# Project Report – Part 1
# Cryptanalysis of substitution ciphers

(CS 6903 Applied Cryptography)

By

***Rakesh Sukla (rrs402)***

***Janhavi Bagwe (jsb643)***

## Table of Contents

1. Team Members………………………………………………………………………… 3
2. Project Tasks……………………………………………………………………………3
3. Assumptions…………………………………………………………………………..3
4. Substitution ciphers…………………………………………………………...............3
5. Algorithm……………………………………………………………………………..4
   Dictionary 1…………………………………………………………………………...5
   Dictionary 2…………………………………………………………………………...5
6. Execution Test Results………………………………………………………………..6
7. Code…………………………………………………………………………………...7

## 1. Team Members

Rakesh Sukla (rrs402)
Janhavi Bagwe (jsb643)

## 2. Project Tasks

We started with reading, understanding and brainstorming the project requirement details. We scripted a brute force algorithm to decrypt the given cipher text. We found the complexity for such an algorithm was too high to execute in 3 minutes. We tried to encrypt sample text with small key length, compared and analyzed the plaintext, cipher text to get this more efficient solution which does not require any combinations of keys to be considered. We brainstormed the problem definition together and then divided the work equally.

## 3. Assumptions

1. The input key set will have unique rotations.
2. The words from dictionary 2 can be repeated for a valid plaintext.

## 4. Cryptanalysis of Substitution ciphers

- A substitution cipher is a method of encrypting units of plaintext to be replaced with cipher text following an encryption scheme where the units may be single letters, pairs of letters, triplets of letters or a combination of all. The cipher text is decrypted by performing the inverse substitution.
- The number of keys possible with the substitution cipher is higher, around 2^88 possible keys.
- Substitution ciphers can be deciphered by exhaustively searching through key space for the key that produces the decrypted text most closely resembling meaningful word, but an efficient algorithm would be to exploit patterns and redundancy in the patterns to significantly narrow their search.
- The substitution cipher which operates on single letters, it is termed a simple substitution cipher and that operates on larger groups of letters is termed poly-graphic. A **mono-alphabetic** cipher uses fixed substitution over the entire message, whereas a **polyalphabetic** cipher uses a number of substitutions at different positions in the message.

- This encryption type varies with what kind of substitution is considered. We have
  **Simple Substitution**:
  Entire plaintext is rotated by a fixed key.
  **Homophonic Substitution**:
  Single plaintext letters are replaced by different cipher text letters. They are generally much more difficult to break than standard substitution ciphers.
  **Polygraphic substitution**:
  Pairs of letters are substituted is bigraphic.
  **One-Time Pad:**
  Plaintext is paired with a random secret key and each character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition.
- **Cryptanalysis:**
  - Deciphering the substitution ciphers starts by making an initial guess about what the key is, and is generally based on analysis of the ciphertext, partial knowledge of the key or purely random.
  - The algorithm then uses this guess as a key to decrypt the ciphertext.
  - The resulting text is probably non-readable, but its contents will have a certain similarity to the expected language of the plaintext depending on how many correct symbols there were in the guess in the first place.
  - The more correct symbols in the assumed key, the more quickly the algorithm will converge to a solution.
  - If the contents of the new resulting text are closer to the expected language than those of the previously decrypted text, we keep the new key for the next iteration, if not the old one.

## 5. Algorithm - Decrypt an L-symbol cipher text

**INPUT:**

1. 100 characters cipher text (c) which includes {<space>, a, b … z} only.
2. No of keys symbols (t) used to encrypt the given cipher text, maximum up to 18 keys preferred.
3. Dictionary 1 - Plaintexts obtained as a sequence of space-separated English words.
4. Dictionary 2 – set of English words

**OUTPUT:**

The most probable plaintext from Dictionary 1 or Dictionary 2.

