the ciphertext there is a great amount of *"X"* characters which suggests us that these characters are not part from the original text but from the padding used in the cipher.

"...FIBNASENXSXDXIE"

This padding can disclose important information about the keylength used for encrypting the message. If the number of characters from the first occurrence of an "X" to the end of the text are counted the result is 7. This shows that it is probable that the keylength used is 7 too. Because of this, in order to discover the permutation used the following string is going to be the one analyzed.

"XSXDXIE"

The next step to be performed once that the keylength has been guessed is to find the permutation used. In order to find it the padding is going to be analyzed again. As the padding, the "X" in our case, are added to the end of the text it is known that the original message should be on the following way.

"(permutation of the letters S,D,I and E)XXX"

Consequently, the correct combination of the four remaining letters is needed to be found. In order to achieve it, try and error can be used in order to find a reasonable word or part of word which could be the one used. In this case the word *"SIDE"* is found. Consequently, the guessed original string should be similar to the next one.

"SIDEXXX"

The permutation for the four first letters is known at this point, (2,6,4,7,X,X,X). However, as the "X" characters can be permuted between then with no effect in the previous string it is needed to use other string in order to discover the this permutation. In order to achieve it the following string is going to be used.

"VCALIUT"

As before, using just try and fail into this string it can be found that the original string is the following one.

"CULTIVA"

Consequently, the permutation and key used for ciphering the plaintext is the following one.

$$\textbf{Key used, } \sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 6 & 4 & 7 & 5 & 1 & 3 \end{pmatrix}$$

The corresponding plaintext to the ciphertext is the following one.

"CULTIVATION CONSUMPTION AND CULTURAL USE OF CACAO WERE EXTENSIVE IN MESOAMERICA WHERE THE CACAO TREE IS NATIVE WHEN POLLINATED THE SEED OF THE CACAO TREE EVENTUALLY FORMS A KIND OF SHEATHOREAR TWENTY INCH LONG HANGING FROM THE TREE TRUNK IT SELF WITHIN THE SHEATHARE THIRTY TO FOURTY BROWN ISH RED ALMOND SHAPED BEANS EMBEDDED IN A SWEET VISCOUS PULP WHILE THE BEANS THEMSELVES ARE BITTER DUE TO THE ALKALOIDS WITHIN THEM THE SWEET PULP MAY HAVE BEEN THE FIRST ELEMENT CONSUMED BY HUMANS CACAO PODS THEMSELVES CAN RANGE IN A WIDE RANGE OF COLORS FROM PALE YELLOW TO BRIGHT GREEN ALL THE WAY TO DARK PURPLE OR CRIMSON THE SKIN CANAL SO VARY GREATLY SOME ARE SCULPTED WITH CRATERS OR WARTS WHILE OTHERS ARE COMPLETELY SMOOTH THIS WIDE RANGE IN TYPE OF PODS IS UNIQUE TO CACAOS IN THAT THEIR COLOR AND TEXTURE DOES NOT NECESSARILY DETERMINE THE RIPENESS OR TASTE OF THE BEANS INSIDE"

1.3. Section C

As in the previous sections, the *Index of Coincidence* as well as a *Frequency Analysis* needs to be accomplish in order to try to guess the encryption schema used.

First of all, the *Index of Coincidence* of the ciphertext is calculated.

$$IC = \sum_{i=A}^{Z} \frac{n_i(n_i - 1)}{N(N-1)} = 0.04135 \tag{1.3}$$

It can be observed that in this case the resulting Index of Coincidence of the ciphertext is not close to the value of a plaintext written in English, which is 0.0667. Consequently, it is highly probable that the encryption scheme used in this case is polyalphabetic, so a Vigenere encryption is the most probable encryption scheme used in this situation.

As a consequence of the encryption scheme being polyalphabetic, a *Frequency Analysis* performed to the actual ciphertext like in the first section would be useless. However, a different type of analysis can be accomplished in order to obtain more information about the ciphertext. This analysis is called the *Kasiski Test*.

