ELEC6242 Coursework Assignment: Cryptanalysis of Three Ciphers

Author: Rishin Amin
Student I.D: 26725479
ra8g14@soton.ac.uk
Personal Tutor: Alex Weddell
March 2018

Abstract

Three cipher texts are analysed using cryptanalysis techniques with two successfully solved. The odd numbered sections contain the plaintext for each cipher challenge and the even numbered sections provide a summary of the techniques used to solve the ciphers. The code used to crack/ analyse the ciphers is provided in the appendices.

1 Solution for Cipher 1

The team already is working to find asteroids that might be a threat to our planet, and while we have found 95 percent of the large asteroids near the Earth's orbit, we need to find all those that might be a threat to Earth,This Grand Challenge is focused on detecting and characterizing asteroids and learning how to deal with potential threats. We will also harness public engagement, open innovation and citizen science to help solve this global problem.

1.1 Key

KEY = **HJV**

2 Cipher 1 Cryptanalysis

2.1 Cipher Text

Aqz anvt jgynvkh dz fjytdup ov odum vzczyxdkb oojo trboc wl j ooazhc ov xpy yghwza, jik fcpuz dn chez mxpum 95 klaxlwo vo oon ghabl jnanmvryz wzha oon Zhaoo'b jykda, fz unzk cj mrik jgs ccvbz aqva vdnqo in v aqmljo ax Zhaoo,Ccpb Byjik Lchuglwbl rn mxxbbzk xi knollopwb hwy jqvyjxanmpidup vzczyxdkb vum gljmurin qjd cj knvs fdaq kvczucdhu ooazhcn. Dn rpug hunv qvywzzb kbkgpl zupvnnhlwo, vyzu riuxqhcdvw vum xpcdgni zldlwxl cj ongw bjsez aqdz pgvkvs ymvkglv.

2.2 Distinguishing the Type of Cipher

The cipher text contains feasible word lengths and allocation of spaces implying the use of a substitution cipher. This is corroborated by the repetition of some words, for example, "oon" and "vum".

The Index of Coincidence (IC) of the cipher text can be used to determine if this is a monoalphabetic or polyalphabetic substitution, this studies the probability of finding repeated letters in the text. Standard English text has an IC of 0.0667 [ref]. Using code found in Appendix A (function IC), the IC of the cipher text is calculated to be 0.0447. If the cipher text was a result of a monoalphabetic substitution cipher, it would have an IC of close to 0.0667, as the sum of all the normalised frequencies would be the same. However, this IC is indicative of a polyalphabetic cipher, specifically a Vigenere cipher [1].

2.3 Determining the Key Length

To solve this cipher the key must be found. The first step in finding the key is to determine and verify the key length. A probable key length is established using the Kasisky test (code in Appendix A). This involves finding the distances between repeated substrings of length at least three and calculating the Greatest Common Denominator (GCD) of these distances. This method is valid, as repetitions in the plaintext separated by multiples of the key length are encrypted in the same way. The ciphertext is first stripped of all punctuation and whitespace and turned to lowercase. Figure 1 shows the repeated substrings (of length at least three) and the distances between the stripped text, figure2 shows just the distances. The GCD of all the distances is one, however, closer inspection of the distances reveals an anomaly (247 - highlighted in the figure). The GCD without the anomaly is three, suggesting a probable key length of three.

```
>>> kasisky_test(strip_to_lc_alphabet("cipher1.txt"))
{'czy': [201], 'zhao': [6, 57, 51], 'haoo': [6, 57, 51], 'zhaoo': [6, 57, 51], 'anm': [117], 'oo
azhc': [222], 'zyxdk': [201], 'ong': [249], 'azhc': [222], 'oazhc': [222], 'zyxdkb': [201], 'vzc
zyxd': [201], 'vzczyxdkb': [201], 'qvy': [72], 'czyxd': [201], 'zczyxdkb': [201], 'zcz': [201],
'zaq': [204], 'hao': [6, 57, 51], 'vzc': [201], 'wov': [219], 'jik': [114], 'vzczyxdk': [201], '
czyxdkb': [201], 'lwo': [219], 'zczyx': [201], 'dup': [207], 'zczyxd': [201], 'zczyxdk': [201],
'oaz': [222], 'vzcz': [201], 'ooazh': [222], 'xdkb': [201], 'ooa': [222], 'aoo': [6, 57, 51], 'c
zyxdk': [201], 'oon': [21], 'xdk': [201], 'mxp': [247], 'vzczy': [201], 'lwov': [219], 'yxdk': [
201], 'yxdkb': [201], 'oazh': [222], 'zyxd': [201], 'ovo': [69], 'yxd': [201], 'azh': [222], 'vz
czyx': [201], 'ljo': [117], 'zczy': [201], 'vum': [87], 'uri': [69], 'ooaz': [222], 'zha': [6, 5
7, 51], 'zhc': [222], 'dkb': [201], 'upv': [75], 'czyx': [201], 'zyx': [201]}
```