**ALGORITHM:**

**For Dictionary 1:**

```
1. Import the Dictionary 1 in memory.
2. Read the next 100 character sentence from Dictionary 1, say p
3. Perform an operation (Cipher Text (c) – Plaintext (p)) to get a string of 100 numbers.
4. Calculate the number of Unique Numbers in this string, say n.
5. If n == t,
      (Yes)
       If (The unique symbols are repeated as much as t)
       (If t=5, L=100, there must be 5 unique symbols repeating 20 times)
             (Yes)
             This is the most likely required plaintext, we print it and terminate the
             program.

6. If n != t, continue from step 2
7. If no matches are found in Dictionary 1, we check Dictionary 2.
```

**For Dictionary 2:**

We consider a Cartesian product of all words in the dictionary as multiple possible plaintext (permutations)

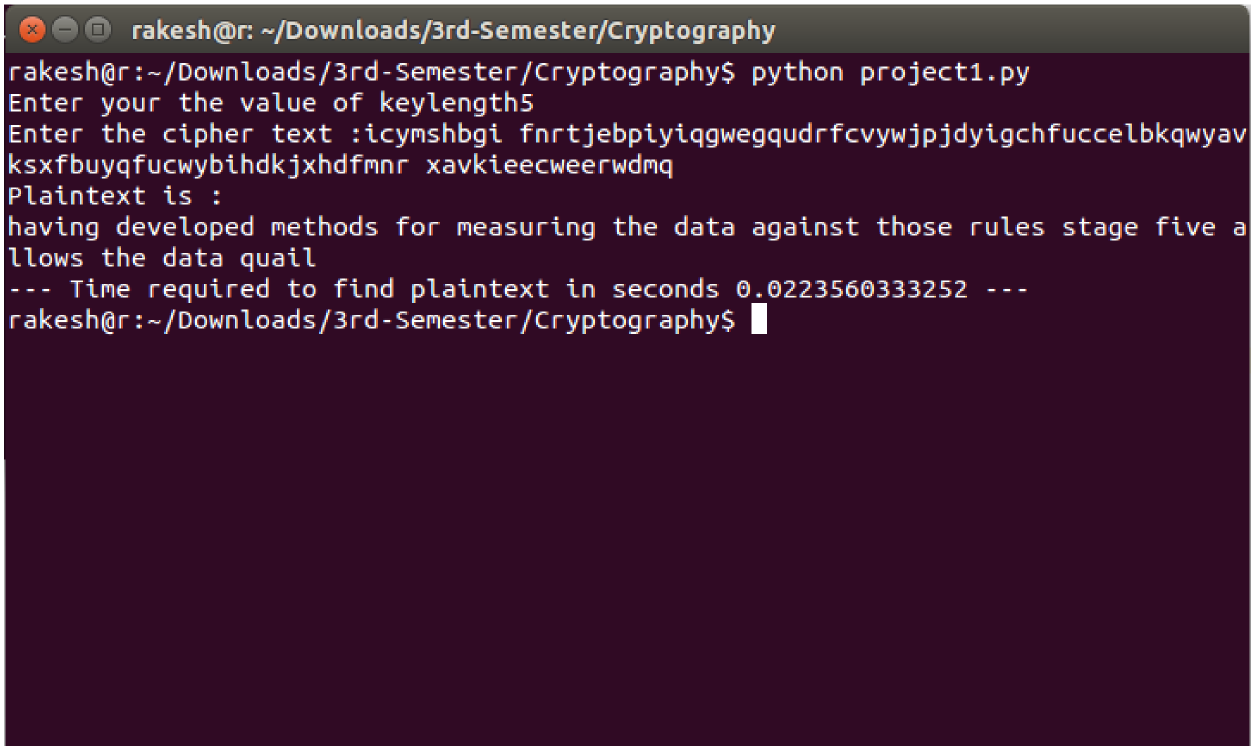
```
1. Import Dictionary 2 in memory.
2. Read next word from the Dictionary 2.
3. Append the word to a string, say s
4. Perform the operation (Cipher Text – Plain text).
5. Calculate the number of Unique Numbers in this string, say n
   If n <= t continue steps 1 through 4.

   If n > t, discard this and go to next permutation.
6. When the string append is of 100 characters
   If n == t,
      (Yes)
       If (The unique characters are repeated as much as t)
       (If t=5, L=100, there must be 5 unique symbols repeating 20 times)
          (Yes)
             This is the most likely required plaintext, we print it and terminate the
             program.

   If n != t, discard this and go to next permutation
   Backtrack if the current permutation no longer is the probable candidate for plaintext.
```