The *Kasiski Test* is a method to obtain information from the ciphertext encrypted with a polyalphabetic substitution cipher such as the key length. The Kasiski test consists on searching for repeated sections of the ciphertext.

Once that these sections are found, the distance between consecutive repeated sequences are calculated and these distances are supposed to be multiples of the key length. After that all the distances between repeated sequence are found, the most probable key lengths are obtained.

If this method is applied to the ciphertext provided in the statement of this section the following result is obtained regarding the length between the repeated sequences.

Kasiski Test	
Length between sequences	Occurrences
3	619
9	607
2	313
6	269
4	151
12	131
5	114
15	105

Kasiski Test	
Length between sequences	Occurrences
7	102
8	74

Table 1.3: Kasiski Test performed to the ciphertext of Section 1 C

Consequently, the most probable key lengths for the ciphertext are 3, 9 and 2.

After that the key's length of the ciphertext has been guessed, the next step of the Kasiski Test is to divide the ciphertext into N different messages, N been the length of the key. This division is done taking the Nth letter of each segment in order to obtain N messages, each one encrypted with a different letter.

Consequently, once that the division has been accomplished, the problem to solve is just a mono-alphabetic cipher, as the one of the first section.

However, as the two most frequent lengths found with the Kasiski Test show very similar results, 619 and 607 respectively, a Frequency Analysis is going to be performed to their N messages in order to determine which key-length is the best candidate.

If a Frequency Analysis is performed to the set of N messages of both key-lengths, the result is the following one.

Kasiski Test		
Keylength	Message	Frequency Analysis Result
3	1	5.67% - 4.33%
	2	7.59% - 4.46%
	3	6.51% - 4.17%
9	1	6.9% - 3.94%
	2	9% - 4.5%
	3	7.73% - 4.35%
	4	8.96% - 4.48%
	5	7.46% - 3.98%
	6	7.22% - 3.61%
	7	6.12% - 4.59%

Kasiski Test		
Keylength	Message	Frequency Analysis Result
	8	7.32% - 3.9%
	9	8.08% - 4.55%

Table 1.4: Frequency Analysis comparison for the key length 3 and 9 of the ciphertext of Section 1 C

The column called "Frequency Analysis Result" represents the frequency of the letter with more occurrences in the left and the tenth in the right, in order to have and idea of the distribution of the frequency of the occurrences.

As it can be seen in the previous table, the keylength which has the most similar result in the Frequency Analysis to the Frequency of an English text is the keylength 9. Because of this, 9 is going to be the first choice in order to try to obtain the key.

Once that the Frequency Analysis has been accomplish in order to calculate the shift displacement of each of the 9 "messages" the key used to encrypt the original message is found. The key is the following one.

"MYTHOLOGY"

Once that the key is found, the original message can be easily recover shifting each of the messages the correct position and then merging then in the original order. The original plain text which is obtained is the following one.