Figure 1: Repeated substrings of length at least 3 with distances between repetitions

```
[201, 6, 57, 51, 6, 57, 51, 6, 57, 51, 117, 222, 201, 249, 222, 222, 201, 201, 201, 72, 201, 201
, 201, 204, 6, 57, 51, 201, 219, 114, 201, 201, 219, 201, 207, 201, 201, 222, 201, 222, 201, 222
, 6, 57, 51, 201, 21, 201, 247, 201, 219, 201, 201, 222, 201, 69, 201, 222, 201, 117, 201, 87, 6
9, 222, 6, 57, 51, 222, 201, 75, 201, 201]
```

Figure 2: Distances between repeated substrings (247 highlighted as anomaly)

[1] https://en.wikipedia.org/wiki/Index_of_coincidence

The Friedman test is used to statically reaffirm the key length. This requires splitting the cipher text into subtexts by alternating over each component in the key. Figure 3 shows the console output for each subtext and the IC for each subtext. The ICs for each subtext is close to that of standard English text (0.0667), corroborating the suggested key length of three. Each subtext is now a simple monoalphabetic substitution cipher that can be solved using frequency analysis to acquire the three letters in the key.

```
aatykzyuvuzykotolohvyhakpdhmullvohlavzhohoyaukmksvaanialahopykhllmbkklphjyapuzykulundksavuhohdph
vyzbpunlvuuhvupgzllowsazvsvl
qnjnhftpomcxbjrcjacxywjfunexmawonajnrwanabkfncrjcbqvqnqjxacbjluwrxbxnlwwqjnipcxbmjrqcnfqccuacnuu
qwbklpnwyrxcwmcnlwcnbeqpkykv
zvgvdjdodvzdoobwozopgziczczpkxoogbnmyzozojdzzjigczvdovmozocbicgbnxzioobyvxmdvzdvgmijjvdkzdoznrgn
vzkgzvhoziqdvxdidxjgjzdgvmg
[0.06805664830841857, 0.06031995803829005, 0.08356657337065174]
```

Figure 3: Three subtexts produced as a result of the Friedman method with the ICs for each subtext

2.4 Finding the Key

Each subtext corresponds to a letter in the key. A guess for each letter can be made by translating the most frequent letters in the cipher text to "E" (the most frequent letter in the English language). Figure 4 shows the frequencies for each letter in the alphabet for all three subtexts. Taking the frequencies greater than 10% (highlighted in yellow) and translating those to the letter "E" yields 16 estimates for the key.

Subtext1		Subtext2		Subtext3	
Letter	Frequency	Letter	Frequency	Letter	Frequency
h	10.4839	n	12.0968	z	16.2602
l	10.4839	c	11.2903	o	13.0081
a	9.6774	j	8.0645	d	11.3821
u	9.6774	w	8.0645	v	10.5691
v	8.8710	q	7.2581	g	8.9431
k	8.0645	b	6.4516	i	5.6911
o	7.2581	x	6.4516	j	5.6911
y	7.2581	a	5.6452	b	4.0650
p	5.6452	r	4.8387	c	4.0650
z	5.6452	u	4.0323	m	4.0650
s	3.2258	f	3.2258	x	4.0650
d	2.4194	k	3.2258	n	3.2520
m	2.4194	l	3.2258	k	2.4390
n	2.4194	m	3.2258	p	1.6260
b	1.6129	p	3.2258	y	1.6260
t	1.6129	y	2.4194	h	0.8130
g	0.8065	e	1.6129	q	0.8130
i	0.8065	o	1.6129	r	0.8130
j	0.8065	v	1.6129	w	0.8130
w	0.8065	h	0.8065	a	0.0000
c	0.0000	i	0.8065	e	0.0000
e	0.0000	t	0.8065	f	0.0000
f	0.0000	d	0.0000	l	0.0000
q	0.0000	g	0.0000	s	0.0000
r	0.0000	s	0.0000	t	0.0000
x	0.0000	z	0.0000	u	0.0000

Figure 4: Frequency analysis for each subtext

The IC for a decryption of the cipher text using each key estimate is given in figure 5. This table reveals that "HJV" is the most probable key, yielding an IC that is closest to 0.0667. Manually inspecting the decryption using this key reveals a reasonable plaintext with proper English.

