DAT 510: Assignment 1
Cryptanalysis of primitive ciphers
Author: Sondre Tennø

Abstract

In this assignment the goal was to get familiar with primitive ciphers in cryptanalysis. Task 1 was about cracking a poly-alphabetic cipher, while in task 2, we would get familiar with the simplified version of the DES algorithm. In Task 1 I managed to decipher the cipher, calculate execution times for different key lengths and I did not manage to decipher the last cipher with the same tools as I did the first. In task 2 I deciphered both tables with SDES and TripleSDES, but I did not manage to do task 3 and 4 on part 2.

Part 1

Task 1

The cipher text to decrypt:
'BQZRMQ KLBOXE WCCEFL DKRYYL BVEHIZ NYJQEE BDYFJO PTLOEM EHOMIC UYHHTS GKNJFG EHIMK NIHCTI HVRIHA RSMGQT RQCSXX CSWTNK PTMNSW AMXVCY WEOGSR FFUEEB DKQLQZ WRKUCO FTPLOT GOJZRI XEPZSE ISXTCT WZRMXI RIHALE SPRFAE FVYORI HNITRG PUHITM CFCDLA HIBKLH RCDIMT WQWTOR DJCNDY YWMJCN HDUWOF DPUPNG BANULZ NGYPQU LEUXOV FFDCEE YHQUXO YOXQUO DDCVIR RPJCAT RAQVFS AWMJCN HTSOXQ UODDAG BANURR REZJGD VJSXOO MSDNIT RGPUHN HRSSSF VFSINH MSGPCM ZJCSLY GEWGQT DREASV FPXEAR IMLPZW EHQGMG WSEIXE GQKPRM XIBFWL IPCHYM OTNXYV FFDCEE YHASBA TEXCJZ VTSGBA NUDYAP IUGTLD WLKVRI HWACZG PTRYCE VNQCUP AOSPEU KPCSNG RIHLRI KUMGFC YTDQES DAHCKP BDUJPX KPYMBD IWDQEF WSEVKT CDDWLI NEPZSE OPYIW'

The plaintext I gathered from the decryption:
anoriginalmessageisknownastheplaintextwhilethecodedmessageiscalledtheciphertex ttheprocessofconvertingfromplaintexttociphertextisknownasencipheringorencryptionr estoringtheplaintextfromtheciphertextisdecipheringordecryptionthemanyschemesuse dforencryptionconstitutetheareaofstudyknownascryptographysuchaschemeisknowna sacryptographicsystemoraciphertechniquesusedfordecipheringamessagewithoutany knowledgeoftheencipheringdetailsfallintotheareaofcryptanalysiscryptanalysisiswhatth elaypersoncallsbreakingthecodetheareasofcryptographyandcryptanalysistogetherare calledcryptology

The plaintext I gathered from the decryption sorted:

‘An original message is known as the plaintext while the coded message is called the ciphertext. The process of converting from plaintext to ciphertext is known as enciphering or encryption. Restoring the plaintext from the ciphertext is deciphering or decryption. The many schemes used for encryption constitute the area of study known as cryptography. Such a scheme is known as a cryptographic system or a cipher. Techniques used for deciphering a message without any knowledge of the enciphering details fall into the area of cryptanalysis. Cryptanalysis is what the layperson calls breaking the code. The areas of cryptography and cryptanalysis together are called cryptology’

Started out by getting a letter of frequency of the english language, so I could later use this to crack the key. Also added in the constants, such as cipher, maximum key length, etc.