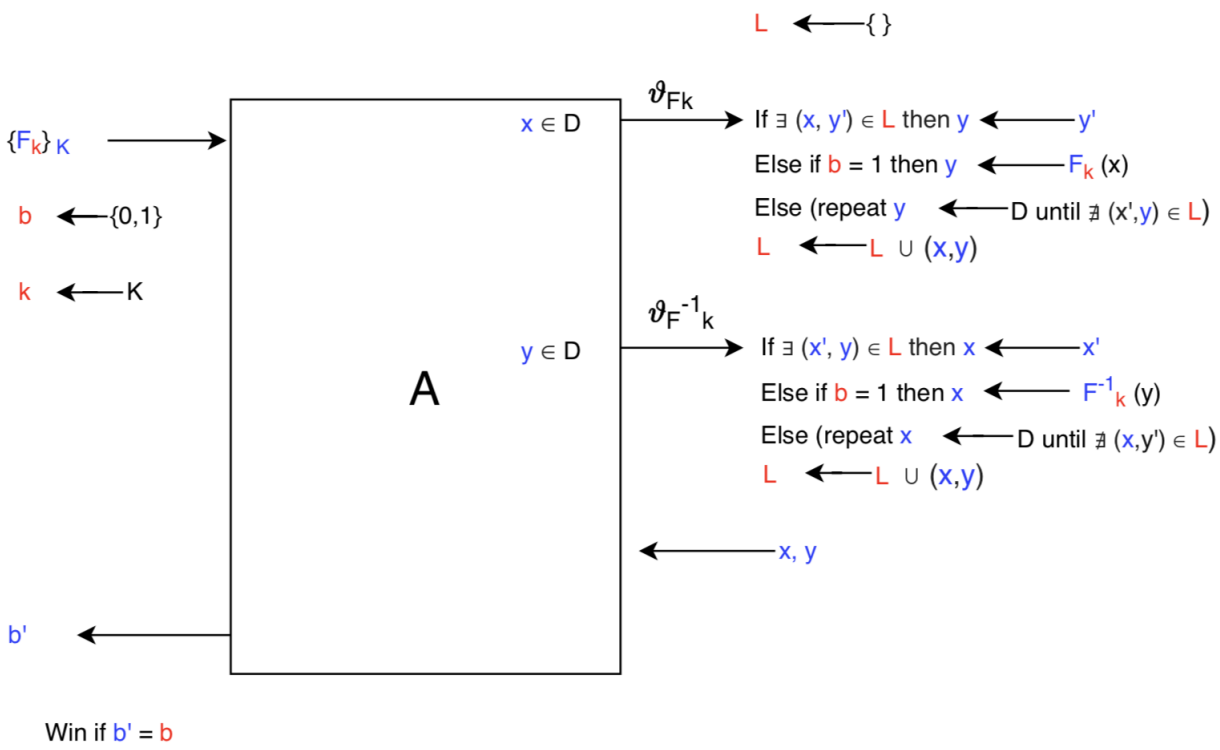
"THE SEVEN WONDERS OF THE WORLD OR THE SEVEN WONDERS OF THE ANCIENT WORLD IS A LIST OF REMARKABLE CONSTRUCTIONS OF CLASSICAL ANTIQUITY GIVEN BY VARIOUS AUTHORS IN GUIDEBOOKS OR POEMS POPULAR AMONG ANCIENT HELLENIC TOURISTS. ALTHOUGH THE LIST, IN ITS CURRENT FORM, DID NOT STABILISE UNTIL THE RENAISSANCE, THE FIRST SUCH LISTS OF SEVEN WONDERS DATE FROM THE SECOND CENTURY BC. THE ORIGINAL LIST INSPIRED INNUMERABLE VERSIONS THROUGH THE AGES, OFTEN LISTING SEVEN ENTRIES. OF THE ORIGINAL SEVEN WONDERS, ONLY ONE THE GREAT PYRAMID OF GIZA (ALSO CALLED THE PYRAMID OF KHUFU, AFTER THE PHARAOH WHO BUILT IT), THE OLDEST OF THE ANCIENT WONDERS REMAINS RELATIVELY INTACT. THE COLOSSUS OF RHODES, THE LIGHTHOUSE OF ALEXANDRIA, THE MAUSOLEUM AT HALICARNASSUS, THE TEMPLE OF ARTEMIS AND THE STATUE OF ZEUS WERE ALL DESTROYED. THE LOCATION AND ULTIMATE FATE OF THE HANGING GARDENS ARE UNKNOWN, AND THERE IS SPECULATION THAT THEY MAY NOT HAVE EXISTED AT ALL. THE LIST COVERED ONLY THE SCULPTURAL AND ARCHITECTURAL MONUMENTS OF THE MEDITERRANEAN