Subtext Letters	Extracted Key	IC
hnz	DJV	0.0613
hno	DJK	0.0558
hnd	DJZ	0.0545
hnv	DJR	0.0589
hcz	DYV	0.0560
hco	DYK	0.0589
hcd	DYZ	0.0548
hcv	DYR	0.0589
lnz	HJV	0.0707
lno	HJK	0.0620
lnd	HJZ	0.0616
lnv	HJR	0.0563
lcz	HYR	0.0518
lco	HYV	0.0608
lcd	HYZ	0.0574
lcv	HYR	0.0518

Figure 5: ICs of decrypt using key estimate extracted from subtext frequency analysis of subtexts

3 Solution for Cipher 2

A 24 year old boy seeing out from the train's window shouted...with IC

"Dad, look the trees are going behind!"

Dad smiled and a young couple sitting nearby, looked at the 24 year old's childish behavior with pity, suddenly he again exclaimed...

"Dad, look the clouds are running with us!"

The couple couldn't resist and said to the old man...

"Why don't you take your son to a good doctor?" The old man smiled and said... "I did and we are just coming from the hospital, my son was blind from birth, he just got his eyes today."

Every single person on the planet has a story. Don't judge people before you truly know them. The truth might surprise you.

3.1 Key

KEY = **0x13, 0x20**

4 Cipher 2 Cryptanalysis

4.1 Distinguishing the Type of Cipher

The cipher text is in hex format, converting the hex to ASCII characters reveals nonsensical text, as shown in figure 6. The first character of the plaintext is given as a hint. The most probable operation to convert the hex char to the ASCII hex value representing the letter “A” is an XOR operation. This type of operation is common in cryptography (e.g. in the One Time Pad).

```
R!3YvAa|Lwq0j`EvI}G30fT3Fa0~gHvgRrI}�dI}D|W3S{OfTvD�-�WAw
                                                       3L|OxgHvgRvE`rRvtOzNtqE{I}D2�WAw`MzLvD3A}D3A3Y|U}G3C|UcLv`IgTzNt}ErRqY? O|KvD3AggHv!3YvAa|Lw�p
HzLwI`H3BvHrVzOadIgH3PzTj
                                  3SfDwE}Lj{E3AtAzN3EkC AzMvD�-�WAw
                                                                       3L|OxgHvpL|UwS3AaE3RfN}I}G3WzT{fS2�*GHvpOfP E3C|U D}�gaE`I`T3A}D3SrIwg03T{E30 D
3MrN�-�Hjw0}�gj0fgAxE3Y|Ua`0}g03A3G|OwwOpT|R,�t{E30 D3MrN3S~I EwrNw`AzD��ZwIwrNwdE3AaE3JfSgp0~I}G3Fa0~gHv{0`PzTrL?~Y3S|N3WrS3B I}D3Fa0~qIaT{
          3HvyU`T3G|T3HzS3EjE`gOwAj�-eeEaY3SzNtLvcEaS|N30}gHvcLrNvT3HrS3A3Sg0aY=W0}�gyUwGvcE|P E3BvF|Rvj0fgRfLjxN|W3T{E~3t{E3TaUgH3MzG{T3SfRcRzS
vj0f_
```

Figure 6: ASCII decode of hex file

4.2 Finding the First Part of the Key

A mathematical property of the XOR operation is that XORing the result with one of the operands gives the other operand. Consequently, XORing the hex value for “A” with the first letter of the cipher text reveals the operand used. In this case the key used is ‘0x13’. Applying this key to the rest of the cipher text reveals more nonsense (as shown in figure 7), however the text does appear to have a little more resemblance to the English language. Repetitions in the text (e.g. “JoF” and “JeRo_d”), suggest that this line of thought is correct, but there must be more elements in the key (see Appendix B for code).

```
>>> decrypt1("secret.hex", 0x13)
A2 JeRro_db\ysVeZnT \uG Ur\mt[etAaZn�swZnWoD @h\uGeW�
>
9DRds^i_eW RnW R JoFnT PoFp_esZtGi]gnVaAbJ,l\oXeW Rtt[e2 JeRro_d�sc[i_dZs[ Qe[aEi\rwZt[ CiGy @uWdVn_yhV RgRi] VxPlRi^eW�
>
9T[ec\uClV PoFlWn�trVsZsG RnW @aZdt\ GhV \lW ^a]�
>
�W[yd\n�ty\utRkV JoFrs\nt\ R To\dd\cGoA?�ghV \lW ^a] @mZlVda]dsRiW��IdZda]dwV RrV Yu@tc\mZnT Ur\mt[eh\sCiGa_,mJ @o] Da@ QlZnW Ur\mbZrGh [e
jFsG ToG [i@ VyVst\dRy�
>
vvVrJ @i]g_epVr@o] \nt[ep_a]eG [a@ R @t\rJ.D\n�tjFdTepVoClV QeUoAey\utAu_yk]oD GhVm ghV GrFt[ ^iThG @uApAi@ey\u
```

