# Software Optimization Homework

P1SO01

M.Sc. Software Engineering
University of Europe for Applied Sciences

Berkin Öztürk
Prof. Dr. Rand Kouatly
20 May 2024

# INTRODUCTION

Tasks are designed to provide hands-on experience with encryption and decryption processes, highlighting the strengths and vulnerabilities of each method.

The first task requires the implementation of the Monoalphabetic Substitution Cipher system, where a message is encrypted and decrypted using a single, consistent substitution alphabet. The key for this substitution is randomly generated and securely shared between the sender, Alice, and the receiver, Bob. Additionally, a third party, Oscar, attempts to decrypt the message without access to the key, prompting an exploration of effective hacking techniques.

The second task extends this exploration to the Polyalphabetic Substitution Cipher, which utilizes multiple substitution alphabets based on a repeating key. This method increases security by mitigating the frequency analysis attacks that are effective against monoalphabetic ciphers. The task involves assessing the time and complexity required for Oscar to break this more complex encryption.

The third task introduces the DES (Data Encryption Standard) cipher, a widely used symmetric-key algorithm for securing digital data. The assignment requires implementing the full DES encryption and decryption process for both text and bitmap images.

The assignment was done using Python 3.0 and not using any libraries for encryption. The comprehensive report documents the solution, screenshots, results, and all related code, ensuring a thorough understanding and clear presentation of the work performed.

# PART 1 - Monoalphabetic Substitution Cipher

Monoalphabetic substitution cipher is a simple encryption method that uses a single substitution alphabet to encrypt a text. In this encryption method, each letter is replaced by another letter according to a certain rule. For example, in alphabetical order, the letter A is replaced with Z, the letter B with Y, and the letter C with X. This substitution alphabet used as the key is the same in encryption and decryption processes. In the encryption process, each letter in the plaintext is changed according to the key alphabet. In the decryption process, each letter in the ciphertext is changed according to the reverse of the key. This method is weak against frequency analysis as it does not change letter frequencies and can therefore be easily broken.

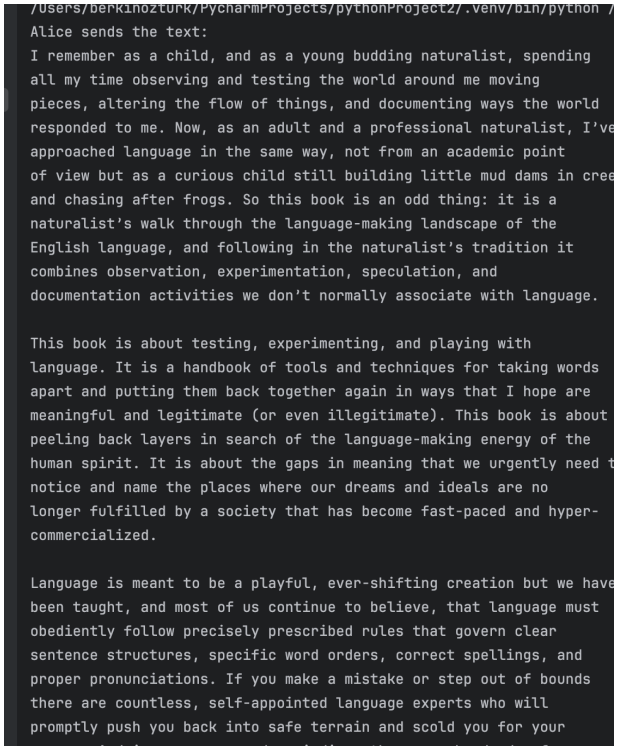
As seen below, we define a random key so that our key does not change constantly. In this way, we can easily apply the frequency analysis method, crack the password and read the secret text.

```
8
9   # Define constant key
10  CONSTANT_KEY = {'a': 'z', 'b': 'y', 'c': 'x', 'd': 'w', 'e': 'v',
11                  'f': 'u', 'g': 't', 'h': 's', 'i': 'r', 'j': 'q',
12                  'k': 'p', 'l': 'o', 'm': 'n', 'n': 'm', 'o': 'l',
13                  'p': 'k', 'q': 'j', 'r': 'i', 's': 'h', 't': 'g',
14                  'u': 'f', 'v': 'e', 'w': 'd', 'x': 'c', 'y': 'b', 'z': 'a'}
15
```

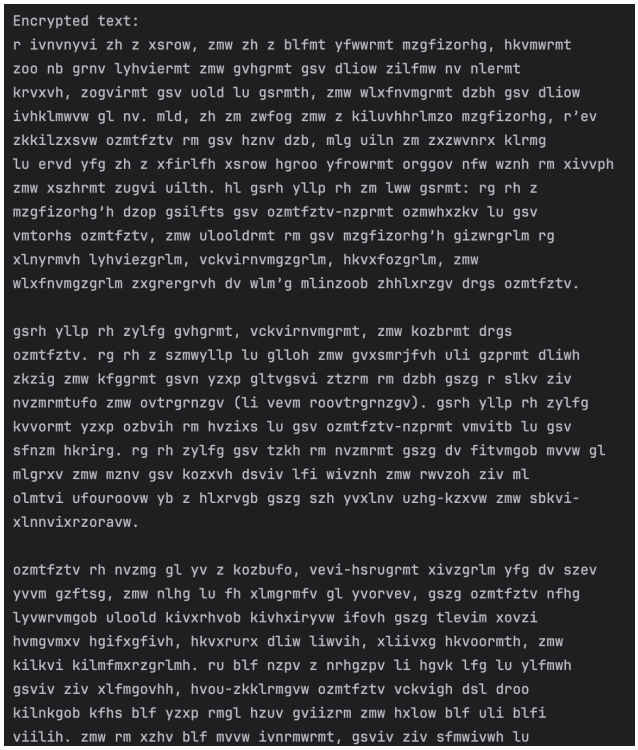
Encryption and Decryption operations are performed with the functions listed below. The encrypt function takes a message and a key (a dictionary that maps each letter to another letter). It loops through each character in the message, adds the corresponding encrypted letter from the key dictionary if it is a letter, otherwise adds it as is, and returns the resulting encrypted message. The decrypt function takes the encrypted message and the same key, creates a new dictionary by reversing the key (matching the modified letters to the original letters). It loops checks each character in the encrypted message, if it is a letter it adds the corresponding original letter from the reverse key dictionary, if not it adds it as is and returns the resulting decrypted message.

```python
# Encryption
3 usages
def encrypt(message, key):
    encrypted_message = ''
    for char in message.lower():
        if char.isalpha():
            encrypted_message += key[char]
        else:
            encrypted_message += char
    return encrypted_message


# Decryption
2 usages
def decrypt(encrypted_message, key):
    decrypted_message = ''
    reversed_key = {v: k for k, v in key.items()}
    for char in encrypted_message.lower():
        if char.isalpha():
            decrypted_message += reversed_key[char]
        else:
            decrypted_message += char
    return decrypted_message
```

We can see the original text, encrypted text and decrypted text below.