Got the frequency from https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

```
### Assignment 1 in DAT510. Cryptanalysis of primitive ciphers ###
### Part 1. Poly-alphabetic Ciphers ###

### Task 1: Decipher cipher with key length no longer than 10

#Array of the letter frequency of the letters in the English language. Data gathered from https://www3.nd.edu/~busiforc/handouts/
letter_frequency = [0.084966,0.020720,0.045388,0.033844,0.111607,0.018121,0.024705,0.030034,0.075448,0.001965,0.011016,0.054893,0

#Cipher to decrypt
cipher_to_decrypt = "BQZRMQ KLBOXE WCCEFL DKRYYL BVEHIZ NYJQEE BDYFJO PTLOEM EHOMIC UYHHTS GKNJFG EHIMK NIHCTI HVRIHA RSMGQT RQCS

#Max key length
key_length_max = 10

#English alphabet
english_alphabet = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']
```

Then I created the functions to find the key. Here I would calculate the letter frequency table for each letter of the key by using chi squared.

```python
# Get key using freq_analysis function.
def get_cipher_key(cipher_to_decrypt, key_length):
    key = ''
    key_length_range = range(key_length)

    # Calculate letter frequency table for each letter of the key
    for i in key_length_range:
        sequence=""

        # breaks the ciphertext into sequences
        for j in range(len(0,cipher_to_decrypt[i:]), key_length):
            sequence+=cipher_to_decrypt[i+j]
        key+=freq_analysis(sequence)
    return key
```

```python
def freq_analysis(sequence):
    all_chi_squareds = [0] * 26

    for i in range(26):

        chi_squared_sum = 0.0

        sequence_offset = [chr(((ord(sequence[j])-97-i)%26)+97) for j in range(len(sequence))]
        v = [0] * 26
        for l in sequence_offset:
            v[ord(l) - ord('a')] += 1
        for j in range(26):
            v[j] *= (1.0/float(len(sequence)))

        for j in range(26):
            chi_squared_sum+=((v[j] - float(letter_frequency[j]))**2)/float(letter_frequency[j])

        all_chi_squareds[i] = chi_squared_sum
    shift = all_chi_squareds.index(min(all_chi_squareds))

    return chr(shift+97)
```

Then I turned the ciphertext and key into ascii and used the key to decrypt the ciphertext.

```python
def decrypt(ciphertext, key):
    cipher_ascii = [ord(letter) for letter in ciphertext]
    key_ascii = [ord(letter) for letter in key]
    plain_ascii = []

    for i in range(len(cipher_ascii)):
        plain_ascii.append(((cipher_ascii[i]-key_ascii[i % len(key)]) % 26) +97)

    plaintext = ''.join(chr(i) for i in plain_ascii)
    return plaintext
```

I didn’t use any statistical techniques to find the key length, instead I tested the key length for 1-10(max), and saw which one gave a sensible output when I decrypted.

```python
def find_key_length(ciphertextfiltered,key_length):
    key_length = key_length+1

    for x in range(key_length):
        if x != 0:
            key = get_key(ciphertextfiltered, x)
            plaintext = decrypt(ciphertextfiltered, key)

            print("Key: {}".format(key))
            print("Plaintext: {}".format(plaintext))
```