AND MIDDLE EASTERN REGIONS, WHICH THEN COMPRISED THE KNOWN WORLD FOR THE GREEKS. HENCE, EXTANT SITES BEYOND THIS REALM WERE NOT CONSIDERED AS PART OF CONTEMPORARY ACCOUNTS. THE PRIMARY ACCOUNTS, COMING FROM HELLENISTIC WRITERS, ALSO HEAVILY INFLUENCED THE PLACES INCLUDED IN THE WONDERS LIST. FIVE OF THE SEVEN ENTRIES ARE A CELEBRATION OF GREEK ACCOMPLISHMENTS IN THE ARTS AND ARCHITECTURE (THE EXCEPTIONS BEING THE PYRAMIDS OF GIZA AND THE HANGING GARDENS OF BABYLON). THE SEVEN WONDERS ON ANTIPATER'S LIST WON PRAISES FOR THEIR NOTABLE FEATURES, RANGING FROM SUPERLATIVES OF THE HIGHEST OR LARGEST OF THEIR TYPES, TO THE ARTISTRY WITH WHICH THEY WERE EXECUTED. THEIR ARCHITECTURAL AND ARTISTIC FEATURES WERE IMITATED THROUGHOUT THE HELLENISTIC WORLD AND BEYOND. THE GREEK INFLUENCE IN ROMAN CULTURE, AND THE REVIVAL OF GRECO-ROMAN ARTISTIC STYLES DURING THE RENAISSANCE CAUGHT THE IMAGINATION OF EUROPEAN ARTISTS AND TRAVELLERS. PAINTINGS AND SCULPTURES ALLUDING TO ANTIPATER'S LIST WERE MADE, WHILE ADVENTURERS FLOCKED TO THE ACTUAL SITES TO PERSONALLY WITNESS THE WONDERS. LEGENDS CIRCULATED TO FURTHER COMPLEMENT THE SUPERLATIVES OF THE WONDERS"

Furthermore, a partial implementation of this decryption process written in python has been accomplished. Its code is annexed in the appendix. The Frequency Analysis as well as the counting of repeated occurrences are not implemented, page 28.

2. SECURITY GAMES

The draw for the security game for a family of pseudo-random permutations F_k^-1, for an attacker A with access to two oracles ϑ_{F_k} and $\vartheta_{F_k}^-1$ is the following one.



Furthermore, the probability of the adversary is described in the following equation.

$$Adv^{PRP}_{\{F_K\}_K}(A) = 2 * |Pr[A^{\vartheta_{F_k}, \vartheta^-_{F_k}1} wins] - \frac{1}{2}| \tag{2.1}$$

In order to get a better understanding of the draw of the security game of above, some of the main concepts of it are going to be explained.

The objective of this game for A, the adversary, is to discover what is the selection bit chose by the other player. If the adversary guess it correctly she wins, otherwise she losses the game.

The player of the left, the one which chooses the selection bit, knows the secret key

and the selection bit. This information is unknown to the adversary.

Furthermore, both players know a set of pseudo-random permutations functions, however the choice of the left player to chose the function is done regarding the secret key. Consequently, the chosen function from the set is unknown to the adversary but not to the left player.

Nevertheless, the adversary has access to two different oracles: ϑ_{F_k} and $\vartheta^{-}_{F_k}1$. ϑ_{F_k} oracle gives the result of the permutation function using any input value which the adversary wants to use.

Moreover, the oracle $\vartheta^{-}_{F_k}1$ gives the original value of the function using any input. It has to be noted that as the functions are pseudo-random permutations, one to one functions, the domain of x and y is the same.

In order to both oracles to be consistent, they have to have a memory of previous results. This is needed in order to give consistent results to the adversary when the selection bit is equal to 0.