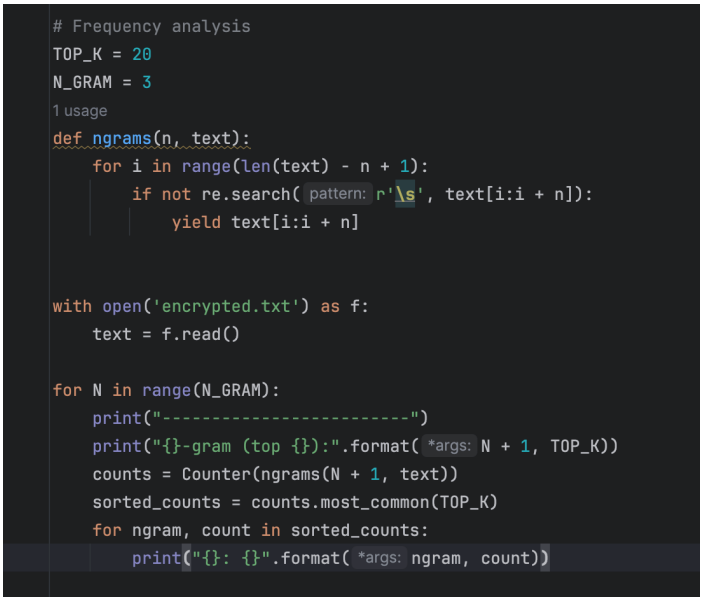
```
/Users/berkinoztürk/PycharmProjects/pythonProject27.venv/bin/python /
Alice sends the text:
I remember as a child, and as a young budding naturalist, spending
all my time observing and testing the world around me moving
pieces, altering the flow of things, and documenting ways the world
responded to me. Now, as an adult and a professional naturalist, I've
approached language in the same way, not from an academic point
of view but as a curious child still building little mud dams in cree
and chasing after frogs. So this book is an odd thing: it is a
naturalist's walk through the language-making landscape of the
English language, and following in the naturalist's tradition it
combines observation, experimentation, speculation, and
documentation activities we don't normally associate with language.

This book is about testing, experimenting, and playing with
language. It is a handbook of tools and techniques for taking words
apart and putting them back together again in ways that I hope are
meaningful and legitimate (or even illegitimate). This book is about
peeling back layers in search of the language-making energy of the
human spirit. It is about the gaps in meaning that we urgently need t
notice and name the places where our dreams and ideals are no
longer fulfilled by a society that has become fast-paced and hyper-
commercialized.

Language is meant to be a playful, ever-shifting creation but we have
been taught, and most of us continue to believe, that language must
obediently follow precisely prescribed rules that govern clear
sentence structures, specific word orders, correct spellings, and
proper pronunciations. If you make a mistake or step out of bounds
there are countless, self-appointed language experts who will
promptly push you back into safe terrain and scold you for your
```



```
Encrypted text:
r ivnvnyvi zh z xsrow, zmw zh z blfmt yfwwrmt mzgfizorhg, hkvmwrmt
zoo nb grnv lyhviermt zmw gvhgrmt gsv dliow zilfmw nv nlermt
krvxvh, zogvirmt gsv uold lu gsrmth, zmw wlxfnvmgrmt dzbh gsv dliow
ivhklmwvw gl nv. mld, zh zm zwfog zmw z kiluvhhrlmzo mzgfizorhg, r'ev
zkkilzxsvw ozmtfztv rm gsv hznv dzb, mlg uiln zm zxzwvnrx klrmg
lu ervd yfg zh z xfirlfh xsrow hgroo yfrowrmt orggov nfw wznh rm xivvph
zmw xszhrmt zugvi uilth. hl gsrh yllp rh zm lww gsrmt: rg rh z
mzgfizorhg'h dzop gsilfts gsv ozmtfztv-nzprmt ozmwhxzkv lu gsv
vmtorhs ozmtfztv, zmw ulooldrmt rm gsv mzgfizorhg'h gizwrgrlm rg
xlnyrmvh lyhviezgrlm, vckvirnvmgzgrlm, hkvxfozgrlm, zmw
wlxfnvmgzgrlm zxgrergrvh dv wlm'g mlinzoob zhhlxrzgv drgs ozmtfztv.

gsrh yllp rh zylfg gvhgrmt, vckvirnvmgrmt, zmw kozbrmt drgs
ozmtfztv. rg rh z szmwyllp lu glloh zmw gvxsmrjfvh uli gzprmt dliwh
zkzig zmw kfggrmt gsvn yzxp gltvgsvi ztzrm rm dzbh gszg r slkv ziv
nvzmrmtufo zmw ovtrgrnzgv (li vevm roovtrgrnzgv). gsrh yllp rh zylfg
kvvormt yzxp ozbvih rm hvzixs lu gsv ozmtfztv-nzprmt vmvitb lu gsv
sfnzm hkrirg. rg rh zylfg gsv tzkh rm nvzmrmt gszg dv fitvmgob mvvw gl
mlgrxv zmw mznv gsv kozxvh dsviv lfi wivznh zmw rwvzoh ziv ml
olmtvi ufouroovw yb z hlxrvgb gszg szh yvxlnv uzhg-kzxvw zmw sbkvi-
xlnnvixrzoravw.

ozmtfztv rh nvzmg gl yv z kozbufo, vevi-hsrugrmt xivzgrlm yfg dv szev
yvvm gzftsg, zmw nlhg lu fh xlmgrmfv gl yvorvev, gszg ozmtfztv nfhg
lyvwrvmgob uloold kivxrhvob kivhxiryvw ifovh gszg tlevim xovzi
hvmgvmxv hgifxgfivh, hkvxrurx dliw liwvih, xliivxg hkvoormth, zmw
kilkvi kilmfmxrzgrlmh. ru blf nzpv z nrhgzpv li hgvk lfg lu ylfmwh
gsviv ziv xlfmgovhh, hvou-zkklrmgvw ozmtfztv vckvigh dsl droo
kilnkgob kfhs blf yzxp rmgl hzuv gviizrm zmw hxlow blf uli blfi
viilih. zmw rm xzhv blf mvvw ivnrmwrmt, gsviv ziv sfmwivwh lu
```

```
Decrypted text:
i remember as a child, and as a young budding naturalist, spending
all my time observing and testing the world around me moving
pieces, altering the flow of things, and documenting ways the world
responded to me. now, as an adult and a professional naturalist, i've
approached language in the same way, not from an academic point
of view but as a curious child still building little mud dams in creeks
and chasing after frogs. so this book is an odd thing: it is a
naturalist's walk through the language-making landscape of the
english language, and following in the naturalist's tradition it
combines observation, experimentation, speculation, and
documentation activities we don't normally associate with language.

this book is about testing, experimenting, and playing with
language. it is a handbook of tools and techniques for taking words
apart and putting them back together again in ways that i hope are
meaningful and legitimate (or even illegitimate). this book is about
peeling back layers in search of the language-making energy of the
human spirit. it is about the gaps in meaning that we urgently need to
notice and name the places where our dreams and ideals are no
longer fulfilled by a society that has become fast-paced and hyper-
commercialized.

language is meant to be a playful, ever-shifting creation but we have
been taught, and most of us continue to believe, that language must
obediently follow precisely prescribed rules that govern clear
sentence structures, specific word orders, correct spellings, and
proper pronunciations. if you make a mistake or step out of bounds
there are countless, self-appointed language experts who will
promptly push you back into safe terrain and scold you for your
errors. and in case you need reminding, there are hundreds of
```

To break the cipher, we must apply frequency analysis and compare the most repeated letters with the most repeated letters in English.

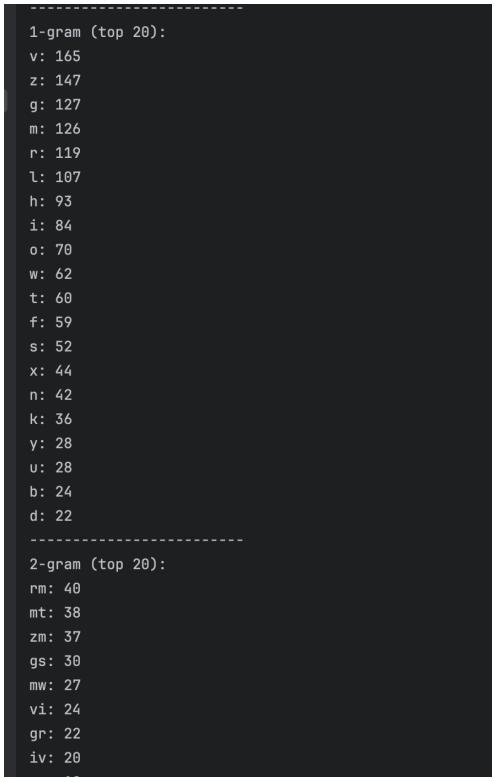


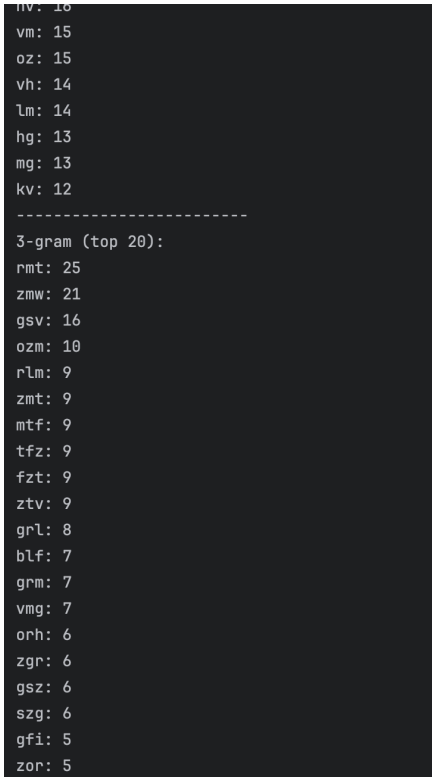
```python
# Frequency analysis
TOP_K = 20
N_GRAM = 3
1 usage
def ngrams(n, text):
    for i in range(len(text) - n + 1):
        if not re.search( pattern: r'\s', text[i:i + n]):
            yield text[i:i + n]


with open('encrypted.txt') as f:
    text = f.read()

for N in range(N_GRAM):
    print("-------------------------")
    print("{}-gram (top {}):".format( *args: N + 1, TOP_K))
    counts = Counter(ngrams(N + 1, text))
    sorted_counts = counts.most_common(TOP_K)
    for ngram, count in sorted_counts:
        print("{}: {}".format( *args: ngram, count))
```

Constants: TOP_K defines how many top n-grams to display, and N_GRAM sets the maximum length of n-grams to analyze.

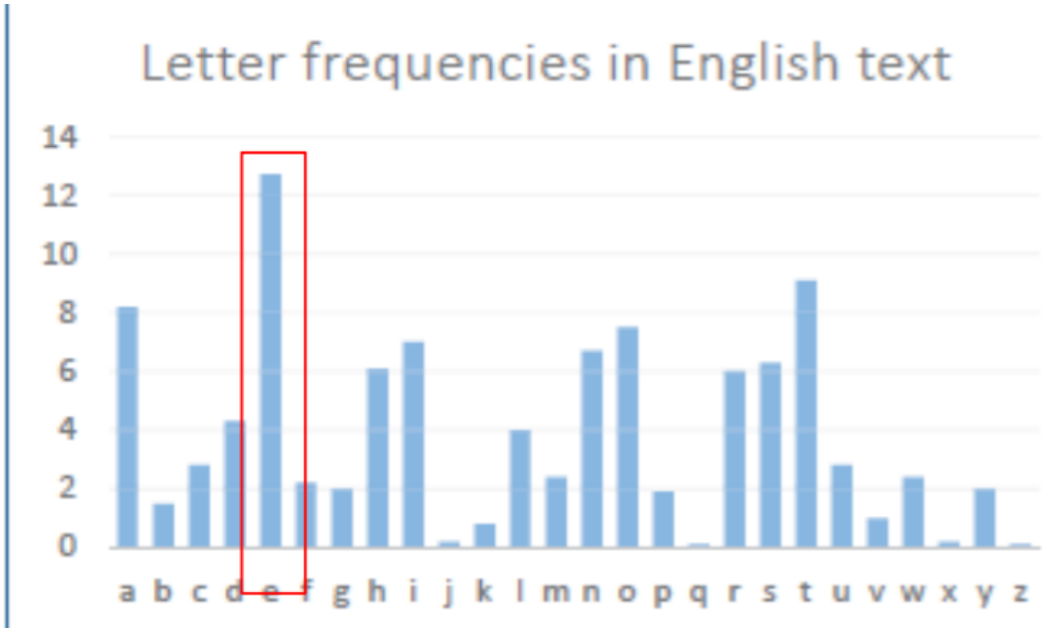
Ngrams Function: Generates n-grams of a specified length from the text, excluding those with whitespace.

Frequency Analysis: Loops through n-gram lengths from 1 to N_GRAM, generates n-grams, counts their occurrences, and prints the top TOP_K most common n-grams for each length.