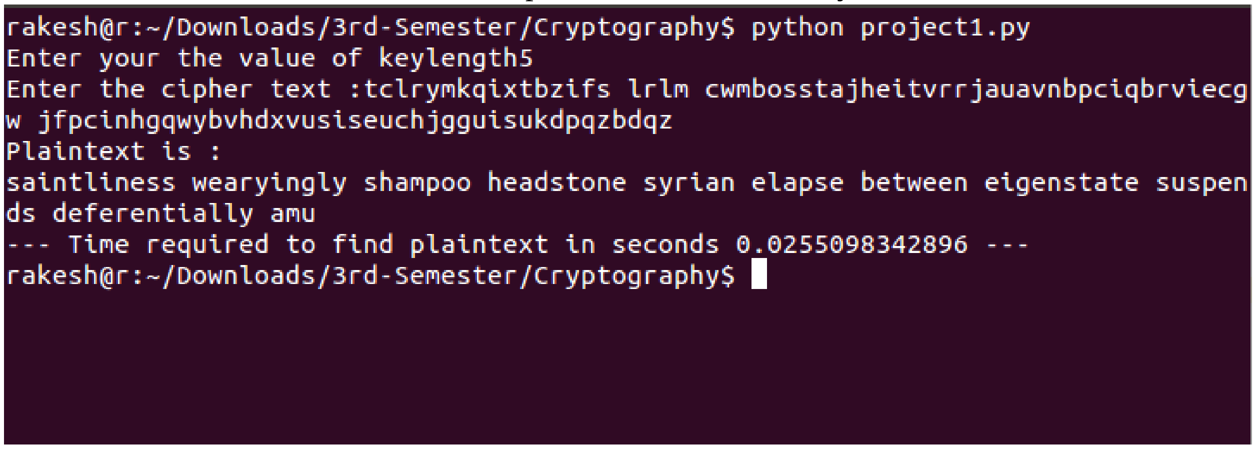
## 6. Execution Results:

For key length: 5
Execution Time: 0.025 secs, plaintext from Dictionary 1



```
rakesh@r: ~/Downloads/3rd-Semester/Cryptography
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ python project1.py
Enter your the value of keylength5
Enter the cipher text :icymshbgi fnrtjebpiyiqgwegqudrfcvywjpjdyigchfuccelbkqwyav
ksxfbuyqfucwybihdkjxhdfmnr xavkieecweerwdmq
Plaintext is :
having developed methods for measuring the data against those rules stage five a
llows the data quail
--- Time required to find plaintext in seconds 0.0223560333252 ---
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ 
```

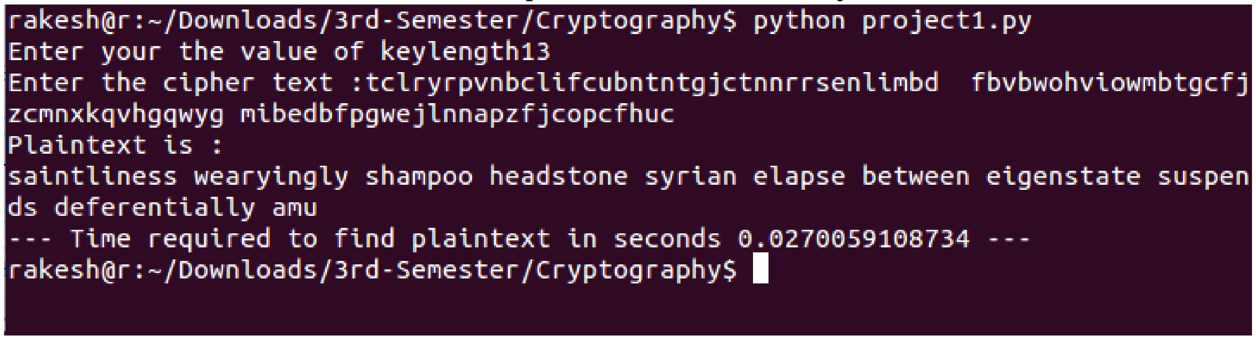
For key length: 5
Execution Time: 0.025 secs, plaintext from Dictionary2



```
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ python project1.py
Enter your the value of keylength5
Enter the cipher text :tclrymkqixtbzifs lrlm cwmbosstajheitvrrjauavnbpciqbrviecg
w jfpcinhgqwybvhdxvusiseuchjgguisukdpqzbdqz
Plaintext is :
saintliness wearyingly shampoo headstone syrian elapse between eigenstate suspen
ds deferentially amu
--- Time required to find plaintext in seconds 0.0255098342896 ---
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ 
```

For key length: 13

Execution Time: 0.027 secs, plaintext from Dictionary 2



```
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ python project1.py
Enter your the value of keylength13
Enter the cipher text :tclryrpvnbclifcubntntgjctnnrrsenlimbd  fbvbwohviowmbtgcfj
zcmnxkqvhgqwyg mibedbfpgwejlnnapzfjcopcfhuc
Plaintext is :
saintliness wearyingly shampoo headstone syrian elapse between eigenstate suspen
ds deferentially amu
--- Time required to find plaintext in seconds 0.0270059108734 ---
rakesh@r:~/Downloads/3rd-Semester/Cryptography$ 
```