3. INFORMATION THEORETIC SECURITY

3.1. Section A: Calculate the probability of each plaintext conditioned on each ciphertext occurrence

In order to calculate the probability of each plaintext conditioned on each of the different ciphertexts the next formula is going to be used.

$$p(P = m|C = c) = \frac{p(P = m) * p(C = c|P = m)}{p(C = c)} \tag{3.1}$$

Furthermore, in order to calculate the probability of a ciphertext conditioned on a plaintext the next formula is going to be used.

$$p(C = c|P = m) = \sum_{k:m=d_k(c)} p(K = k) \tag{3.2}$$

In addition to this, the probability of each ciphertext needs to be calculated. This calculation is going to be performed with the following formula.

$$p(C = c) = \sum_{k:c\in C(k)} p(K = k) * p(P = d_k(c)) \tag{3.3}$$

$$p(1) = \sum_{k:1\in C(k)} p(K = k) * p(P = d_k(1)) = p(k1) * p(b) + p(k2) * p(a)+$$
$$p(k3) * p(c) + p(k4) * p(c) = 0.25$$

$$p(2) = \sum_{k:2\in C(k)} p(K = k) * p(P = d_k(2)) = p(k1) * p(d) + p(k2) * p(c)+$$
$$p(k3) * p(a) + p(k4) * p(b) = 0.25$$

$$p(3) = \sum_{k:3 \in C(k)} p(K = k) * p(P = d_k(3)) = p(k1) * p(c) + p(k2) * p(b) +$$
$$p(k3) * p(d) + p(k4) * p(d) = 0.214$$

$$p(4) = \sum_{k:4 \in C(k)} p(K = k) * p(P = d_k(4)) = p(k1) * p(a) + p(k2) * p(d) +$$
$$p(k3) * p(b) + p(k4) * p(a) = \frac{2}{7}$$

3.1.1. Plaintext A

First of all, the probability of each ciphertext conditioned in the plaintext A needs to be calculated.

$$p(1|A) = \sum_{k:A=d_k(1)} p(K = k) = p(k2) = 0.25$$

$$p(2|A) = \sum_{k:A=d_k(2)} p(K = k) = p(k3) = 0.25$$

$$p(3|A) = \sum_{k:A=d_k(3)} p(K = k) = 0$$

$$p(4|A) = \sum_{k:A=d_k(4)} p(K = k) = p(k1) + p(k4) = 0.5$$

Once that the probability of each ciphertext conditioned in A has been calculated the probability of A conditioned in each of the ciphertexts can be calculated.

$$p(A|1) = \frac{p(A) * p(1|A)}{p(1)} = \frac{2}{7}$$

$$p(A|2) = \frac{p(A) * p(2|A)}{p(2)} = \frac{2}{7}$$

$$p(A|3) = \frac{p(A) * p(3|A)}{p(3)} = 0$$

$$p(A|4) = \frac{p(A) * p(4|A)}{p(4)} = 0.5$$

3.1.2. Plaintext B

First of all, the probability of each ciphertext conditioned in the plaintext B needs to be calculated.

$$p(1|B) = \sum_{k:B=d_k(1)} p(K = k) = p(k1) = 0.25$$

$$p(2|B) = \sum_{k:B=d_k(2)} p(K = k) = p(k4) = 0.25$$

$$p(3|B) = \sum_{k:B=d_k(3)} p(K = k) = p(k2) = 0.25$$

$$p(4|B) = \sum_{k:B=d_k(4)} p(K = k) = p(k3) = 0.125$$

Once that the probability of each ciphertext conditioned in B has been calculated the probability of B conditioned in each of the ciphertexts can be calculated.

$$p(B|1) = \frac{p(B) * p(1|B)}{p(1)} = \frac{1}{7}$$

$$p(B|2) = \frac{p(B) * p(2|B)}{p(2)} = \frac{1}{7}$$

$$p(B|3) = \frac{p(B) * p(3|B)}{p(3)} = 0.166$$

$$p(B|4) = \frac{p(B) * p(4|B)}{p(4)} = 0.166$$

3.1.3. Plaintext C

First of all, the probability of each ciphertext conditioned in the plaintext C needs to be calculated.