With the frequency analysis we applied to ciphertext above, we can see the most recurring 1-gram, 2-gram and 3-grams. We will try to solve ciphertext without a key by comparing it with the 1-gram, 2-gram and 3-grams that are most commonly used in English.




| Trigram frequency in English | | | | | |
|---|---|---|---|---|---|
| THE : | 1.81 | ERE : | 0.31 | HES : | 0.24 |
| AND : | 0.73 | TIO : | 0.31 | VER : | 0.24 |
| ING : | 0.72 | TER : | 0.30 | HIS : | 0.24 |
| ENT : | 0.42 | EST : | 0.28 | OFT : | 0.22 |
| ION : | 0.42 | ERS : | 0.28 | ITH : | 0.21 |
| HER : | 0.36 | ATI : | 0.26 | FTH : | 0.21 |
| FOR : | 0.34 | HAT : | 0.26 | STH : | 0.21 |
| THA : | 0.33 | ATE : | 0.25 | OTH : | 0.21 |
| NTH : | 0.33 | ALL : | 0.25 | RES : | 0.21 |
| INT : | 0.32 | ETH : | 0.24 | ONT : | 0.20 |

| Bigram frequency in English | | | | | |
|---|---|---|---|---|---|
| TH : | 2.71 | EN : | 1.13 | NG : | 0.89 |
| HE : | 2.33 | AT : | 1.12 | AL : | 0.88 |
| IN : | 2.03 | ED : | 1.08 | IT : | 0.88 |
| ER : | 1.78 | ND : | 1.07 | AS : | 0.87 |
| AN : | 1.61 | TO : | 1.07 | IS : | 0.86 |
| RE : | 1.41 | OR : | 1.06 | HA : | 0.83 |
| ES : | 1.32 | EA : | 1.00 | ET : | 0.76 |
| ON : | 1.32 | TI : | 0.99 | SE : | 0.73 |
| ST : | 1.25 | AR : | 0.98 | OU : | 0.72 |
| NT : | 1.17 | TE : | 0.98 | OF : | 0.71 |

We compare them with our own results based on the tables above.

Our most recurring letter is v, and the single most used letter in English is e. Thus we can say v = E.

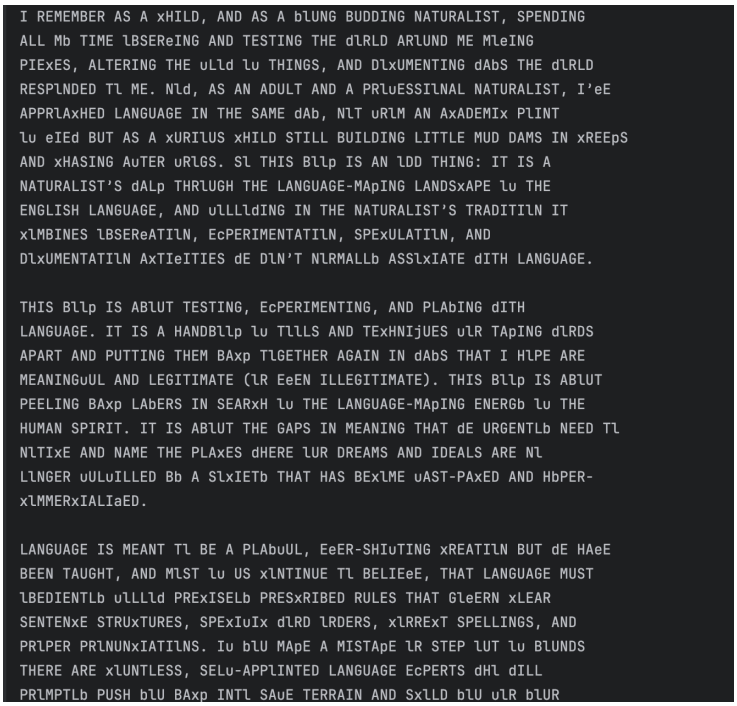
When we look at our bigram table, vi = ER, rm = IN, rmt = ING , zmw = AND

```
I REnEnyER Ah A xsIoD, AND Ah A blfNG yfDDING NAgfRAoIhg, hkENDING
Aoo nb gInE lyhEReING AND gEhgING gsE dlRoD ARlfND nE nleING
kIExEh, AogERING gsE uold lu gsINGh, AND DlxfnENgING dAbh gsE dlRoD
REhklNDED gl nE. Nld, Ah AN ADfog AND A kRluEhhIlNAo NAgfRAoIhg, I'eE
AkkRlAxsED oANGfAGE IN gsE hAnE dAb, Nlg uRln AN AxADEnIx klINg
lu eIEd yfg Ah A xfRIlfh xsIoD hgIoo yfIoDING oIggoE nfD DAnh IN xREEp
AND xsAhING AugER uRlGh. hl gsIh yllp Ih AN lDD gsING: Ig Ih A
NAgfRAoIhg'h dAop gsRlfGs gsE oANGfAGE-nApING oANDhxAkE lu gsE
ENGoIhs oANGfAGE, AND ulooldING IN gsE NAgfRAoIhg'h gRADIgIlN Ig
xlnyINEh lyhEReAgIlN, EckERInENgAgIlN, hkExfoAgIlN, AND
DlxfnENgAgIlN AxgIeIgIEh dE DlN'g NlRnAoob AhhlxIAgE dIgs oANGfAGE.
```

When we look at the second word, it becomes very clear that n = M, gs = TH, h = S.

```
I REMEMyER AS A xHIoD, AND AS A blfNG yfDDING NATfRAoIST, SkENDING
Aoo Mb TIME lySEReING AND TESTING THE dlRoD ARlfND ME MleING
kIExES, AoTERING THE uold lu THINGS, AND DlxfMENTING dAbS THE dlRoD
RESklNDED Tl ME. Nld, AS AN ADfoT AND A kRluESSIlNAo NATfRAoIST, I'eE
AkkRlAxHED oANGfAGE IN THE SAME dAb, NlT uRlM AN AxADEMIx klINT
lu eIEd yfT AS A xfRIlfS xHIoD STIoo yfIoDING oITToE MfD DAMS IN xREEpS
AND xHASING AuTER uRlGS. Sl THIS yllp IS AN lDD THING: IT IS A
NATfRAoIST'S dAop THRlfGH THE oANGfAGE-MApING oANDSxAkE lu THE
ENGoISH oANGfAGE, AND ulooldING IN THE NATfRAoIST'S TRADITIlN IT
xlMyINES lySEReATIlN, EckERIMENTATIlN, SkExfoATIlN, AND
DlxfMENTATIlN AxTIeITIES dE DlN'T NlRMAoob ASSlxIATE dITH oANGfAGE.
```

We continue solving and among the missing words we find y = B, k = P, o = L, f = U etc. We can remove letters like this.




Now it's easy to guess the remaining words: x = C, bl = YO, dl = WO and so on.

```python
# Trying to hack with replacing the words
frequency_analysis_key = "virmtzwngshykofxblducepjaq"
change_key = "ERINGADMTHSBPLUCYOWFXVKQZJ"
hack = tr(frequency_analysis_key, change_key, encrypted_text)
print(hack)
```

This is the final version, we can read the encrypted text by replacing all the letters we find with the ones in the ciphertext. We break the key.

```
I REMEMBER AS A CHILD, AND AS A YOUNG BUDDING NATURALIST, SPENDING
ALL MY TIME OBSERVING AND TESTING THE WORLD AROUND ME MOVING
PIECES, ALTERING THE FLOW OF THINGS, AND DOCUMENTING WAYS THE WORLD
RESPONDED TO ME. NOW, AS AN ADULT AND A PROFESSIONAL NATURALIST, I'VE
APPROACHED LANGUAGE IN THE SAME WAY, NOT FROM AN ACADEMIC POINT
OF VIEW BUT AS A CURIOUS CHILD STILL BUILDING LITTLE MUD DAMS IN CREEKS
AND CHASING AFTER FROGS. SO THIS BOOK IS AN ODD THING: IT IS A
NATURALIST'S WALK THROUGH THE LANGUAGE-MAKING LANDSCAPE OF THE
ENGLISH LANGUAGE, AND FOLLOWING IN THE NATURALIST'S TRADITION IT
COMBINES OBSERVATION, EXPERIMENTATION, SPECULATION, AND
DOCUMENTATION ACTIVITIES WE DON'T NORMALLY ASSOCIATE WITH LANGUAGE.
```

Let's try to break monoalphabetic ciphering in a different language, for example Turkish, with a new key.

```
# Alice will send a message in another language, such as "Select
# your mother language" or "German language," and repeat steps
# 2 and 3.


# Define a new constant key
CONSTANT_KEY_2 = {
    'a': 'q', 'b': 'w', 'c': 'e', 'd': 'r', 'e': 't',
    'f': 'y', 'g': 'u', 'h': 'i', 'i': 'o', 'j': 'p',
    'k': 'a', 'l': 's', 'm': 'd', 'n': 'f', 'o': 'g',
    'p': 'h', 'q': 'j', 'r': 'k', 's': 'l', 't': 'z',
    'u': 'x', 'v': 'c', 'w': 'v', 'x': 'b', 'y': 'n', 'z': 'm'
}



new_alice_text = open("turkish_example.txt")
txt2 = new_alice_text.read()
new_alice_text.close()
new_encrypted_text = encrypt(txt2, CONSTANT_KEY_2)
new_decrypted_text = decrypt(new_encrypted_text, CONSTANT_KEY_2)

frequency_analysis_key2 = "ognewftxkrdlcusgamzih"
change_key2 = "IAYCBNEURDMSVGLOKZTHP"
new_hack = tr(frequency_analysis_key2, change_key2, new_encrypted_text)
print(new_encrypted_text)
print(new_decrypted_text)
print(new_hack)
```

```
# Frequency analysis
TOP_K = 20
N_GRAM = 3
1 usage
def ngrams(n, text):
    for i in range(len(text) - n + 1):
        if not re.search( pattern: r'\s', text[i:i + n]):
            yield text[i:i + n]


with open('turkish_encrypted.txt') as f:
    text = f.read()

for N in range(N_GRAM):
    print("-------------------------")
    print("{}-gram (top {}):".format( *args: N + 1, TOP_K))
    counts = Counter(ngrams(N + 1, text))
    sorted_counts = counts.most_common(TOP_K)
    for ngram, count in sorted_counts:
        print("{}: {}".format( *args: ngram, count))
```

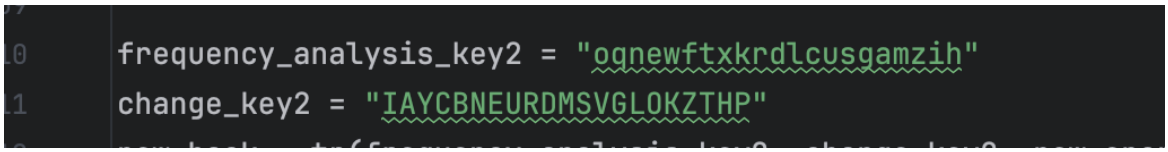
We define a new random constant key table for a new key and encrypt our Turkish text. We extract the results of frequency analysis and compare them with the most used letters in Turkish (Only 1-gram results are available on the internet). In this way, we can successfully crack our password again by trial and error.

