CSC3621 Cryptography **Exercise 2** **Georgi Nikolov**

B4036790

The second part of the coursework I found way more challenging, compared to the first one. It took me some time to figure out how to implement certain functionalities. The approach I followed was to re-use as much as possible from the Utils I have built for the first part of the coursework. Which led to the creation of a new class VigenereUtils. I have added some methods that return stream from a file or Buffered readers to iterate through the files more quickly, initially I have tried with streams but it was slow.

It was hard to get the decryption working, because I end up in a situation where it works for the text that I encrypted myself but it would not work for the given ciphered text even though I have cracked the cipher. It turned out that, changing the alphabet size fixed it, but I still guess there is a hidden bug that I most likely have not been resolved, one wrong increment was breaking it all and again used integer values to flip around the asci values.

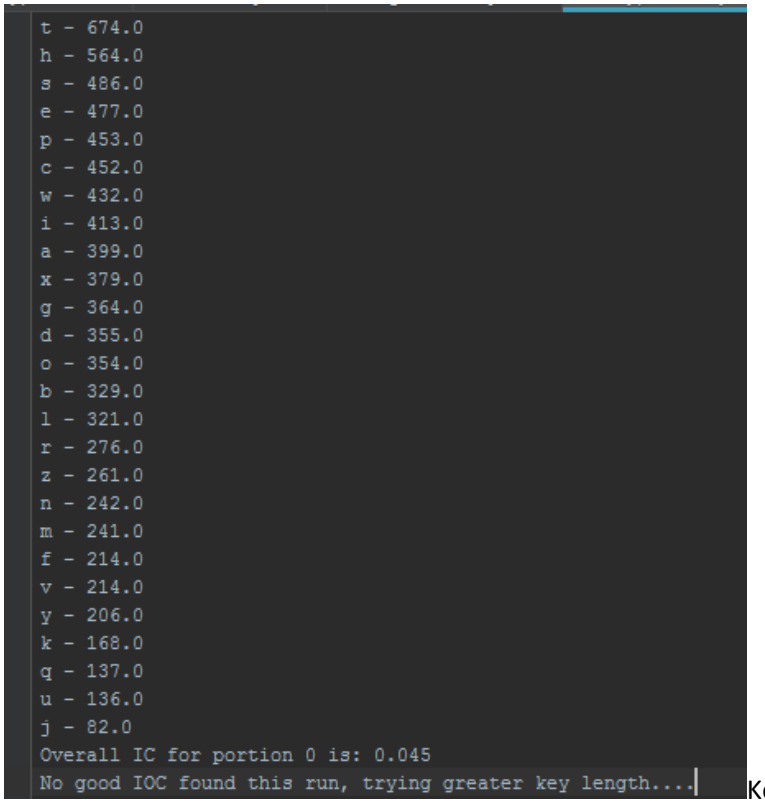
I have implemented simplistic encryption and decryption methods, which given a BufferedRead read the input, transform it on the go and write it back to a new file. The encryption method takes an extra parameter which is alphabet size, because I have done something wrong and I couldn't find the issue, but this patches it so it works. Encryption and decryption methods, called with false Boolean value will prompt the user to type in a password which is then checked by a regex, if that Boolean is not set, then the user must specify a key manually by passing a sting to the methods.

I tried to devise the method which finds the coincidence index in a way that will then try to produce the full plaintext key, which was half way success, it works for the ciphered text, but it flips some letters when I tried to recover the password for the file that I encrypted. More on this is available as comment in my code, I could not figure out why it was happening. The way I implemented the CI index method was, by creating a Map<Int,Map> data structure, which will then be initialised by a given key length, then a placeholder is added to the inner map to track count of all the characters for this partition of the text, which then is used to calculate the IC index, on per key basis, once a whole partition is scanned average IC is returned, in the meanwhile we have another data structure which takes the first most frequent letter from each partition and stores it temporary. Afterwards an overall index for all partitions is calculated, if this index in between 0.063 and 0.068 then we assume we have found the key length. With the help of those top values we saved from each partition then we try to get the plaintext key, by reading each of those top characters and based on their Integer value we decide how to subtract from the most common letter in the English alphabet 'e'. Once we get the shift value for a partition we rotate the characters accordingly and build a plaintext message containing the key which is then passed to the decrypt function implemented earlier. In case we don't find any "good" key length we increment the key length value and do the process over until a letter distribution close enough to plain English text is found.