$$p(1|C) = \sum_{k:C=d_k(1)} p(K = k) = p(k3) + p(k4) = 0.5$$

$$p(2|C) = \sum_{k:C=d_k(2)} p(K = k) = p(k2) = 0.25$$

$$p(3|C) = \sum_{k:C=d_k(3)} p(K = k) = p(k1) = 0.25$$

$$p(4|C) = \sum_{k:C=d_k(4)} p(K = k) = 0$$

Once that the probability of each ciphertext conditioned in B has been calculated the probability of C conditioned in each of the ciphertexts can be calculated.

$$p(C|1) = \frac{p(C) * p(1|C)}{p(1)} = \frac{6}{7}$$

$$p(C|2) = \frac{p(C) * p(2|C)}{p(2)} = \frac{3}{7}$$

$$p(C|3) = \frac{p(C) * p(3|C)}{p(3)} = 0.5$$

$$p(C|4) = \frac{p(C) * p(4|C)}{p(4)} = 0$$

3.1.4. Plaintext D

First of all, the probability of each ciphertext conditioned in the plaintext D needs to be calculated.

$$p(1|D) = \sum_{k:D=d_k(1)} p(K = k) = 0$$

$$p(2|D) = \sum_{k:D=d_k(2)} p(K = k) = p(k1) = 0.25$$

$$p(3|D) = \sum_{k:D=d_k(3)} p(K = k) = p(k3) + p(k4) = 0.5$$

$$p(4|D) = \sum_{k:D=d_k(4)} p(K = k) = p(k2) = 0.25$$

Once that the probability of each ciphertext conditioned in B has been calculated the probability of D conditioned in each of the ciphertexts can be calculated.

$$p(D|1) = \frac{p(D) * p(1|D)}{p(1)} = 0$$

$$p(D|2) = \frac{p(D) * p(2|D)}{p(2)} = \frac{1}{7}$$

$$p(D|3) = \frac{p(D) * p(3|D)}{p(3)} = 0.333$$

$$p(D|4) = \frac{p(D) * p(4|D)}{p(4)} = 0.125$$

3.2. Section B: Compute the entropy H(P|C = 4)

In order to calculate the entropy of a plaintext given the observation of the ciphertext 4 the next formula is needed to be used.

$$H(X|Y = y) = -\sum_{x} p(X = x|Y = y) * log_2 p(X = x|Y = y) \tag{3.4}$$

If this formula is applied to the ciphertext 4 the result is the following one.

$$H(P|4) = -\sum_{x} p(P = x|4) * log_2 p(P = x|4) = -(p(A|4) * log_2 p(A|4) +$$
$$p(B|4) * log_2 p(B|4) + p(C|4) * log_2 p(C|4) + p(D|4) * log_2 p(D|4)) = 1.305$$

3.3. Section C: Compute the entropy H(P|C)

In this case, in order to calculate the entropy of P conditioned in C the formula which is going to be used is the following one.

$$H(X|Y) = \sum_{y} p(Y = y) * H(X|Y = y) =$$
$$-\sum_{x}\sum_{y} p(Y = y) * p(X = x|Y = y) * log_2 p(X = x|Y = y) \tag{3.5}$$

In order to be easier to read the operations first H(X|Y = y) for all y is going to be calculated and then the final result will be calculated.

3.3.1. Compute the entropy H(P|1)

The entropy H(P|1) is going to be calculated using the following formula.

$$H(P|1) = -\sum_{x} p(P = x|1) * log_2 p(P = x|1) = -(p(A|1) * log_2 p(A|1) +$$
$$p(B|1) * log_2 p(B|1) + p(C|1) * log_2 p(C|1) + p(D|1) * log_2 p(D|1)) = 1.108$$

3.3.2. Compute the entropy H(P|2)

The entropy H(P|2) is going to be calculated using the following formula.