## 7. Code

Code to decrypt the cipher text

```
from collections import Counter
from collections import defaultdict
import time

characterMap = {' ': 0, 'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 7, 'f': 6, 'i': 9, 'h': 8, 'k': 11, 'j': 10, 'm': 13, 'l': 12, 'o': 15, 'n': 14, 'q': 17, 'p':
16, 's': 19, 'r': 18, 'u': 21, 't': 20, 'w': 23, 'v': 22, 'y': 25, 'x': 24, 'z': 26}

sentenceMap = defaultdict()

file1 = open('Dictionary1-2.txt', 'rb')
file2 = open('Dictionary2-1.txt', 'rb')

plaintext = []

for line in file2:
    plaintext.append(line.rstrip('\r\n'))

key = input("Enter your the value of keylength")

remainder = 100 % key
quotient = 100 / key
repeatElement = key - remainder

cipherText = raw_input("Enter the cipher text :").strip().lower()
cipherMap = map(lambda x: characterMap[x], cipherText)

for i in range(len(plaintext)):
    #creating mapping for dictionary 2
    sentenceMap[plaintext[i]] = plaintext[0:i] + plaintext[i+1:]

def dictionary1():
    '''
    Find the plaintext from dictionary 1, by checking the number of unique shifts
    '''
    start_time = time.time()
    for line in file1:
        line = str(line).rstrip('\r\n')
        plainTextMap = map(lambda x: characterMap[x], line)
        frequencyMap = [a_i - b_i if a_i - b_i > 0 else a_i - b_i + 27 for a_i, b_i in zip(cipherMap, plainTextMap)]
        if len(Counter(frequencyMap).keys()) == key:
            print "Plaintext is :"
            print line
            print("--- Time required to find plaintext in seconds %s ---" % (time.time() - start_time))
            return True
    return False
```

```
def dictionary2(graph, start, path=[]):
    '''
    First generate a sentence using backtracking, and then check whether it is our possible plaintext candidate
    '''
    path = path + [start]
    if len(" ".join(path)) >= 100:
        return " ".join(path)[0:100]
    plainTextMap = map(lambda x: characterMap[x], " ".join(path))
    frequencyMap = [a_i - b_i if a_i - b_i > 0 else a_i - b_i + 27 for a_i, b_i in zip(cipherMap, plainTextMap)]
    if len(Counter(frequencyMap).keys()) <= key:
        for node in graph[start]:
            if node not in path:
                result = dictionary2(graph, node, path)
                if result:
                    plainTextMap = map(lambda x: characterMap[x], result)
                    frequencyMap = [a_i - b_i if a_i - b_i > 0 else a_i - b_i + 27 for a_i, b_i in zip(cipherMap, plainTextMap)]
                    if len(Counter(frequencyMap).keys()) == key and Counter(frequencyMap).values().count(quotient) ==
repeatElement:
                        print "Plaintext is :"
                        print result
                        print("--- Time required to find plaintext in seconds %s ---" % (time.time() - start_time))
    else:
        return

if dictionary1():
    #if plaintext is found in dictionary 1 then exit here
    exit()
else:
    for i in plaintext:
        #Create a sentence of 100 characters and test whether it will match with ciphertext
        start_time = time.time()
dictionary2(sentenceMap, i)
```

Code to find encrypted text for testing:

```
characterMap = {' ': 0, 'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 7, 'f': 6, 'i': 9, 'h': 8, 'k': 11, 'j': 10, 'm': 13, 'l': 12, 'o': 15, 'n': 14, 'q': 17, 'p':
16, 's': 19, 'r': 18, 'u': 21, 't': 20, 'w': 23, 'v': 22, 'y': 25, 'x': 24, 'z': 26}

inv_map = {v: k for k, v in characterMap.items()}

plaintext = 'saintliness wearyingly shampoo headstone syrian elapse between eigenstate suspends deferentially amu'

t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

rotatedArray = (map(lambda x: characterMap[x], plaintext))

for i in range(len(rotatedArray)):

    rotatedArray[i] = (t[i % len(t)] + rotatedArray[i]) % 27

rotatedText = map(lambda x: inv_map[x], rotatedArray)

print "".join(rotatedText)
```