Which gave me the outputs:

```
Key: a
Plaintext: bqzrmqklboxewccefldkryylbvehiznyjqeebdyfjoptloemehomicuyhhtsgknjfgehimknihctihvriharsmgqtrqcsxxcswtnkptmnswamxvcyw
eogsrffueebdkqlqzwrkucoftplotgojzrixepzseisxtctwzrmxirihalesprfaefvyorihnitrgpuhitmcfcdlahibklhrcdimtwqwtordjcndyywmjcnhduwof
dpupngbanulzngypquleuxovffdceeyhquxoyoxquoddcvirrpjcatraqvfsawmjcnhtsoxquoddagbanurrrezjgdvjsxoomsdnitrgpuhnhrsssfvfsinhmsgpc
mzjcslygewgqtdreasvfpxearimlpzwehqgmgwseixegqkprmxibfwlipchymotnxyvffdceeyhasbatexcjzvtsgbanudyapiugtldwlkvrihwaczgptrycevnqc
upaospeukpcsngrihlrikumgfcytdqesdahckpbdujpxkpymbdiwdqefwsevktcddwlinepzseopyiw
Key: aq
Plaintext: bazbmakvbyxowmcofvduriyvbferijnijaeobnypjypdlyewerowimuihrtcguntfqeriwkxircdirvbirabswgatbqmshxmsgtxkztwncwkmhvmyg
eygcrpfeeobnkalazgruumoptzlytqotzbihezzcesshtmtgzbmhibiravecpbfkepviobirnstbgzuridmmfmdvarilkvhbcniwtgqgtyrnjmnnyiwwjmnrdewyf
npepxglaxuvzxgipauveexyvpfncoeihauhoiohqeondmvsrbptcktbaavpskwwjmnrtcohqeondkglaxubrbejjqdfjcxyowsnnstbgzurnrrcscfffcixhwsqpm
mjjmsvyqeggatnroacvpphekrsmvpjwohagwggsoiheqqupbmhilfglspmhimytxxivpfncoeihksladehctzftcglaxunykpsuqtvdgluvbirwkcjgztbymefnac
epkocpouupmsxgbirlbiuuwgpcitnqosnarcupldejzxupimldswnqofgsovutmdnwvixezzceypiig
Key: aaa
```

```
Plaintext: bqzrmqklboxewccefldkryylbvehiznyjqeebdyfjoptloemehomicuyhhtsgknjfgehimknihctihvriharsmgqtrqcsxxcswtnkptmnswamxvcyw
eogsrffueebdkqlqzwrkucoftplotgojzrixepzseisxtctwzrmxirihalesprfaefvyorihnitrgpuhitmcfcdlahibklhrcdimtwqwtordjcndyywmjcnhduwof
dpupngbanulzngypquleuxovffdceeyhquxoyoxquoddcvirrpjcatraqvfsawmjcnhtsoxquoddagbanurrrezjgdvjsxoomsdnitrgpuhnhrsssfvfsinhmsgpc
mzjcslygewgqtdreasvfpxearimlpzwehqgmgwseixegqkprmxibfwlipchymotnxyvffdceeyhasbatexcjzvtsgbanudyapiugtldwlkvrihwaczgptrycevnqc
upaospeukpcsngrihlrikumgfcytdqesdahckpbdujpxkpymbdiwdqefwsevktcddwlinepzseopyiw
```


```
Key: adcy
Plaintext: bnxtmninblvgwzagfibmrvwnbscjiwlajncgbawhjlnvllcoeemoizsaherughllfdcjijipieaviettieytsjestooesuvestrpkmronpucmuteyt