Figure 7: Decrypt using XOR operation with 0x13

4.3 Finding the Second Part of the Key

The second part of the key is found by cycling the key through the entire cipher text and decrypting the text 128 times using 128 different keys (standard ASCII characters). The frequency of every letter in the resulting decryptions is then calculated. This frequency distribution is compared with that of standard English text (given in figure 8[2]) using the Euclidean distance.

[2] https://www.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html

Letter	Frequency
a	8.167
b	1.492
c	2.782
d	4.253
e	12.702
f	2.228
g	2.015
h	6.094
i	6.966
j	0.153
k	0.772
l	4.025
m	2.406
n	6.749
o	7.507
p	1.929
q	0.095
r	5.987
s	6.327
t	9.056
u	2.758
v	0.978
w	2.36
x	0.15
y	1.974
z	0.074

Figure 8: ICs of decrypt using key estimate extracted from subtext frequency analysis of subtexts

The Euclidean distance provides a measure of similarity for two data sets and can be used to assess how similar the frequency distribution of letters are in the decrypted cipher text to that of standard English text.

Figure 9 shows the console output for calculating the Euclidean distances for every possible combination of the second key (Appendix B provides the code). The minimum Euclidean distance is found to be 6.0398 and appears twice in the list: When the key is '0x00' and '0x20'.

```
>>> find_second_key("secret.hex")
[6.03981137916642, 11.290953233674776, 14.666585241403737, 14.572374108716502, 11.62614485129021, 14.956191304113519, 11.664951030093203, 10.
998343876641389, 13.395884610522701, 11.673148388844618, 11.85619744672304, 14.334551491637654, 13.455763551768348, 12.19939286110613, 13.444
317005383155, 13.07035760718211, 14.784317455298625, 12.150681746787395, 12.289507516576895, 13.170003749742603, 14.555055301904133, 13.50447
3966634688, 13.549774029346587, 14.575390250188955, 14.628889412255132, 15.939589316115928, 12.145134982909191, 11.433303546430313, 11.952315
748944704, 13.586595343352482, 15.403657793691576, 15.492048970285811, 6.03981137916642, 11.290953233674776, 14.666585241403737, 14.572374108
716502, 11.62614485129021, 14.956191304113519, 11.664951030093203, 10.998343876641389, 13.395884610522701, 11.673148388844618, 11.85619744672
304, 14.334551491637654, 13.455763551768348, 12.19939286110613, 13.444317005383155, 13.07035760718211, 14.784317455298625, 12.150681746787395
, 12.289507516576895, 13.170003749742603, 14.555055301904133, 13.504473966634688, 13.549774029346587, 14.575390250188955, 14.628889412255132,
 15.939589316115928, 12.145134982909191, 11.433303546430313, 11.952315748944704, 13.586595343352482, 15.403657793691576, 15.492048970285811,
8.603201948428717, 15.968410782738266, 19.175562322693484, 18.3676096578639, 21.83117762539978, 14.246044169398884, 17.687027634340467, 19.39
5902788385914, 16.82031904715368, 18.106079035413913, 20.387276393581104, 19.642875826662834, 18.268934302167033, 19.56317957930031, 18.92297
2018158248, 17.94051118758084, 20.031334087175168, 20.07214472658615, 19.109439338783275, 18.588294100781898, 18.521242897158913, 20.91395499
0039283, 19.49492759926519, 19.087487862077463, 19.894513344140734, 20.522858517746723, 19.494692685112348, 8.421746120612696, 8.254731868491
747, 8.22004696178577, 8.13611317124547, 8.563874879974225, 8.603201948428717, 15.968410782738266, 19.175562322693484, 18.3676096578639, 21.8
3117762539978, 14.246044169398884, 17.687027634340467, 19.395902788385914, 16.82031904715368, 18.106079035413913, 20.387276393581104, 19.6428
75826662834, 18.268934302167033, 19.56317957930031, 18.922972018158248, 17.94051118758084, 20.031334087175168, 20.07214472658615, 19.10943933
8783275, 18.588294100781898, 18.521242897158913, 20.913954990039283, 19.49492759926519, 19.087487862077463, 19.894513344140734, 20.5228585177
46723, 19.494692685112348, 8.421746120612696, 8.254731868491747, 8.22004696178577, 8.13611317124547, 8.563874879974225]
Min Euclidean Distance is:
6.03981137917
Decrypting using:
[0, 32]
```