Turkish Language Letter Frequency:

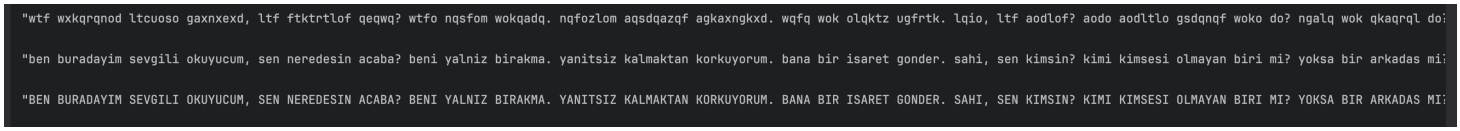
a e i n r l ı d k m u y t s b o ü ş z g ç h ğ v c ö p f j w x q

```
-------------------------
1-gram (top 20):
o: 55
q: 52
f: 44
t: 40
l: 32
x: 30
k: 28
d: 26
a: 25
w: 24
n: 21
s: 21
g: 18
.: 13
r: 12
z: 8
,: 7
c: 6
m: 6
e: 5
-------------------------
```

We placed the most repeated letters in our cipher text with the most repeated letters in Turkish by trial and error.




We can see the encrypted text, the decrypted text and the text we cracked.




As a result, it is quite easy to break monoalphabetic substitution ciphering.

# PART 2 - Polyalphabetic Substitution Cipher.

A polyalphabetic substitution cipher is a method of encrypting text that uses multiple substitution alphabets to encode the message. Unlike a monoalphabetic cipher, which uses a single substitution alphabet for the entire plaintext, a polyalphabetic cipher changes the substitution alphabet regularly, making it more secure against frequency analysis attacks. A key or keyword is determined and each letter of this key is repeated up to the length of the text to be encrypted. Each key letter defines the substitution alphabet by shifting a certain amount. Decryption is accomplished by swiping in the opposite direction using the same key. This method is more secure than single-alphabet passwords because it hides letter frequencies and makes attacks based on frequency analysis difficult.


This function creates translation tables for encryption and decryption based on a given key.

key: The key used for generating the translation tables.

N: The length of the key.

s: The string containing all lowercase ASCII letters.

trantab_enc: A list to store the translation tables for encryption.

trantab_dec: A list to store the translation tables for decryption.

The loop iterates over each character k in the key.

mapping: A list of all lowercase ASCII letters.

random.Random(k).shuffle(mapping): Uses the key character k to seed the random shuffling of mapping.

trantab_enc[i]: Creates a translation table for encryption by mapping each letter in s to its shuffled counterpart.

trantab_dec[i]: Creates a translation table for decryption by reversing the mapping.

Encryption function iterates over each character in the text. If the character is a lowercase letter, it is translated using the appropriate translation table (determined by i % N). Non-lowercase characters are added to ciphertext without change. And decryption function converting ciphertext to original text using the key.

Key; A string used to generate the translation tables for encryption and decryption. In this case, the key is a single character, "b", but it can be any string.

In the below, we create our translation tables and read our file from "example.txt" and transfer it to txt. We encrypt plaintext and then decrypt it. We repeat the same process in our own language.

polyalphabetic.py

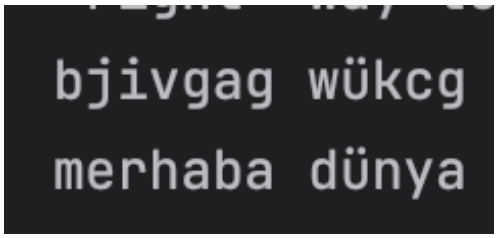
```
40
41  # User-provided key
42  key = "b"
43
44  # Create translation tables
45  trantab_enc, trantab_dec = create_translation_tables(key)
46
47  # Read the text from file
48  with open("example.txt", "r") as myfile:
49      txt = myfile.read()
50  print("Original text:", txt)
51
52  # Encrypt the text
53  enctext = encrypt(txt, trantab_enc, key)
54  print("Encrypted text:", enctext)
55
56  # Decrypt the text
57  dectext = decrypt(enctext, trantab_dec, key)
58  print("Decrypted text:", dectext)
59
60  # Another language
61  text = "merhaba dünya"
62  enctext2 = encrypt(text, trantab_enc, key)
63  print(enctext2)
64
65  dectext2 = decrypt(enctext2, trantab_dec, key)
66  print(dectext2)
```

```
Encrypted text: I ijbjbaji gd g pvynw, gkw gd g cqskm aswwykm kgusignydu, dejkw
gnn bc uybj qadjifykm gkw ujduykm uvj zqinw giqskw bj bqfykm
eyjpjd, gnujiykm uvj lnqz ql uvykmd, gkw wqpsbjkuykm zgcd uvj zqinw
ijdeqkwjw uq bj. Nqz, gd gk gwsnu gkw g eiqljddyqkgn kgusignydu, I'fj
geeiqgpvjw ngkmsgmj yk uvj dgbj zgc, kqu liqb gk gpgwjbyp eqyku
ql fyjz asu gd g psiyqsd pvynw duynn asynwykm nyuunj bsw wgbd yk pijjhd
gkw pvgdykm gluji liqmd. Sq uvyd aqqh yd gk qww uvykm: yu yd g
kgusignydu'd zgnh uviqsmv uvj ngkmsgmj-bghykm ngkwdpgej ql uvj
Ekmnydv ngkmsgmj, gkw lqnnqzykm yk uvj kgusignydu'd uigwyuyqk yu
pqbaykjd qadjifguyqk, jrejiybjkuguyqk, dejpsnguyqk, gkw
wqpsbjkuguyqk gpuyfyuyjd zj wqk'u kqibgnnc gddqpyguj zyuv ngkmsgmj.

Tvyd aqqh yd gaqsu ujduykm, jrejiybjkuykm, gkw engcykm zyuv
ngkmsgmj. Iu yd g vgkwaqqh ql uqqnd gkw ujpvkytsjd lqi ughykm zqiwd
gegiu gkw esuuykm uvjb agph uqmjuvji gmgyk yk zgcd uvgu I vqej gij
bjgkykmlsn gkw njmyuybguj (qi jfjk ynnjmyuybguj). Tvyd aqqh yd gaqsu
ejjnykm agph ngcjid yk djgipv ql uvj ngkmsgmj-bghykm jkjimc ql uvj
vsbgk deyiyu. Iu yd gaqsu uvj mged yk bjgkykm uvgu zj simjkunc kjjw uq
kquypj gkw kgbj uvj engpjd zvjij qsi wijgbd gkw ywjgnd gij kq
nqkmji lsnlynnjw ac g dqpyjuc uvgu vgd ajpqbj lgdu-egpjw gkw vceji-
pqbbjipygnyxjw.
```

```
Decrypted text: I remember as a child, and as a young budding naturalist, spending
all my time observing and testing the world around me moving
pieces, altering the flow of things, and documenting ways the world
responded to me. Now, as an adult and a professional naturalist, I've
approached language in the same way, not from an academic point
of view but as a curious child still building little mud dams in creeks
and chasing after frogs. So this book is an odd thing: it is a
naturalist's walk through the language-making landscape of the
English language, and following in the naturalist's tradition it
combines observation, experimentation, speculation, and
documentation activities we don't normally associate with language.

This book is about testing, experimenting, and playing with
language. It is a handbook of tools and techniques for taking words
apart and putting them back together again in ways that I hope are
meaningful and legitimate (or even illegitimate). This book is about
peeling back layers in search of the language-making energy of the
human spirit. It is about the gaps in meaning that we urgently need to
notice and name the places where our dreams and ideals are no
longer fulfilled by a society that has become fast-paced and hyper-
commercialized
```

We have successfully seen encrypted and decrypted texts.

We can see a different text sent by Alice, both encrypted and decrypted. Its meaning is "hello world" in Turkish.




Since the place of all letters will change depending on the key we use in polyalphabetic, different letters are used for the same letters. Therefore, we can try to crack the password with the brute-force method. The provided code attempts to brute force the key used to encrypt a message. It tries every possible lowercase letter as the key, decrypts the encrypted text with each key, and prints the result. Thus we can select the meaningful text among them.

```
# Brute force to find the key and decrypt
for key in string.ascii_lowercase:
    trantab_enc, trantab_dec = create_translation_tables(key)
    decrypted_text = decrypt(enctext, trantab_dec, key)
    print(f"Trying key '{key}': {decrypted_text}")
```

Since our key is "b", the text tried with the key "a" does not appear meaningful.

```
Trying key 'a': I zleleplz hr h jgkmv, hbv hr h xyfbq pfvvkbq bhufzhmkru, rnlbvk
hmm ex ukel yprlzskbq hbv ulrukbq ugl cyzmv hzyfbv el eyskbq
nkljlr, hmulzkbq ugl imyc yi ugkbqr, hbv vyjfelbukbq chxr ugl cyzmv
zlrnybvlv uy el. Nyc, hr hb hvfmu hbv h nzyilrrkybhm bhufzhmkru, I'sl
hnnzyhjglv mhbqfhql kb ugl rhel chx, byu izye hb hjhvlekj nykbu
yi sklc pfu hr h jfzkyfr jgkmv rukmm pfkmvkbq mkuuml efv vher kb jzlldr
hbv jghrkbq hiulz izyqr. Sy ugkr pyyd kr hb yvv ugkbq: ku kr h
bhufzhmkru'r chmd ugzyfqg ugl mhbqfhql-ehdkbq mhbvrjhnl yi ugl
Ebqmkrg mhbqfhql, hbv iymmyckbq kb ugl bhufzhmkru'r uzhvkukyb ku
jyepkblr yprlzshukyb, ltnlzkelbuhukyb, rnljfmhukyb, hbv
vyjfelbuhukyb hjukskuklr cl vyb'u byzehmmx hrryjkhul ckug mhbqfhql.

Tgkr pyyd kr hpyfu ulrukbq, ltnlzkelbukbq, hbv nmhxkbq ckug
mhbqfhql. Iu kr h ghbvpyyd yi uyymr hbv uljgbkwflr iyz uhdkbq cyzvr
hnhzu hbv nfuukbq ugle phjd uyqluglz hqhkb kb chxr ughu I gynl hzl
elhbkbqifm hbv mlqkukehul (yz lslb kmmlqkukehul). Tgkr pyyd kr hpyfu
nllmkbq phjd mhxlzr kb rlhzjg yi ugl mhbqfhql-ehdkbq lblzqx yi ugl
gfehb rnkzku. Iu kr hpyfu ugl qhnr kb elhbkbq ughu cl fzqlbumx bllv uy
byukjl hbv bhel ugl nmhjlr cglzl yfz vzlher hbv kvlhmr hzl by
mybqlz ifmikmmlv px h ryjklux ughu ghr pljyel ihru-nhjlv hbv gxnlz-
jyeelzjkhmkalv.
```