cqgpphfrcgbaislnxyrhseocrrllriogxtiucrzpcksurettxtmugtieynepntfxchvvmtielktoeruegvmzdediyjiyinhoafijryqtrqrahenawawjhenebwwld
fprnpgyypuixpgvnsuicwxlthfaagevfsuumaouowoabevfptpgactoysvcqcwjheneruouowoabcgyypuoptewhidshuxlmosalktoerueljrpqufsduikfosdne
mwhesiwietestapgapthpuccrfknpwughneogtqgiuciqhntmugdftjkpzfamlrpxvthfaagevfcsyyveualzsrugyypuawcpfsitibylhttieuccwertoweeslsc
rncopnguhnesketiejtihsogcaataogsayjchnddrhrxhnamybkwaogftqgvhredaunikcrzpcqpvgy
Key: eawaa
Plaintext: xqdrmmkpboteaccafpdknyclbrelizjynqeabhyffottlkeqehkmmcuuhltsckrjfcelimgnmhcpilvrehersigutrmcwxxysatngpxmnowemxrccw
ekgwrfbuiebzkulqvwvkuyojtphoxgofzvixapdseesbtcpwdrmtivihwlispnfeefrysridnmtrcpyhipmgfczlehixkphrydmmtsqatondncnzycwmfcrhdqwsf
dlutngxarulvnkypmupeutozffzcieydqyxouobqukdhcvervpjyaxramvjsasmncndtwoxmusddwgfanqrvrevjkdvfsbooishniprkpudnlrsosjvfoirhmogtc
mvjgslugiwgmthrewszfpteeriiltzwahugmcwweitekqklrqxixfalilclymktrxyrfjdcaechaobetetcnzvpskbajuhyaliygthdalkrrmhwwcdgpprccernuc
ulasspauopconkridlvikqmkfcuthqeodehcgpfdufpbkpumfdisduefssivkpchdwhirepvsiopuia
Key: mqbzta
Plaintext: paystqyvapeekmbfmlruqzflpfdipzbiirlepnxgqoddkplmsrnnpciigiasuumkmgsrhnrnwrbuphjbhihrgwfraremryecggsorphwmtdaahudfw
syftyftedfidyakrgwfutdvfhzkpagctyspxszytlighsdawnblyprwrzmlsdbeblfjinsphbsssnpirhutctmcmhhwljmorqnhnawegspydxmmefykwiduhrevpm
ddeoonboxtmgnuiorblsewpcftnbflyvatyvychpvvdrmujyrdtbbaroaugzakwiduhhcnyxucncbnboxtsyrsjihkvxcwpvmgnmjaruztiuhfcrtmvtchoomgqod
tzxmrmfgsgfradfoztcfdhdbyiavoadevafnnwgohylgeuostxwlexsidmgztohxwzcftnbflyvkrchtshbkgvhcfchninxbwiiqsmkwzuusphkkbanphbxdlvbab
vwaccofbkdmronrwrkspkiwfgjyhnpfzdorblwbreiqekdilckiknpfmwgoulacrnvmpnszytlodihx
```

```
Key: kmakbea
Plaintext: rezhlmkbpondscssfbcgromlruahynnoimeupdoefofhlediexcmybqyxvtifgnztgugemabixbpixjrygwriaggsnqsgxnbowjbkfsinikacwrcok
eeforvtuudxdaelgysraiceeppbctwnfzhwxuovsuwsnsytmnrcweryvabdophtaueryefixmethupkgetcqfschaxwbakdrsricssqmhohcfcdryovijsbhttsov
rpkojgronkkvnwmpgtheklolebdsseogmuncyewmuerdsuerhdjszprqevvrwwcxcdgpselqknzdqubqmqrhfepicdlxsnnkmirnysngfihdgnsigfleoidvmiflc