$$H(P|2) = -\sum_{x} p(P = x|2) * log_2 p(P = x|2) = -(p(A|2) * log_2 p(A|2) +$$
$$p(B|2) * log_2 p(B|2) + p(C|2) * log_2 p(C|2) + p(D|2) * log_2 p(D|2)) = 1.842$$

3.3.3. Compute the entropy H(P|3)

The entropy H(P|3) is going to be calculated using the following formula.

$$H(P|3) = -\sum_{x} p(P = x|3) * log_2 p(P = x|3) = -(p(A|3) * log_2 p(A|3) +$$
$$p(B|3) * log_2 p(B|3) + p(C|3) * log_2 p(C|3) + p(D|3) * log_2 p(D|3)) = 1.458$$

3.3.4. Compute the entropy H(P|4)

The entropy H(P|4) is going to be calculated using the following formula.

$$H(P|4) = 1.305$$

3.3.5. Compute the entropy H(P|C)

The final step to calculate the entropy H(P|C) is calculate the following summation using the previous calculated values.

$$H(X|Y) = \sum_{y} p(Y = y) * H(X|Y = y) = p(1) * H(X|1) +$$
$$p(2) * H(X|2) + p(3) * H(X|3) + p(4) * H(X|4) = 1.465$$

3.4. Section D: Is this scheme perfectly secure?

A scheme is perfectly secure only in the case that the ciphertext does not reveal any information about the plaintext. Consequently, a scheme with perfect secrecy has to comply with the following formula.

$$p(P = m|C = c) = p(P = m) \tag{3.6}$$

However, the scheme which has been analyzed in the previous sections do not fulfill this requirement. An example of this is just the case of A where the probability of p(A) is equal to $\frac{2}{7}$ but its probability conditioned in D is greater, p(A|4) = 0.5.

$$p(A|4) = 0.5 > p(A) = \frac{2}{7}$$

As a consequence of this situation the scheme has no perfect secrecy.

APPENDIX A: IMPLEMENTATION OF SHIFT DECODER IN PYTHON

The implementation of the Shift Decoder is mainly accomplished using four different files.

1. *caesar.py*: This file just contains the defining of the data Object which is going to be used in order to ease the implementation of the deciphering process.

```
class Caesar:


    def __init__(self):
        self.plaintext = ""
        self.ciphertext = ""
        self.shift = None

    def print_caesar(self):
        print(
            "Caesar text with:\nPlaintext: %s\nCiphertext: %s\nShift: %s" %
            (self.plaintext, self.ciphertext, self.shift))
```

Fig. 3.1. caesar.py file for Shift Decoder

2. *const.py*: In this file some useful constant values are stored. This organization of the constant values is specially useful when a new ciphertext is desired to be decrypted with a different shift, so only these variables are changed and not the core code.

```
CYPHERTEXT1 = "../../data/ciphertext1.txt"


CYPHER_MODE = 0
PLAINTEXT_MODE = 1


SPACE = " "


CAESAR_SHIFT = -7
```

Fig. 3.2. const.py file for Shift Decoder

3. *func.py*: This file contains several functions which are used in the decryption process. These functions are the following ones.

(a) *read_text*: This function just read a ciphertext or plaintext and stores it in the correct field of the object passed to it as an argument.

```
def read_text(text_object, filename, mode):
    """
    Read a given plain text and store it into the field "plaintext" of the text_object
    :param mode: Flag representing if what is going to be read is a ciphertext or a plaintext
    :param text_object: Object which is going to save the read plaintext
    :param filename: Name of the file which is going to be read
    :return: 0 if it has not happened no error. Otherwise -1 is returned
    """
    f = open(filename, "r")
    if f.mode == 'r':
        contents = f.read()
        if mode == const.CYPHER_MODE:
            text_object.ciphertext = contents
            return 0
        elif mode == const.PLAINTEXT_MODE:
            text_object.plaintext = contents
            return 0
        else:
            print("Error while reading the text [Incorrect mode code]")
            return -1
    else:
        print("Error while reading the text")
        return -1
```