Figure 9: Euclidean distances of letter frequency distributions following a decrypt using all possible ASCII hex values (0-128) as second key

Manually inspecting the resulting decryption using the key values of '0x13' and '0x20' reveal the plaintext shown in section 3. The key values of '0x13' and '0x00' reveal something close to plaintext, but the extended

ASCII symbols are incorrect and the alternate letters are incorrectly capitalised. Therefore, the key consists of ‘0x13’ and ‘0x20’ and the decryption method is cycling the key over the cipher text and using the XOR operation.

5 Cipher 3 Plaintext

First stage possibility 1: “gaeifdseasoeivanwrytodattlaihrhuiiemeffinasoaetnwrssefdatbfhhrhoimremetrmifrhe”

First stage possibility 2: “dbfjegpfbplfjubmtqzwlgbwwobjkqkvjjfnfeejmbplbfwmtqppfegbwaekkqkljnqfnfwqnjeqkf”

6 Cipher 3 Cryptanalysis

6.1 Distinguishing the Type of Cipher

This cipher contains multiple non-alphabetic characters and we know that the final solution contains only alphabetic characters. This suggests that, like in the previous cipher, an operation is used to transform the cipher text. This is most likely an XOR operation.

6.2 Brute Forcing the First Key

Code has been written (found in Appendix C) to calculate the index of coincidence for every combination of keys from length one to two ranging from 0-128 for each part of the key. Figure 10 shows the results for the combinations that produced an IC greater than 0.65. These results were filtered down further by only investigating the keys that produce a plaintext that only has alphabetic characters. Figure 11 shows the result of this filtering, leaving eight key combinations to investigate. Figure 12 shows the decrypted text produced using these keys. This console output shows that there are two distinct texts produced by these keys, with identical combinations having different capitalisation of letters, this explains the identical ICs produced by the keys.

Key1	Key2	IC
9	23	0.06527
9	55	0.06527
10	20	0.06527
10	52	0.06527
11	21	0.06527
11	53	0.06527
14	16	0.06527
14	48	0.06527
41	23	0.06527
41	55	0.06527
42	20	0.06527
42	52	0.06527
43	21	0.06527
43	53	0.06527
46	16	0.06527
46	48	0.06527

Figure 10: Key combinations producing text with an IC higher than 0.065

```
>>> brute_force_keys()
0.0652680652681
9
23
0.0652680652681
9
55
0.0652680652681
10
20
0.0652680652681
10
52
0.0652680652681
41
23
0.0652680652681
41
55
0.0652680652681
42
20
0.0652680652681
42
52
```

Figure 11: Key combinations producing text with an IC higher than 0.065 and only alphabetic

```
>>> decrypt2(9,23)
'GAEIFDSEASOEIVANWRYTODATTLAIHRHUIIEMEFFINASOAETNWRSSEFDATBFHHRHOIMREMETRMIFRHE'
>>> decrypt2(9,55)
'GaEiFdSeAsOeIvAnWrYtOdAtTlAiHrHuIiEmEfFiNaSoAeTnWrSsEfDaTbFhHrHoImReMeTrMiFrHe'
>>> decrypt2(10,20)
'DBFJEGPFBPLFJUBMTQZWLGBWWOBJKQKVJJFNFEEJMBPLBFWMTQPPFEGBWAEKKQKLJNQFNFWQNJEQKF'
>>> decrypt2(10,52)
'DbFjEgPfBpLfJuBmTqZwLgBwWoBjKqKvJjFnFeEjMbPlBfWmTqPpFeGbWaEkKqKlJnQfNfWqNjEqKf'
>>> decrypt2(41,23)
'gAeIfDsEaSoEiVaNwRyToDaTtLaIhRhUiIeMeFfInAsOaEtNwRsSeFdAtBfHhRhOiMrEmEtRmIfRhE'
>>> decrypt2(41,55)
'gaeifdseasoeivanwrytodattlaihrhuiiemeffinasoaetnwrssefdatbfhhrhoimremetrmifrhe'
>>> decrypt2(42,20)
'dBfJeGpFbPlFjUbMtQzWlGbWwObJkQkVjJfNfEeJmBpLbFwMtQpPfEgBwAeKkQkLjNqFnFwQnJeQkF'
>>> decrypt2(42,52)
'dbfjegpfbplfjubmtqzwlgbwwobjkqkvjjfnfeejmbplbfwmtqppfegbwaekkqkljnqfnfwqnjeqkf'
```