cnjsrhywswwppdhsaiubpnsahhilfnwugmgcuwidexuuqaonmnwbvvhifqholktdlylebdsseogwsrotuwyjpjtifxadidozlikutbcslajrygsasngfsnyssvdpy
ufooioauadcimcryvlhhgucufsxpdgsstzdcadbttfpnypolxdykdgdbwisvasydtklymappgeeouim
Key: bdlaekcy
Plaintext: anoriginalmessageisknownastheplaintextwhilethecodedmessageiscalledtheciphertexttheprocessofconvertingfromplaintext
tociphertextisknownasencipheringorencryptionrestoringtheplaintextfromtheciphertextisdecipheringordecryptionthemanyschemesused
forencryptionconstitutetheareaofstudyknownascryptographysuchaschemeisknownasacryptographicsystemoraciphertechniquesusedfordec
ipheringamessagewithoutanyknowledgeoftheencipheringdetailsfallintotheareaofcryptanalysiscryptanalysisiswhatthelaypersoncallsb
reakingthecodetheareasofcryptographyandcryptanalysistogetherarecalledcryptology
Key: awoqauwap
Plaintext: bulbmwolmobqgciifwdodiyrfvphmlxypuepbhkpjuttwoiyohuqinuctrtykkyjjsohoqkyilodinzrthedcmmutcqgehxiwwenobdmtwwlmbhmyc
iorsvrpukibokuxazcvkfcsrdprstronlbidipksiucxzgthzvyhixmhlliezrleeqvcabinrierkbehoxmnfgpvanmbvlldmdoqthqafyrjncydckgmpgnsdyiyf
jtuanknknapzygcbauriuiozrpdiiejhughoesxbuspncbmrcpnoktxeqgfwmgmpgnstwahqasdoaknknavrcedvqdbnsiosycdtmtcgtgrnnvsdsjhpsorhxskbm
mfncdlcsowmutorimcvltxpavuwlvdwphuswgcwetxisakvvmiifrglotcsyqadndcvqfhooeeladbefoxinzgtwslatydjatuegzpdhlohbinaanzkbdregegnuo
epgssaeywzcyrgcilxbiqymrfgkddwisoalouphhuupbwzysfdtwhcofcwegkxondcpiyetlceutytw
```

```
Key: panzaqawpa
Plaintext: mqmsmakpmoiejdcofpokcylmbfeltzyywreobhjfuoculyeqphzmvduihlesrkakfqeltmvnvicdilgrthnsswguerbcfyxmsaenvpgnncwexxgclx
eygwcfqurfbnkuwqkwelumojepwoghotzvtxppmtessbecewmsmhivthllrtpbfepfgybsirnmerrphiidmgqcolniilkpsrndvntgqaeocdwdnnychmucaidewsq
daucoglarflkntzpauppuioigfncipysqhyoiobbuzdqdvsrvajnagsaavjdahmwdnrtwzxbubedkgflnfresejjkovuskpowshyiertqurnlcsdsswfcirsmdgcd
mjjgdljgrxgathcelsigpheecixlcawohurmrwffihekbkarzyilfawiacuzmytriygfsecoecsadbnuehcnkvestcaxuhjaaihhtvdawkgrviwkcdrperldefnun
uaabtpouoacdntsirlvtkfmtgcithbeddnicupfouupklpimfoihddffgsigkecqewvirppksrppiia
```