We have reached a meaningful text where our key is "b". Thus, we understood that the key is “b”.

```
Trying key 'b': I remember as a child, and as a young budding naturalist, spending
all my time observing and testing the world around me moving
pieces, altering the flow of things, and documenting ways the world
responded to me. Now, as an adult and a professional naturalist, I've
approached language in the same way, not from an academic point
of view but as a curious child still building little mud dams in creeks
and chasing after frogs. So this book is an odd thing: it is a
naturalist's walk through the language-making landscape of the
English language, and following in the naturalist's tradition it
combines observation, experimentation, speculation, and
documentation activities we don't normally associate with language.

This book is about testing, experimenting, and playing with
language. It is a handbook of tools and techniques for taking words
apart and putting them back together again in ways that I hope are
meaningful and legitimate (or even illegitimate). This book is about
peeling back layers in search of the language-making energy of the
human spirit. It is about the gaps in meaning that we urgently need to
notice and name the places where our dreams and ideals are no
longer fulfilled by a society that has become fast-paced and hyper-
commercialized
```

```
Trying key 'c': I apcpcwpa yx y ztbms, yes yx y rgiek wissbek eyoiaymbxo, xupesbek
ymm cr obcp gwxpafbek yes opxobek otp hgams yagies cp cgfbek
ubpzpx, ymopabek otp qmgh gq otbekx, yes sgzicpeobek hyrx otp hgams
apxugesps og cp. Ngh, yx ye ysimo yes y uagqpxxbgeym eyoiaymbxo, I'fp
yuuagyztps myekiykp be otp xycp hyr, ego qagc ye yzyspcbz ugbeo
gq fbph wio yx y ziabgix ztbms xobmm wibmsbek mboomp cis sycx be zappdx
yes ztyxbek yqopa qagkx. Sg otbx wggd bx ye gss otbek: bo bx y
eyoiaymbxo'x hymd otagikt otp myekiykp-cydbek myesxzyup gq otp
Eekmbxt myekiykp, yes qgmmghbek be otp eyoiaymbxo'x oaysbobge bo
zgcwbepx gwxpafyobge, pvupabcpeoyobge, xupzimyobge, yes
sgzicpeoyobge yzobfbobpx hp sge'o egacymmr yxxgzbyop hbot myekiykp.

Ttbx wggd bx ywgio opxobek, pvupabcpeobek, yes umyrbek hbot
myekiykp. Io bx y tyeswggd gq oggmx yes opztebnipx qga oydbek hgasx
yuyao yes uioobek otpc wyzd ogkpotpa ykybe be hyrx otyo I tgup yap
cpyebekqim yes mpkbobcyop (ga pfpe bmmpkbobcyop). Ttbx wggd bx ywgio
uppmbek wyzd myrpax be xpyazt gq otp myekiykp-cydbek pepakr gq otp
ticye xubabo. Io bx ywgio otp kyux be cpyebek otyo hp iakpeomr epps og
egobzp yes eycp otp umyzpx htpap gia sapycx yes bspymx yap eg
mgekpa qimqbmmps wr y xgzbpor otyo tyx wpzgcp qyxo-uyzps yes trupa-
zgccpazbymbjps.
```

A meaningless text appeared again with the "c" key. These keys continue to be tried up to "z" and we are presented with 26 different texts. We already found what we needed in the previous one.

If the key was longer, the time to be solved would increase proportionately.

For a single character key, there are 26 possible keys (since there are 26 letters in the alphabet) which is computationally trivial and very fast.

If the key length increases, the number of possible keys increases exponentially.

For a key of length N, each position in the key can be one of 26 letters.

If decrypting with one key takes a certain amount of time "t", then decrypting with all possible keys for a key length N will take 26^N x t

So, for the answer; yes, Oscar can break the ciphering and the time depends on the length of the key.

After the find out what key length is, we can split the message into *n* parts (columns), and try to break each of it as an individual monoalphabetic cipher, starting with frequency analysis.

# PART 3 - DES Ciphering

DES uses the same key to encrypt and decrypt data and operates on 64-bit blocks of data. DES uses a 64-bit long key, but since every eighth bit is used for parity checking, the effective key length is 56 bits. The key is divided into 16 subkeys (48-bit each) and used in encryption operations. The 64-bit data block is initialized permuted according to a fixed table. This process is the rearrangement of bits according to a certain order. DES performs 16 rounds of encryption, each with the same structure. Each round transforms the block of data using a different subkey. At the end of 16 rounds, the left and right halves are combined to obtain a 64-bit data block. This block of data is subjected to final permutation according to an immutable table. In DES encryption, the ciphertext is obtained by repeating the above processes. Decryption is accomplished using the same key and applying the subkeys in reverse order.

As you can see below, we define our constants and helper functions.

Initial Permutation (IP) Table: Used to permute the bits of the plaintext at the beginning of the encryption process.

Final Permutation (FP) Table: Used to permute the bits of the ciphertext at the end of the encryption process.

Expansion (E) Table: Expands the 32-bit half-block to 48 bits to prepare for the XOR operation with the 48-bit round key.

S-Boxes (S_BOXES): 8 substitution boxes used in the substitution step to transform 6-bit input into 4-bit output.

Permutation (P) Table: Used to permute the bits after substitution.

Permuted Choice 1 (PC1) Table: Used to permute the initial key before key scheduling.

Permuted Choice 2 (PC2) Table: Used to permute the combined halves of the key during key scheduling.

Left Shifts (SHIFTS): Specifies the number of left shifts for each round in the key scheduling process.

```python
# Initial Permutation Table
IP = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

# Final Permutation Table
FP = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]

# Expansion Table
E = [
    32, 1, 2, 3, 4, 5, 4, 5,
    6, 7, 8, 9, 8, 9, 10, 11,
    12, 13, 12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21, 20, 21,
    22, 23, 24, 25, 24, 25, 26, 27,
    28, 29, 28, 29, 30, 31, 32, 1
]
```

```
# S-Boxes
S_BOXES = [
    [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],
    ],
    [
        [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
        [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
        [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
        [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],
    ],
    [
        [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
        [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
        [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
        [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
    ],
    [
        [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
        [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
        [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
        [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
    ],
    [
        [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
        [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
        [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
        [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],
    ],
    [
        [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
        [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
        [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
        [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],
    ],
    [
```

```python
# Permutation Table
P = [
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
    19, 13, 30, 6, 22, 11, 4, 25
]

# Permuted Choice 1 Table
PC1 = [
    57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4
]

# Permuted Choice 2 Table
PC2 = [
    14, 17, 11, 24, 1, 5, 3, 28,
    15, 6, 21, 10, 23, 19, 12, 4,
    26, 8, 16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55, 30, 40,
    51, 45, 33, 48, 44, 49, 39, 56,
    34, 53, 46, 42, 50, 36, 29, 32
]

# Left Shifts per Round
SHIFTS = [
    1, 1, 2, 2, 2, 2, 2, 2,
    1, 2, 2, 2, 2, 2, 2, 1
]
```

Helper functions are;

permute: Rearranges bits according to a given table.
split_bits: Splits a bit array into two halves.
left_shift: Performs a circular left shift on a bit array.
xor: Performs a bitwise XOR operation on two bit arrays.
sbox_substitution: Applies the S-Box substitution on a 48-bit block.
initial_permutation: Applies the initial permutation on a 64-bit block.
final_permutation: Applies the final permutation on a 64-bit block.
expand: Expands a 32-bit block to 48 bits using the expansion table.
key_schedule: Generates the 16 round keys from the initial 56-bit key.
str_to_bit_array: Converts a string into a bit array.
bit_array_to_hex: Converts a bit array into a hexadecimal string.
hex_to_bit_array: Converts a hexadecimal string into a bit array.
binvalue: Converts a value to a binary string with a specified bit size.
pad_text: Pads the text to be a multiple of 8 bytes (64 bits).
unpad_text: Removes the padding from the decrypted text.
des_round: Performs one round of the DES algorithm.
des_encrypt_block: Encrypts a 64-bit block using the DES algorithm.
des_decrypt_block: Decrypts a 64-bit block using the DES algorithm.
bit_array_to_str: Converts a bit array back to a string.