Figure 12: Decrypted texts using keys found in figure 11

Therefore, the first part of this decryption has a key of either '(42, 52)' or '(41, 55)'. The text does not make sense suggesting that there is a second part to this decryption.

6.3 Solving the Second Part of the Cipher

Figure 13 shows the results of a frequency analysis on the texts produced using the keys found in the previous section. The high index of coincidences and comparable frequency distribution to that of figure 8 suggest that the next stage in decrypting this cipher is a monoalphabetic substitution or transposition cipher. Another point of note is that frequency distributions shown in figure 13 imply that one text is a monoalphabetic substitution of the other. Consequently, if the next stage is to decrypt the text using a substitution cipher, the text used does not matter. However, if the next stage is to decrypt the text using a transposition cipher, the text used will make a difference.

Letter	Key = (42,52)	Key=(41,55)
	Frequency	Frequency
a	1.282	10.256
b	10.256	1.282
c	0.000	0.000
d	1.282	3.846
e	7.692	12.821
f	12.821	7.692
g	3.846	1.282
h	0.000	7.692
i	0.000	10.256
j	10.256	0.000
k	7.692	0.000
l	5.128	1.282
m	3.846	5.128
n	5.128	3.846
o	1.282	5.128
p	6.410	0.000
q	8.974	0.000
r	0.000	8.974
s	0.000	6.410
t	2.564	7.692
u	1.282	1.282
v	1.282	1.282
w	7.692	2.564
x	0.000	0.000
y	0.000	1.282
z	1.282	0.000