Where the only sensible output was key length = 8. With key = ‘bdlaekcy’
Which gave me the answer:

```
# We can see that key length 8 is the only key length giving
# us a sensible plaintext. Key length 8 must be correct.

key_length = 8
key = get_key(filtered_text, key_length)
plaintext = decrypt(filtered_text, key)

print('Key: ' + key)
print('Plaintext: ' + plaintext)
```

```
Key: bdlaekcy
Plaintext: anoriginalmessageisknownastheplaintextwhilethecodedmessageiscalledtheciphertexttheprocessofconvertingfromplaintextto
ciphertextisknownasencipheringorencryptionrestoringtheplaintextfromtheciphertextisdecipheringordecryptionthemanyschemesusedfore
ncryptionconstitutetheareaofstudyknownascryptographysuchaschemeisknownasacryptographicsystemoraciphertechniquesusedfordecipheri
ngamessagewithoutanyknowledgeoftheencipheringdetailsfallintotheareaofcryptanalysiscryptanalysisiswhatthelaypersoncallsbreakingt
hecodetheareasofcryptographyandcryptanalysistogetherarecalledcryptology
```

Task 2: Try shorter and longer key lengths in my program to find out program execution time

Here I used the ‘magic commands’ in jupyter notebook to time the execution time. I went through the decryption with key length 1-10 and timed them 100 times and gathered the mean.

```
%%timeit
key_length = 1
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
7.72 ms ± 472 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 2
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
8.95 ms ± 548 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 3
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
8.9 ms ± 430 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 4
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
9.93 ms ± 415 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 5
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
10.5 ms ± 192 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 6
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
11.6 ms ± 569 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 7
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
11.9 ms ± 462 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 8
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
12 ms ± 730 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 9
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
13.1 ms ± 489 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
key_length = 10
key = get_key(ciphertextfiltered, key_length)
plaintext = decrypt(ciphertextfiltered, key)
```

```
13.4 ms ± 745 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Mean times for 100 runs:
Key length = 1 = 7,72ms
Key length = 2 = 8,95ms
Key length = 3 = 8,9ms
Key length = 4 = 9,93ms
Key length = 5 = 10,5ms
Key length = 6 = 11,6ms
Key length = 7 = 11,9ms
Key length = 8 = 12ms
Key length = 9 = 13,1ms
Key length = 10 = 13,4ms

Here we can clearly see that increasing the key length, will increase the decryption time.

Task 3: Reproduce the decryption on another cipher with the same key that has an addition in the encryption process.

New cipher to decrypt: 'BQZRMQ KLAYAV AYITET EOFGWT EALRRD HNIFML BIHHQY XXEXYV LPHFLW UOJILE GSDLKH BZGCTA LHKAIZ BIOIGK SZXLZS UTCPZW JHNPUS MSDITN OSKSJI EOKVIL BKMSZB XZOEHA KTAWXP WLUEJM AIWGLR TZLVHZ SATVQI HZWAXX ZXDCIV TMLBIQ RWZMLB VNGVQK AIZBXZ HVVMMA MJLRIW GKITZL VHZRRV YCBTVM FVOIYE FSKGKJ

AVWHUV BUHZSA EFLHMQ HHVSGZ XIKYTS YZXUUC KBTOGU VABLDP BGJCGF NLIIYA HJFWGG PSCPVA ZEASME MLGOYR CGFXVG EJTTTW TSAAIL QFKEEP CPULXW WZRLVI VVYUMS MSILRI IBLWJI TKWUXZ GUZEJG DUCQEE QEOBTP SIHTGW UALVMA ILTAEZ TFLDPE IVEGYH PLZRTC YJVYGX ABFNPQ XLCEYA RGIFCC WHBGIF WSYLBZ MDWFPX KZSYCY APJTFR CKTYYU YICYLR ZALETS DWHMGR PTTGUW CGFNTB JTRNWR AADNPQ XLTBGP RZMJTF KGTSPV DTVAPE ZPRIP'

Using the same tools as I did with the previous cipher and the same key:

```
cipher_to_decrypt2 = 'BQZRMQ KLAYAV AYITET EOFGWT EALRRD HNIFML BIHHQY XXEXYV LPHFLW UOJILE GSDLKH BZGCTA LHKAIZ BIOIGK SZXLZS UT
filtered_text2 = filtertext(cipher_to_decrypt2)

plaintext = decrypt(filtered_text2,'bdlaekcy')
```

Results plaintext:
'uhilcalgerhuldsubknnjogbzncnukdnkucbdsxiekkcxriumogyninpegqzbglqharfudyowctbjlrcvkjnbrcw bqcvyyacdkbtnvromlcwfjopcywjkclilurnddgodjscjoggozunrueosscrpywdtktnqgplmnuouteptzfabblp scebpdkwmmzcjrrstognphvybmcncvtejkaqpwdxmmwalacptzivnjldomvgqdefeirqjqcemqunlrrnkmc ssllrfkopesqamwbewubptmfbkflxnyimqdwbeyvkjrzrlxinnscumjcsphmqvktvjyyqmmjvbnhpuxswwplj ezrsqdzfynpajzkyimjnposofvcgbqgukklavhncmvsnjdopmjjuyvlmybnyfdulncgqjgrnemrnlfqqfjcgisdn bkrvbgbemktqkhrpzliyzqlzntzyudiahscjjcadmxfnkkdrfrccbdsaskczbnhamzeywizpiciljmlucmhandsx yeykjhdyxpjnhqabvtcqxlyeykfmovtvmdmqvzppdqbssilwiaczhdugmphojiayrcanpkdamjcxmreckkynj qmsmtpzddtpckahmblwtuwjgdphmspjjbrickonaqloimcxjfslxkijhsh'

I was not able to decrypt the second cipher with the same key as I used in the previous cipher.

Part 2
Task 1 and 2:

Results of the test cases is Task 1 and 2:

Raw key	**Plaintext**	**Ciphertext**
0000000000	00000000	11110000
0000011111	11111111	11100001
0010011111	11111100	10011101
0010011111	10100101	10010000
1111111111	11111111	00001111
0000011111	00000000	01000011

1000101110	00111000	00011100
1000101110	00001100	11000010