You can see helper functions in the next pages.

```
6 usages
def permute(block, table):
    return [block[x - 1] for x in table]


3 usages
def split_bits(bits, n):
    return bits[:n], bits[n:]


2 usages
def left_shift(bits, n):
    return bits[n:] + bits[:n]


2 usages
def xor(bits1, bits2):
    return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]


1 usage
def sbox_substitution(block):
    output = []
    for i in range(8):
        chunk = block[i * 6:(i + 1) * 6]
        row = int(f"{chunk[0]}{chunk[5]}", 2)
        col = int(''.join(map(str, chunk[1:5])), 2)
        sbox_val = S_BOXES[i][row][col]
        output.extend([int(x) for x in f"{sbox_val:04b}"])
    return output


2 usages
def initial_permutation(block):
    return permute(block, IP)


2 usages
def final_permutation(block):
    return permute(block, FP)
```

```
1 usage
def expand(block):
    return permute(block, E)


2 usages
def key_schedule(key):
    key = permute(key, PC1)
    left, right = split_bits(key, n: 28)
    keys = []
    for shift in SHIFTS:
        left = left_shift(left, shift)
        right = left_shift(right, shift)
        combined = left + right
        round_key = permute(combined, PC2)
        keys.append(round_key)
    return keys


3 usages
def str_to_bit_array(text):
    array = []
    byte_array = text.encode('utf-8')  # Encode text to bytes
    for byte in byte_array:
        binval = binvalue(byte, bitsize: 8)
        array.extend([int(x) for x in list(binval)])
    return array


1 usage
def bit_array_to_hex(array):
    hex_str = ''.join([f"{int(''.join(map(str, array[i:i + 4])), 2):x}" for i in range(0, len(array), 4)])
    return hex_str
```

```python
def hex_to_bit_array(hex_str):
    bit_array = []
    for char in hex_str:
        binval = binvalue(int(char, 16), bitsize: 4)
        bit_array.extend([int(x) for x in binval])
    return bit_array


2 usages
def binvalue(val, bitsize):
    binval = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]
    if len(binval) > bitsize:
        raise ValueError("binary value larger than the expected size")
    while len(binval) < bitsize:
        binval = "0" + binval
    return binval


1 usage
def pad_text(text):
    pad_len = 8 - (len(text.encode('utf-8')) % 8)
    return text + chr(pad_len) * pad_len


1 usage
def unpad_text(text):
    pad_len = ord(text[-1])
    return text[:-pad_len]


2 usages
def des_round(left, right, round_key):
    expanded_right = expand(right)
    xored = xor(expanded_right, round_key)
    substituted = sbox_substitution(xored)
    permuted = permute(substituted, P)
    new_right = xor(left, permuted)
    return right, new_right
```

```python
1 usage
def des_encrypt_block(block, keys):
    block = initial_permutation(block)
    left, right = split_bits(block, n: 32)
    for round_key in keys:
        left, right = des_round(left, right, round_key)
    combined = right + left
    return final_permutation(combined)


1 usage
def des_decrypt_block(block, keys):
    block = initial_permutation(block)
    left, right = split_bits(block, n: 32)
    for round_key in reversed(keys):
        left, right = des_round(left, right, round_key)
    combined = right + left
    return final_permutation(combined)


1 usage
def bit_array_to_str(array):
    byte_array = bytearray()
    for i in range(0, len(array), 8):
        byte = int(''.join([str(x) for x in array[i:i + 8]]), 2)
        byte_array.append(byte)
    return byte_array.decode( encoding: 'utf-8', errors='ignore')  # Decode bytes to text, ignore errors


```

The encrypt function encrypts a plaintext message using a given key. It converts the key to a bit array and generates the round keys. It pads the plaintext and splits it into 64-bit blocks. It encrypts each block using the DES algorithm and converts the result to a hexadecimal string.

The decrypt function decrypts a ciphertext message using a given key. It converts the key to a bit array and generates the round keys. It converts the ciphertext from hexadecimal to a bit array and splits it into 64-bit blocks. It decrypts each block using the DES algorithm and converts the result to a string. It unpads the decrypted text.

```
1 usage
def encrypt(text, key):
    key = str_to_bit_array(key)
    keys = key_schedule(key)
    text = pad_text(text)
    blocks = [text[i:i + 8] for i in range(0, len(text), 8)]
    encrypted_blocks = []
    for block in blocks:
        block = str_to_bit_array(block)
        if len(block) != 64:
            block.extend([0] * (64 - len(block)))  # Pad the block to 64 bits if necessary
        encrypted_block = des_encrypt_block(block, keys)
        encrypted_blocks.extend(encrypted_block)
    return bit_array_to_hex(encrypted_blocks)


1 usage
def decrypt(ciphertext, key):
    key = str_to_bit_array(key)
    keys = key_schedule(key)
    bit_array = hex_to_bit_array(ciphertext)
    blocks = [bit_array[i:i + 64] for i in range(0, len(bit_array), 64)]
    decrypted_blocks = []
    for block in blocks:
        decrypted_block = des_decrypt_block(block, keys)
        decrypted_blocks.extend(decrypted_block)
    decrypted_text = bit_array_to_str(decrypted_blocks)
    return unpad_text(decrypted_text)
```

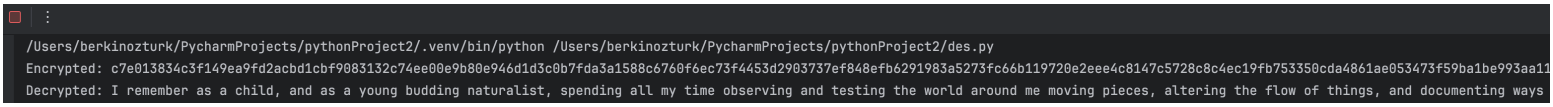
The key is an 8-byte string, which is required for DES. The key "berkinso" is used to generate the round keys for the encryption and decryption processes.

The encrypt function converts the key into a bit array and generates the round keys. The plaintext is padded to ensure its length is a multiple of 8 bytes. The padded plaintext is split into 64-bit blocks, and each block is encrypted using the DES algorithm. The encrypted blocks are concatenated and converted into a hexadecimal string.

The decrypt function converts the key into a bit array and generates the round keys. The ciphertext is converted from hexadecimal to a bit array and split into 64-bit blocks. Each block is decrypted using the DES algorithm, and the decrypted blocks are concatenated. The resulting bit array is converted back into a string and unpadded to remove any padding added during encryption.

```
# Usage
plaintext = "I remember as a child, and as a young budding naturalist, spending a
key = "berkinso"  # 8-byte key
ciphertext = encrypt(plaintext, key)
print("Encrypted:", ciphertext)
decrypted_text = decrypt(ciphertext, key)
print("Decrypted:", decrypted_text)
```

We can see the encrypted and decrypted text in the below.

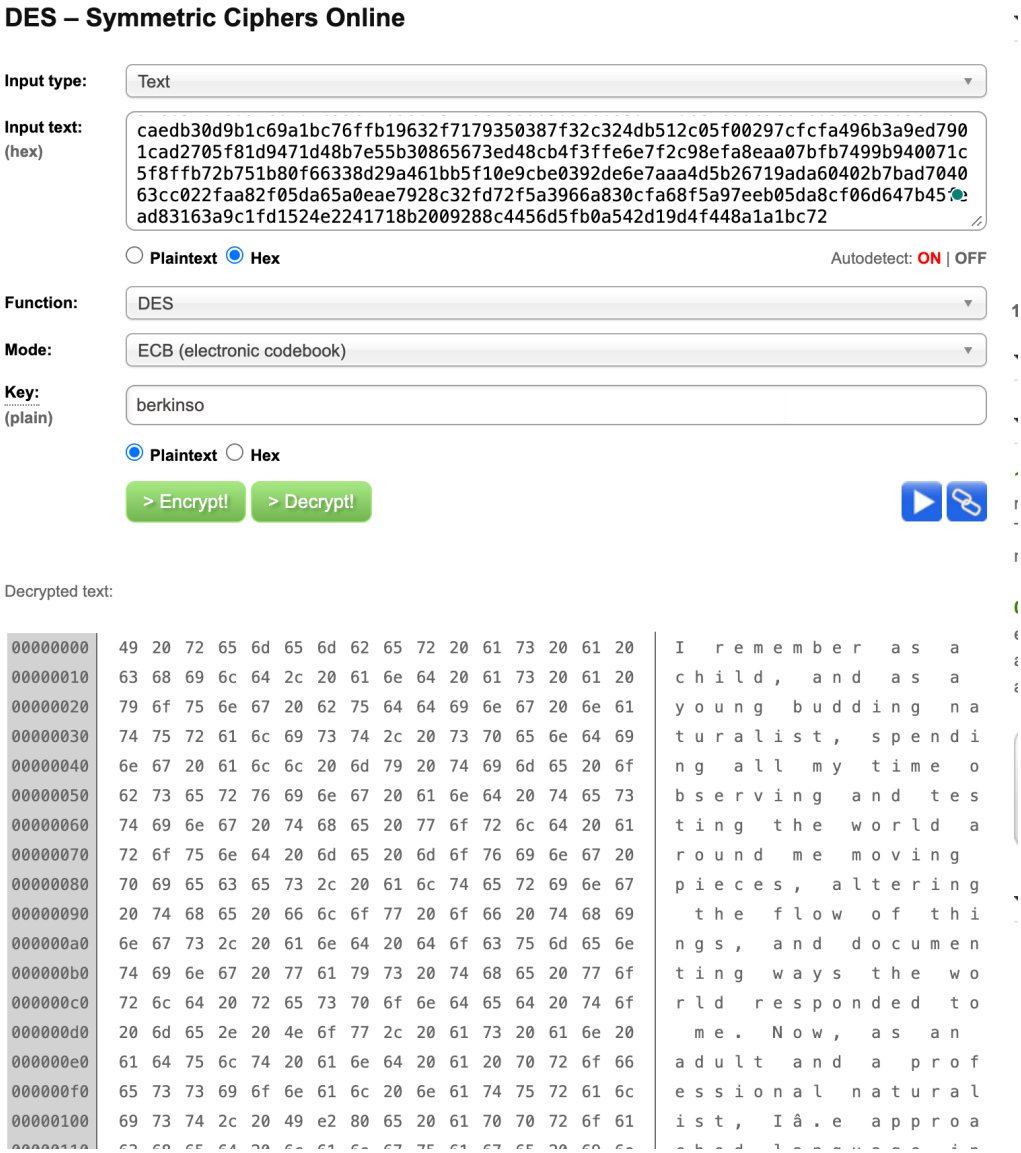



The full encrypted text is:

c7e013834c3f149ea9fd2acbd1cbf9083132c74ee00e9b80e946d1d3c0b7fda3a1588c6760f6
ec73f4453d2903737ef848efb6291983a5273fc66b119720e2eee4c8147c5728c8c4ec19fb75
3350cda4861ae053473f59ba1be993aa1153db0b61425b85de9f39de57ff77532a913364dd2
6862aad8780679b8a7d2fde982b720dbdbb257fc5958b1c3def37ea0b57a6cf72e0115ed364
ebc9bea4ac6f58e446ccca177935729a53f55601fb5dc06fc5122404103096c31cfaabe64019
b42ad20d2944ad9ba386ee32cd464a09b73821fe4efcec2f27a2df40958287a1900fd2fd3550
1a98e379dae893eab997459b685831804bd8a59de09856ad745cdc7960d93ff64aaa9455ae6
80718ab10611575b044cba6fedc1d10fb17a21f13d55d1fe5ad000bb20734eb9a7559a5ea14
77941b8fa5f27966990d1f7a0e11d7fe6575f014e54379b86ef8768f5708f7a43575680f560ff
3c44466c95be6a0e505ed93e622e9f3fc52efe0a1b319a25400c70d570ce901abd1cf68dda8
08e3f8ff963408de55f96196d08ec17064b7a18bdab66b15cf1eef0e8a2c2c51792e256b8d1a
9b90ec88cdb20465aa130b349cd8c31fce05c161f87b30cbaf25fbc9058d20618a3d4dee583
7c1fbeb2bcbefb9e416e79856ad745cdc7960fab78c5ca8539edb9c12ef26e754bb2bf90393e
5c90a8e5a96e398b93c78440aad9a7cf0d1ff1c2729b9022e330453a606b478df4ccb09a00af
2ac4ac15574badbddba8aa88a0d23b2383ebd694914e79babe90f90a37a203a8bbcbf984100
857b079eace66b2250dba9c856602a5e56f1dbe570db93d20a348260597cb10613d7cb211b
f4caa95c619958ffc1db116bcca23251c9e14c6f63108fd237be7f032c22d49884c4ab1b7410
aa9700e70e2af55601fb5dc06fc594e16be598e955d6586e06f456f698fca630f5bf76ebd44a
c1abf5c80f37e684a6465e2a25fddc03e480579fa28fa4d8aa9dbaac2e79fd1cfe1c0b73ebdfd
ee5239cd170f812bb999f86dc8172f73b2495bc453de86933e0e4a5852b8bec40c5761e4d1
5357923786ae555d70ed40303b2d1204cc508ef3af457576635ee6343587dd1e6bb551a4a9

b4cf3661ee5d7aec90f70d9efcef8f60bcfb92807f0840a2c54f05819e9b9dee7bae6f1898f81
15effd0c91771e0cf50693f60e603d4447368547ba2ff8e7a0f5328f55b86af6b217ac550d57
ed778c8048237c555c9c6d28a93d59c181dfe9e4e9f37cdc47072f2dd52616987f75a5d2564
df7e2ff49b184f64c071fd7d3c60ee740cc2f4c671e1914e47321bc6bf3a2d2d304ef25ddab5
8f7490af0355e102b769ab903c9a99c4764efabda1cf4960392261b95158422811f1bdcc470
6ef7956b149cac0786611fa736f41dddf64c46913bb1f83d4077f1bd6f8e45c5a24963cb9454
04c60b43afe35245f5475ffafb1ba298b796bdfb06b478df4ccb09a06d47e519f5740ae23d65
8b498b85506c93a91fac31768be53089ce9083d5dcb9c676ac4be42a89e1cda9cc29b195c8
68a63efa4e756a9c2b3463cdec600e199f1e0200ba85a988927526e6ea9100c94b28a035513
7b9179ab4704bb9860a6cb74fb18e339eac3227aa05d77dc9011893804ad1b1b821562c376
8691dd11ea39ba7c8a5956093127ba359359815397895cc55c0e22626a28c481de15c73687
568d2b7d0964def316394cf68c86a77ca3e5dee35ccfa61ba55754ea3663f251c876d7f9964
b99a27a6d17e394315de65f61592e7107a3bfb185ced8898cf1d184feb6fdfdb7783068aa9ca
bbf79d1b6d9fa7873e28c6142613becf821d62e9f36e0fed7bfa334374fe2773c62eab9a0490
39a7f1235b482ac83bef070e14cecd842cf7463b8b6462787b396c8eb9db6f06f8166219ce7
639f7125e34ad2efbd625b59b53001b0f20e4c350b921cf8650dd3c383b45881855ac0a29ec
de4343a626aa4dd6a57a9f153623a64a580c4c97da8c6aa1089e9bce31095167809f30870b7
89da8340c70f521718fd17db1f531f126f9a80b95bfbc7db804cc32025f0708bdebedff0ebc6
a1186a3c93cda5db54f3677c8024ad9e5026833bc95f4dff24781b19cc0b872c5737e6c8b01
f5091635ac3880f6f9b21dbc6df6d21dc201fe59680eb7882a595a6817699e1527e493a2b94
12c8dd9f548dc52334b9827ea1f73ca5fedd5bc419a16abfd9838be40b174623fe1c1eb16d5
5250aaed3e9a368afbc2f64f309dbfe206d737369c1cb7388c81fa9dc45f9a691aff7b0f04dbb
18768d0df1155e0a53184cec79804522367889aab90885fa5e1e4ace17b6d032a927a7cb326
16d92c61d1c99c220ad04311acf80005d9b8b83d1ff0ce2ed0cabefed9ee33903c7194caedb3
0d9b1c69a1bc76ffb19632f7179350387f32c324db512c05f00297cfcfa496b3a9ed7901cad2
705f81d9471d48b7e55b30865673ed48cb4f3ffe6e7f2c98efa8eaa07bfb7499b940071c5f8ff
b72b751b80f66338d29a461bb5f10e9cbe0392de6e7aaa4d5b26719ada60402b7bad704063
cc022faa82f05da65a0eae7928c32fd72f5a3966a830cfa68f5a97eeb05da8cf06d647b45fead
83163a9c1fd1524e2241718b2009288c4456d5fb0a542d19d4f448a1a1bc72

We can prove that the process is successful when we control it using a DES decryption tool from the internet. You can see in the below.



I tried to break different ciphertexts using the brute force method, but I did not get any results.

```
Encrypted: 48f8ddac080a0698
Decrypted: hello
Brute-forcing DES:   0%|          | 225850877/218340105584896 [27:20<434927:51:00, 139448.34key/s]
```

When I researched, I came to some conclusions about the breaking of DES. The time required to break DES depends on the processing power of the computer used and the attack method applied.

## Attacking Symmetric Encryption

- **Average Time Required** for Exhaustive Key Search, assuming that it takes 1 ns to perform a single decryption.

| Key Size (bits) | Cipher | Number of Alternative Keys | Time Required at 1 Decryption / ms | Time Required at $10^9$ Decryption / ms |
|---|---|---|---|---|
| 56 | DES | $2^{56} = 7.2 \times 10^{16}$ | $2^{55}$ ns = 1125 years | 1 hours |
| 128 | AES | $2^{128} = 3.4 \times 10^{38}$ | $2^{127}$ ns = $5.4 \times 10^{21}$ years | $5.4 \times 10^{17}$ years |
| 168 | Triple DES | $2^{168} = 3.7 \times 10^{50}$ | $2^{168}$ ns = $5.8 \times 10^{33}$ years | $5.8 \times 10^{29}$ years |
| 192 | AES | $2^{192} = 6.3 \times 10^{57}$ | $2^{191}$ ns = $9.8 \times 10^{40}$ years | $9.8 \times 10^{36}$ years |
| 256 | AES | $2^{256} = 1.2 \times 10^{77}$ | $2^{255}$ ns = $1.8 \times 10^{60}$ years | $1.8 \times 10^{56}$ years |

Prof. Dr. Rand Kouatly, Software Optimization - University Europe, of Applied Sciences 2024 93

The duration of a brute force attack depends on the processing power of the computers available.

DES is vulnerable to brute force attacks with today's processing power due to its 56-bit key length. While a brute force attack can take longer time with an average modern computer, this time can be reduced to minutes with special hardware or supercomputers.

I did not include it in my code because I could not break the encryption of the code with brute-force by waiting for a long time.

# PART 4 - Bitmap Image Encryption

DES (Data Encryption Standard) bitmap image encryption and decryption secures image data using the DES algorithm. During encryption, each pixel value of the bitmap image is processed block by block with the DES algorithm; Each block consists of 64 bits of data and is encrypted using a key. The encrypted data is converted into an encrypted bitmap file, preserving the original layout of the image. The decryption process works in reverse; The encrypted bitmap file is decrypted block by block using the same key and the original pixel values are restored. This process is possible thanks to the symmetric structure of the DES algorithm, meaning the same key is used for both encryption and decryption. This way, the bitmap image can be transmitted securely and read only by authorized individuals with the correct key.

We use the same constant values and helper functions for DES encryption here, so I don't add them here again as a screenshot.

I explain the steps on how we save the bitmap image:

The code in below defines three functions to load original, encrypted, and decrypted images respectively. These functions ensure that the images are in RGB mode:

```
157  # Step 1: Load the bitmap images
     1 usage
158  def load_image(image_path):
159      image = Image.open("sample_640×426.bmp")
160      if image.mode != 'RGB':
161          image = image.convert('RGB')
162      return image
163
164
     1 usage
165  def load_encrypted_image(image_path):
166      image = Image.open("encrypted_image.bmp")
167      if image.mode != 'RGB':
168          image = image.convert('RGB')
169      return image
170
171
     1 usage
172  def load_decrypted_image(image_path):
173      image = Image.open("decrypted_image.bmp")
174      if image.mode != 'RGB':
175          image = image.convert('RGB')
176      return image
```

The function image_to_byte_array converts the image into a flattened byte array:

```
78
79      # Step 2: Convert the image to a byte array
        1 usage
80      def image_to_byte_array(image):
81          image_bytes = np.array(image).flatten().tolist()
82          return image_bytes
83
```

The function encrypt_image encrypts the byte array using DES encryption. It handles padding to ensure that the data length is a multiple of 8 bytes, which is required by DES. DES encryption works on 64-bit blocks. Therefore, the length of the image byte array must be a multiple of 64 bits (8 bytes). If not, padding is added to fill in the missing part. The byte array is divided into blocks of 8 bytes. Each block and key is converted into a sequence of bits. The DES algorithm encrypts each block. The encrypted bits are converted back to bytes and added to the encrypted byte array:

```
184
185     # Step 3: Encrypt the byte array using DES
        1 usage
186     def encrypt_image(byte_array, key):
187         padded_size = len(byte_array) + (8 - len(byte_array) % 8) if len(byte_array) % 8 != 0 else len(byte_array)
188         byte_array += [0] * (padded_size - len(byte_array))  # Padding
189         encrypted_bytes = []
190         for i in range(0, len(byte_array), 8):
191             block = byte_array[i:i + 8]
192             block_bits = bytes_to_bits(block)
193             key_bits = bytes_to_bits(key)
194             encrypted_bits = des_encrypt(block_bits, key_bits)
195             encrypted_block = bits_to_bytes(encrypted_bits)
196             encrypted_bytes.extend(encrypted_block)
197         return encrypted_bytes, padded_size
198
```