Figure 13: Frequency analysis of texts produced using keys found in section 6.2

A Code for Cracking Cipher 1

```
from __future__ import division
import re
from fractions import gcd
from string import ascii_lowercase
import math
from itertools import cycle
import string
import csv

alphabet = "abcdefghijklmnopqrstuvwxyz"

# Frequency of all letters, in the English language, in alphabetic order
english_freqs = [8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015, 6.094, 6.966, 0.153, 0.772, 4.025, 2.406,
                                6.749, 7.507, 1.929, 0.095, 5.987, 6.327, 9.056, 2.758, 0.978, 2.360, 0.150, 1.974, 0.074]

probable_keys = ["DJV", "DJK", "DJZ", "DJR", "DYV", "DYK", "DYZ", "DYR", "HJV", "HJK", "HJZ", "HJR", "HYR", "HYV", "HYZ", "HYR"]

def strip_to_lc_alphabet(cipher_text_file):
        # """
        # Reads the cipher text from a text file (name given in input parameter),
        # removes the whitespace and outputs the cipher text in lowercase format.
        # """
        with open(cipher_text_file, "r") as f:
                for line in f:
                        cipher_text = re.sub("[^a-zA-Z]+", "", line).lower()
        return cipher_text

def remove_punctuation(string):
        # """
        # Remove the punctuation from a string - for finding spaces when decrypting
        # """
        return re.sub("[^a-zA-Z\s]+", "", string).lower()

def IC(cipher):
```

```
        # """
        # Use the index of coincidence work out what kind of cipher this is.
        # """
        ic =[]
        length = len(cipher)
        for letter in ascii_lowercase:
                n = cipher.count(letter)
                ic.append((n*(n -1))/(length*(length-1)))
        return sum(ic)


def kasisky_test(cipher):
        # """
        # Performs the kasisky test on the cipher text to find a probable key length.
        #- find distances between repeated substrings of length at least 3.
        #- start with three and increse length until no more repetitions found.
        #- find distance between pairs.
        #- return as dictionary with sequence and distances as key value pairs.
        # """
        substrDistances = {}
        substrlen = 3
        repfound = True
        while(repfound == True):
                # iterate from 0 to end of message minus subs string length
                repfound = False
                for i in range(0, len(cipher)-substrlen):
                        # substring to search for in message
                        substr = cipher[i:i+substrlen]
                        # search for substr in remainder of message
                        for j in range(i+substrlen, len(cipher)-substrlen):
                                # repeated substr found
                                if cipher[j:j+substrlen] == substr:
                                        repfound = True
                                        if substr not in substrDistances:
                                                substrDistances[substr] = []
                                        substrDistances[substr].append(j-i)
                substrlen = substrlen + 1
        return substrDistances

def extract_prob_key_length(substrDistances):
```

```
        # """
        # Determines the probable key length from the distances between repeated substrings
        # - calculates the greatest common denominator (GCD) of all the distances
        # """
        dists = [val for dists in substrDistances.values() for val in dists]
        print dists
        print reduce(lambda x,y:gcd(x,y),dists)


def friedman(cipher, guess):
        # """
        # Statistically reaffirm the key length using IC (Friedman test)
        # """
        ic = []
        matrix = [cipher[i::guess] for i in range(guess)]
        for row in matrix:
                print row
                ic.append(IC(row))
        print ic
        return matrix

def vigenere_decrypt(cipher, key):
        # """
        # Decrypt a cipher text using a given key.
        # """
        with open(cipher,'r') as f:
                cipher_text = f.read()
        spaces = []
        # get index of all spaces in orginal cipher text
        for i, c in enumerate(remove_punctuation(cipher_text)):
                if c==' ':
                        spaces.append(i)
        cipher_no_spaces = ''.join(remove_punctuation(cipher_text).split())
        pairs = zip(cipher_no_spaces, cycle(key.lower()))
        result = ''
        for pair in pairs:
                # difference in indexes between key char and cipher text char
                diff = alphabet.index(pair[0]) - alphabet.index(pair[1])
                # modulo 26 of the difference position in the alphabet
                result += alphabet[diff%26]
        # add spaces to plain text
```

```python
        for space in spaces:
                result = result[:space] + " " + result[space:]
        return result

def f_analysis(subtexts):
        # """
        # Performs frequency analysis on every letter in each subtext
        # """
        freqs = []
        subtext_freqs = []
        top_freqs  = []
        for i, text in enumerate(subtexts):
                fa =[]
                length = len(text)
                for letter in ascii_lowercase:
                        n = text.count(letter)
                        fa.append((n/length)*100)
                freqs.append(fa)
                subtext_freqs.append(zip(alphabet,fa))
                top_freqs.append([x for x in subtext_freqs[i] if x[1] >= 10.0])
        print top_freqs
        print freqs
        with open("output.csv", 'wb') as csvfile:
                fwriter = csv.writer(csvfile)
                for subtext_freq in subtext_freqs:
                        for f in subtext_freq:
                                fwriter.writerow(f)
        return subtext_freqs

def test_probable_keys(keys):
        # """
        # Decrypts and calculates the IC for all of the probable keys.
        # """
        ic = []
        for key in keys:
                ic.append(IC(vigenere_decrypt("cipher1.txt", key)))
        combo = zip(keys, ic)
        with open("output.csv", 'wb') as csvfile:
                fwriter = csv.writer(csvfile)
                for val in combo:
                        fwriter.writerow(val)
        return combo
```

B Code for Cracking Cipher 2

```python
from __future__ import division
import re
from string import ascii_lowercase
import math
from math import sqrt

# Frequency of all letters, in the English language, in alphabetic order
english_freqs = [8.167, 1.492, 2.782, 4.253, 12.702, 2.228, 2.015, 6.094, 6.966, 0.153, 0.772, 4.025, 2.406,
                                6.749, 7.507, 1.929, 0.095, 5.987, 6.327, 9.056, 2.758, 0.978, 2.360, 0.150, 1.974, 0.074]

def decrypt1(hex_file, key):
    # """
    # Decode hex file to ascii charachters and XOR with supposed key.
    # """
    with open(hex_file) as fp:
        hex_list = ["{0:2x}".format(ord(c)) for c in fp.read()]
    decoded = hex_list
    for i in range(0,len(hex_list)):
        decoded[i] = chr(int(hex_list[i], 16))# ^ key)
    print ''.join(decoded)
    #return hex_string

def euclidean_dis(data_set1, data_set2):
    # """
    # Calculates the euclidean distance between two datasets
    # """
    diffs_squared = []
    for i, data in enumerate(data_set1):
        diffs_squared.append(pow((data - data_set2[i]),2))
    return sqrt(sum(diffs_squared))


def FA(cipher):
    # """
    # Compares frequency of each letter in text with that of the english language and returns the euclidean distance
    # between both sets of frequencies
    # """
```

```
    cipher = ''.join(cipher.split()) # remove whitespace
    cipher = re.sub("[^a-zA-Z\s]+", "", cipher).lower()
    fa =[]
    length = len(cipher)
    for letter in ascii_lowercase:
        n = cipher.count(letter)
        fa.append((n/length)*100)
    return fa

def decrypt2(hex_file, key1, key2):
    # """
    # Decrypts the cipher text using two input keys to XOR with alternatively.
    # """
    key_ic = []
    with open(hex_file, 'r') as fp:
        hex_list = ["{0:2x}".format(ord(c)) for c in fp.read()]
    decoded = hex_list
    for i in range(0,len(hex_list), 2):
        decoded[i] = chr(int(hex_list[i], 16) ^ key1)
    for i in range(1,len(hex_list), 2):
        decoded[i] = chr(int(hex_list[i], 16) ^ key2)
    #print ''.join(decoded)
    return ''.join(decoded)

def find_second_key(hex_file):
    # """
    # Use euclidean distances for every key combination from 0-128 to find most likely possibility for second key.
    # euclidean distance is used to determine the most appropriate key
    # """
    edists = []
    for key in range(0,128):
        edists.append(euclidean_dis(FA(decrypt2(hex_file, 0x13, key)), english_freqs))
    print edists
    print "Min Euclidean Distance is:"
    print min(edists)
    print "Decrypt using:"
    keys = [i for i, x in enumerate(edists) if x == min(edists)]
    print keys
```