Fig. 3.3. read_text function for Shift Decoder

(b) *shift_text*: In this case the function shifts the given text the amount of places as it is given in the parameter *shift*. The resulting shifted text is returned.

```
def shift_text(origin, shift):
    """
    Shifts each of the letters of the given text the amount of positions given
    :param origin: The text which is going to be shifted
    :param shift: The amount of positions the original text os shifted
    :return: The resulting text from the shifting of letter positions
    """
    # Defining the variable which will store the resulting text
    result = ""
    for letter in origin:
        # The space is not shifted
        if letter == const.SPACE:
            result += const.SPACE
        # Shift the letter
        else:
            pos = ord(letter)
            pos = (pos - 65 + shift) % 26 + 65
            result += chr(pos)
    return result
```

Fig. 3.4. shift_text function for Shift Decoder

4. *decypher_a.py*: This file is just the main file which calls to the functions previously described in order to decrypt the ciphertext.

```
# Create the object
text = Caesar()

# Read the ciphertext
if func.read_text(text, const.CYPHERTEXT1, const.CYPHER_MODE) == -1:
    exit(-1)

# Define the amount of places which the text is going to be shifted
text.shift = const.CAESAR_SHIFT

# Shifting the text
text.plaintext = func.shift_text(text.ciphertext, text.shift)
```

Fig. 3.5. decypher_a.py file for Shift Decoder

APPENDIX B: IMPLEMENTATION OF VIGENERE DECODER IN PYTHON

In this case, the implementation of the Vigenere decoder has been accomplished partially. The Frequency Analysis and the counting of occurrences during the Kasiski Test are not implemented.

The partial implementation consist of the reading of the ciphertext, obtaining the keylength from Kasiski (no implementation of the Test is done, it is only returned the previous calculated keylength) and dividing the ciphertext into N messages.

This implementation accomplished using three different files.

1. *vigenere.py*: Contains the data structure object in order to perform the decryption implementation as easy and intuitive as possible.

```
class Vigenere:


    def __init__(self):
        self.plaintext = ""
        self.ciphertext = ""
        self.keylength = None
        self.key = ""

    def print_vigenere(self):
        print(
            "Vigenere text with:\nPlaintext: %s\nCiphertext: %s\nKeylength: %s\nKey: %s" % (
                self.plaintext, self.ciphertext, self.keylength, self.key))
```

Fig. 3.6. vigenere.py file for Vigenere Decoder

2. *func.py*: This fie is the same as in the previous implementations. However, in this case only one method is being used.

    (a) *divide_text*: This method divides the given text, in this case ciphertext, into N different messages. The amount of messages to divide the original text is given by the parameter *"keylength"*.

```python
def divide_text(text, keylength):
    # Length of the text
    div_number = int(len(text) / keylength)
    # Calculate the number of letters per division of the text
    if len(text) % keylength != 0:
        div_number += 1

    # List with all the divisions of the text
    divisions = []

    # Add padding to the text
    if len(text) % keylength != 0:
        for i in range(len(text), len(text) + len(text) % keylength):
            text += " "

    # Iterate through the different divisions of the cipher text
    for i in range(0, keylength):
        aux = ""
        # Iterate through the ciphertext and add letters to each division
        for j in range(0, div_number):
            # Add the correct letter to the string
            aux += text[j * keylength + i]
        # Add the string to the array of divisions
        divisions.append(aux)
    return divisions
```

Fig. 3.7. divide_text function for Vigenere Decoder

3. *decypher_c.py*: This file just calls to the functions needed as well as it creates the needed data structure in order to perform the decryption.

```python
# Create the object
text = Vigenere()

# Obtain the key length
text.keylength = func.kasiski()

# Read the ciphertext
func.read_text(text, const.CYPHERTEXT3, const.CYPHER_MODE)

divisions = func.divide_text(text.ciphertext, text.keylength)
```

Fig. 3.8. decypher_c.py file for Vigenere Decoder