```
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key: 0000000000
Input binary value of Plaintext: 00000000
encrypted result:  0b11110000
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key: 0000011111
Input binary value of Plaintext: 11111111
encrypted result:  0b11100001
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key: 0010011111
Input binary value of Plaintext: 11111100
encrypted result:  0b10011101
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key: 0010011111
Input binary value of Plaintext: 10100101
encrypted result:  0b10010000

Enter e/E for Encrypt, d/D for Decrypt and 0 to exit:
```

Raw Key 1	Raw key 2	Plaintext	Ciphertext
1000101110	0110101110	11010111	10111001
1000101110	0110101110	10101010	11100100
1111111111	1111111111	00000000	11101011
0000000000	0000000000	01010010	10000000
1000101110	0110101110	11111101	11100110
1011101111	0110101110	01001111	01010000
1111111111	1111111111	10101010	00000100
0000000000	0000000000	00000000	11110000

```
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key k1: 1000101110
Input binary value of Raw Key k2: 0110101110
Input binary value of Plaintext: 10101010
encrypted result:  0b11100100
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key k1: 1111111111
Input binary value of Raw Key k2: 1111111111
Input binary value of Plaintext: 00000000
encrypted result:  0b11101011
```

I started the project by implementing all the constants used for SDES. P10, P8, IP, IP^-1,S0,S1,E/P,P4 tables.

```
### Assignment 1 in DAT510. Cryptanalysis of primitive ciphers ###
### Part 2. Simplified DES

#Import all constants

key_length = 10
subkey_length = 8

P10 = (3,5,2,7,4,10,1,9,8,6)
P8 = (6,3,7,4,8,5,10,9)
IP = (2,6,3,1,4,8,5,7)
Ipinverse = (4,1,3,5,7,2,8,6)
EP = (4,1,2,3,2,3,4,1)
S0 = (1,0,3,2,3,2,1,0,0,2,1,3,3,1,3,2)
S1 = (0,1,2,3,2,0,1,3,3,0,1,0,2,1,0,3)
P4 = (2,4,3,1)
```

Then I implemented the functions to permute input byte according to permutation tables

```
def Ipinverse_func(inputByte):
    return perm(inputByte, Ipinverse)
```

```
def ip(inputByte):
    return perm(inputByte, IP)
```

```
def perm(inputByte, permTable):
    outputByte = 0
    for index, elem in enumerate(permTable):
        if index >= elem:
            outputByte |= (inputByte & (128 >> (elem - 1))) >> (index - (elem - 1))
        else:
            outputByte |= (inputByte & (128 >> (elem - 1))) << ((elem - 1) - index)
    return outputByte
```

Then I implemented the function to generate key.

```
def key_gen(key):
    keyList = [(key & 1 << i) >> i for i in reversed(range(key_length))]
    permKeyList = [None] * key_length
    for index, elem in enumerate(P10):
        permKeyList[index] = keyList[elem - 1]
    shiftedOnceKey = leftShift(permKeyList)
    shiftedTwiceKey = leftShift(leftShift(shiftedOnceKey))
    subKey1 = subKey2 = 0
    for index, elem in enumerate(P8):
        subKey1 += (128 >> index) * shiftedOnceKey[elem - 1]
        subKey2 += (128 >> index) * shiftedTwiceKey[elem - 1]
    return (subKey1, subKey2)
```

Then I implemented functions to encrypt and decrypt

```
def decrypt(key, ciphertext):
    data = fk(key_gen(key)[1], ip(ciphertext))
    return Ipinverse_func(fk(key_gen(key)[0], swapNibbles(data)))
```

```
def encrypt(key, plaintext):
    data = fk(key_gen(key)[0], ip(plaintext))
    return Ipinverse_func(fk(key_gen(key)[1], swapNibbles(data)))
```

In the main function of SDES I added the functionality to run and input data, and give out the correct decryption or encryption based on the input.