C Code for Cracking Cipher 3

```
from __future__ import division
import re
from fractions import gcd
from string import ascii_lowercase
import math
from itertools import cycle
import string
import csv

cipher3 = "NVL^OSZRHDFR@AHY^EPCFSHC][H^AEAB@^LZLQO^GVZXHR]Y^EZDLQMV]UO_AEAX@Z[
    RDR]ED^OEAR"

def decrypt2(key1, key2):
        # """
        # Decrypts the cipher text using two input keys to XOR with
            alternatively.
        # """
        decoded = list(cipher3)
        key_ic = []
        for i in range(0,len(cipher3), 2):
                decoded[i] = chr(ord(cipher3[i]) ^ key1)
        for i in range(1,len(cipher3), 2):
                decoded[i] = chr(ord(cipher3[i]) ^ key2)
        # print decoded
        return ''.join(decoded)

def decrypt1(key):
        # """
        # Decrypts the cipher text using two input keys to XOR with
            alternatively.
        # """
        decoded = list(cipher3)
        key_ic = []
        for i in range(0,len(cipher3)):
                decoded[i] = chr(ord(cipher3[i]) ^ key)
        # print decoded
        return ''.join(decoded)


def FA(cipher):
        # """
```

```
    # Compares frequency of each letter in text with that of the english language and returns the euclidean distance
    # between both sets of frequencies
    # """
    cipher = ''.join(cipher.split()) # remove whitespace
    cipher = re.sub("[^a-zA-Z\s]+", "", cipher).lower()
    fa =[]
    length = len(cipher)
    for letter in ascii_lowercase:
        n = cipher.count(letter)
        fa.append((n/length)*100)
    with open("output4.csv", 'wb') as csvfile:
        fwriter = csv.writer(csvfile)
        fwriter.writerow(fa)
    return fa

def IC(cipher):
    # """
    # works out the index of coincidence for a given input string
    # """
    ic =[]
    cipher = cipher.lower()
    length = len(cipher)
    for letter in ascii_lowercase:
        n = cipher.count(letter)
        ic.append((n*(n -1))/(length*(length-1)))
    return sum(ic)

def brute_force_key():
    # """
    # Cycles through all key combinations for one key and returns decrypts with the highest IC
    # """
    high_ics = []
    for i in range(0,128):
        decrypt = (decrypt1(i))
        if decrypt.isalpha():
            ic = IC(decrypt)
            if ic >= 0.065:
                high_ics.append([i])
                print ic
                print i

```

```
def brute_force_keys():
    # """
    # Cycles through all key combinations for two keys and returns the pairs with the highest IC
    # """
    high_ics = []
    for i in range(0,256):
        for j in range(0,256):
            decrypt = (decrypt2(i,j))
            if decrypt.isalpha():
                ic = IC(decrypt)
                if ic >= 0.065:
                    high_ics.append([i,j])
                    print ic
                    print i
                    print j
    with open("output3.csv", 'wb') as csvfile:
        fwriter = csv.writer(csvfile)
        for ic in high_ics:
            fwriter.writerow(ic)
```