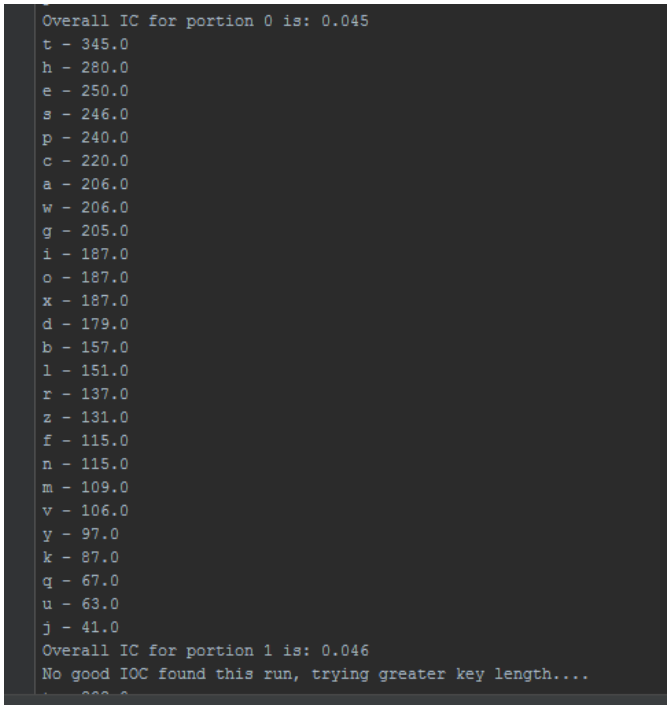
N.B This works only for the provided ciphered text I spent a lot of time trying to fix it and make it work with any text but it fails slightly for some reason, it will always flip a character.

All partition iterations through the text are represented as mod operations based on current iteration and key length. Data structures for the IC calculation are sorted on the go for easier value extraction, overall the algorithm runs fast but is quite fragile. We start with key length one to prevent zero division error, thus we need 25+1 alphabet length to decipher the text correctly. Proportions however are indexed from zero.

Here we observe how the IC values are changing while the key length is growing:



Key length =1



Key length =2

The execution time per partition is less then half a second, this includes I/O operations.

```
j - 2.0
Overall IC for portion 3 is: 0.065
s - 203.0
h - 173.0
w - 157.0
c - 137.0
o - 130.0
b - 129.0
v - 102.0
g - 99.0
f - 83.0
z - 76.0
q - 64.0
t - 58.0
r - 57.0
i - 55.0
u - 36.0
k - 33.0
a - 31.0
p - 25.0
d - 24.0
m - 22.0
j - 16.0
y - 8.0
x - 3.0
l - 2.0
e - 1.0
n - 1.0
Overall IC for portion 4 is: 0.067
IOC for key lenght 5 is 0.067
```

On the picture above, we see that only after iterating through five partitions, we find the key length. Partitions are evenly distributed, each holding about 88300+- characters. However, I could not find any patterns in the sorted by value partitions. No identically distributed values along patitions.

Plain text recovery:

```
Overall IC for portion 4 is: 0.067
IOC for key lenght 5 is 0.067
Found key at length: 5Trying to build plaintext key...

Key is: 15
Key is: 11
Key is: 0
Key is: 19
Key is: 14
plato
Execution time decrypt: 4690800
```

This was achieved with less then two seconds execution time.

Plain text recovered from cipher text:

```
neithermustweforgetthattherepublicisbutth
```

Encryption of plaintext:

```
fshcbrl wvmxkqwvw'l qww qeoxkimvut hc klusehzz lemfxp, fo tkqwmv dhgxc hezex

jibl ttsel bp xsh mab mwu hy pfceox xcqaxfkx pl dp vlhl qow txll
tejdkx oh otkxhjvmfdfw xatqhgilfk.  qsk ftv usfz bq, kywx fi embr lg
hf-npt mj ngatj jix qtjqi hy izi qkhgtux hnmbctihh efrwrif bkrdytfw
laxx mafh irphd dj eoebkt ej ppt.yyjfgubgy.dfm


ubmit: jix xsnidunkbh sv labgdssl alaeii

pmxxpk: pjxxvk zdfed whvaw

ilhlmdh wxiw: bikfa 18, 2011 [fuhlz #1661]
jyslm egwjfw: cgzunuxo 29, 2002

etkvmewf: bcypyta


*** hlehu hc llyt iodbisu zriwrrfkz ttsel met etwxgqjjii hy hzihmhvh zsbnxl ***




iklsmgue uv sr bghkneskt iodbisu zriwrrfkz kgpkomxbg ede clhw cfgxkswd









qww qeoxkimvut hc klusehzz lemfxp

cr

wys toizyh vhkpf tpreb



   y. t huedete xf rpaxjxs
  bf. xxf kbs-lubwxa dighnx
```

There were some tricky parts, where we need to manipulate value manually in order for my code to produce readable and concise results.

In terms of usage, no user interaction is needed there is a runner class under Exercise2 directory. All files again are written to Resources directory within the projet with corresponding filenames and explanation, it’s very important to first create ciphered file and then pass them to the decipher.