```
if __name__ == '__main__':
    choice = ''
    while choice != "0":
        choice = input('Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: ')
        if choice.lower()=='e':
            k = int(input("Input binary value of Raw Key: "), 2)
            pt = int(input("Input binary value of Plaintext: "), 2)
            print("encrypted result: ",bin(encrypt(k, pt)))
        elif choice.lower()=='d':
        #elif choice.lower().startswith('d'):
            k1 = int(input("Input binary value of Raw Key: "), 2)
            ct = int(input("Input binary value of Ciphertext: "), 2)
            print("decrypted result: ", bin(decrypt(k1, ct)))
        elif choice != '0':
            print ('You have entered an invalid input, please try again. \n\n')
    exit()
```

```
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key: 1111111111
Input binary value of Plaintext: 00000000
encrypted result:  0b11101011
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: 0
```

The TripleSDES is very similar, the only difference is in the main class, where I run the algorithm for TripleSDES and it’s a class called TripleSDES.

```
# Instance of Class Main
Object = TripleSDES()

choice = ''
while choice != "0":
    choice = input('Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: ')
    if choice.lower()=='e':
        k1 = int(input("Input binary value of Raw Key k1: "), 2)
        k2 = int(input("Input binary value of Raw Key k2: "), 2)
        pt = int(input("Input binary value of Plaintext: "), 2)
        print("encrypted result: ", bin(Object.encrypt(k1,Object.decrypt(k2, Object.encrypt(k1,pt)))))
    elif choice.lower()=='d':
    #elif choice.lower().startswith('d'):
        k1 = int(input("Input binary value of Raw Key k1: "), 2)
        k2 = int(input("Input binary value of Raw Key k2: "), 2)
        ct = int(input("Input binary value of Ciphertext: "), 2)
        print("decrypted result: ", bin(Object.decrypt(k1,Object.encrypt(k2, Object.decrypt(k1,ct)))))
    elif choice != '0':
        print ('You have entered an invalid input, please try again. \n\n')
exit()
```

```
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: e
Input binary value of Raw Key k1: 0000000000
Input binary value of Raw Key k2: 0000000000
Input binary value of Plaintext: 01010010
encrypted result:  0b10000000
Enter e/E for Encrypt, d/D for Decrypt and 0 to exit: d
Input binary value of Raw Key k1: 1000101110
Input binary value of Raw Key k2: 0110101110
Input binary value of Ciphertext: 11100110
decrypted result:  0b11111101
```

I did not manage to do task 3 and 4 of Part 2.

Conclusion

I learned a lot about ciphers and how to decrypt them.I managed to do all the tasks in Part 1, and task 1 and 2 in Part 2. My implementation to Part 1 was very basic and I could maybe have used a statistical approach to finding the key length, instead of trial and error.
I did not manage to learn how to set-up a server to send responses to, in task 4.

Works Cited

Stallings, William. Appendig G Simplified DES. 2010

Stallings, William. Cryptography and Network Security: Principles and Practice (7th Edition). 2016

Joao H de a Franco. Simplified DES implementation in Python.
https://jhafranco.com/2012/02/10/simplified-des-implementation-in-python/. 2012

Letter frequencies in the english language.
https://www3.nd.edu/~busiforc/handouts/cryptography/letterfrequencies.html

Simplified DES Introduction
https://www.cs.uri.edu/cryptography/dessimplified.htm

Steflik, Dick. Simplified DES
https://www.cs.binghamton.edu/~steflik/cs431/notes/Simplified_DES.pdf