The function save_encrypted_image saves the encrypted byte array back to an image file. The encrypted byte sequence is converted to a NumPy array to match the dimensions of the original image and saved as an encrypted image:

```
# Step 4: Save the encrypted byte array as an image
1 usage
def save_encrypted_image(encrypted_bytes, image_size, output_path):
    encrypted_image_array = np.array(encrypted_bytes[:image_size[0] * image_size[1] * 3], dtype=np.uint8).reshape(
        image_size[1], image_size[0], 3)
    encrypted_image = Image.fromarray(encrypted_image_array)
    encrypted_image.save(output_path)
```

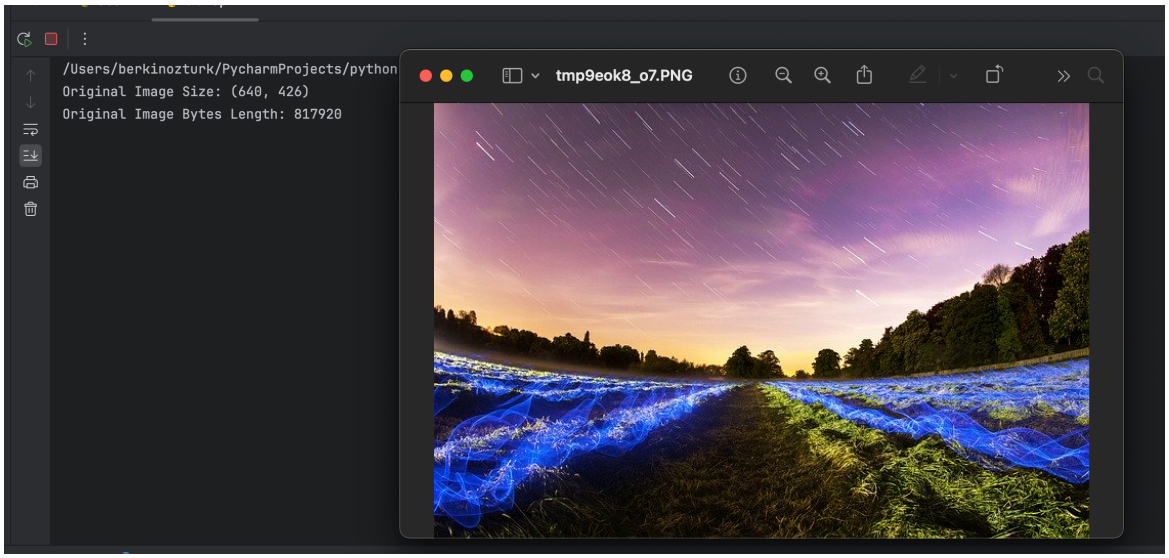
The encrypted byte sequence is divided into blocks of 8 bytes. Each block and key is converted into a sequence of bits. The DES algorithm decodes each block. The decoded bits are converted back to bytes and added to the decoded byte array. Padding is removed to reach the original size. The decoded byte array is converted to a NumPy array to match the original image dimensions and saved as a decoded image.

```
# Step 5: Decrypt the encrypted image
1 usage
def decrypt_image(encrypted_bytes, key, original_size):
    decrypted_bytes = []
    for i in range(0, len(encrypted_bytes), 8):
        block = encrypted_bytes[i:i + 8]
        block_bits = bytes_to_bits(block)
        key_bits = bytes_to_bits(key)
        decrypted_bits = des_decrypt(block_bits, key_bits)
        decrypted_block = bits_to_bytes(decrypted_bits)
        decrypted_bytes.extend(decrypted_block)
    return decrypted_bytes[:original_size]
```

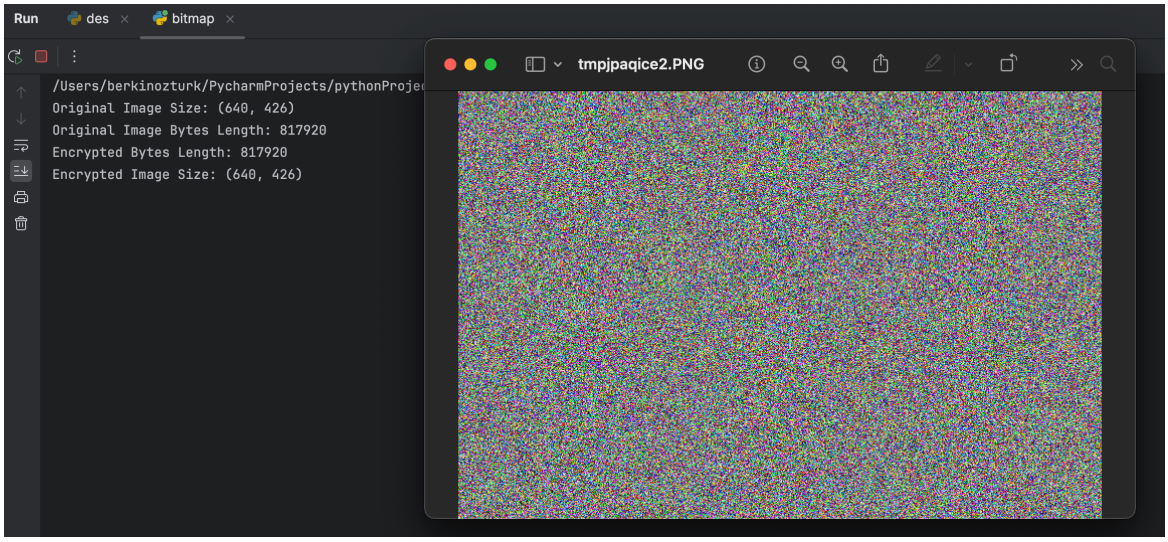
```
# Load original image
original_image_path = 'sample_640x426.bmp'
original_image = load_image(original_image_path)
print(f"Original Image Size: {original_image.size}")
original_image.show()

# Convert image to byte array
image_bytes = image_to_byte_array(original_image)
print(f"Original Image Bytes Length: {len(image_bytes)}")

# Key for DES (must be 8 bytes long)
key_hex = "133457799BBCDFF1"
key = bytes.fromhex(key_hex)

# Encrypt the byte array
encrypted_bytes, padded_size = encrypt_image(image_bytes, key)
print(f"Encrypted Bytes Length: {len(encrypted_bytes)}")

# Save encrypted image
encrypted_image_path = 'encrypted_image.bmp'
save_encrypted_image(encrypted_bytes, original_image.size, encrypted_image_path)

# Load and show encrypted image
encrypted_image = load_encrypted_image(encrypted_image_path)
print(f"Encrypted Image Size: {encrypted_image.size}")
encrypted_image.show()

# Decrypt the byte array
decrypted_bytes = decrypt_image(encrypted_bytes, key, len(image_bytes))
print(f"Decrypted Bytes Length: {len(decrypted_bytes)}")

# Convert decrypted byte array to image and save
decrypted_image_array = np.array(decrypted_bytes, dtype=np.uint8).reshape(original_image.size[1],
                                                                          original_image.size[0], 3)
decrypted_image = Image.fromarray(decrypted_image_array)
decrypted_image_path = 'decrypted_image.bmp'
decrypted_image.save(decrypted_image_path)

# Load and show decrypted image
decrypted_image = load_decrypted_image(decrypted_image_path)
print(f"Decrypted Image Size: {decrypted_image.size}")
decrypted_image.show()
```

The original_image_path variable specifies the file path of the bitmap image to be loaded. The load_image function loads the image using this file path and converts it to RGB mode if necessary. The print command prints the dimensions (width and height) of the loaded image. The original_image.show() command shows the loaded image on the screen. The image_to_byte_array(original_image) function converts the pixels of the image into a plain byte array. The print(f"Original Image Bytes Length: {len(image_bytes)}") command prints the length of the created byte array.

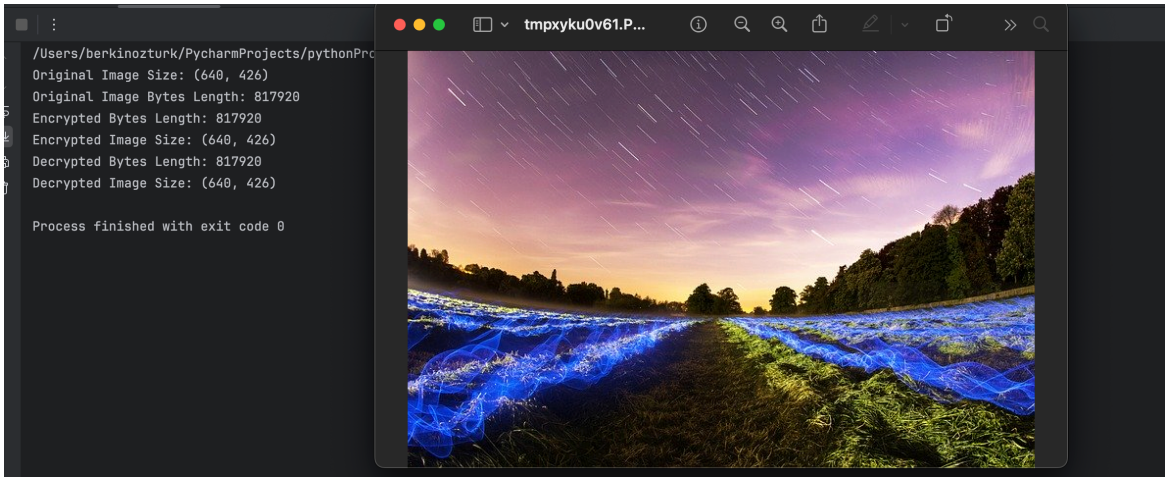
The key_hex variable specifies the 16-character (8 byte) hex format key to be used for DES encryption. The bytes.fromhex(key_hex) command converts the key in hex format to byte format.



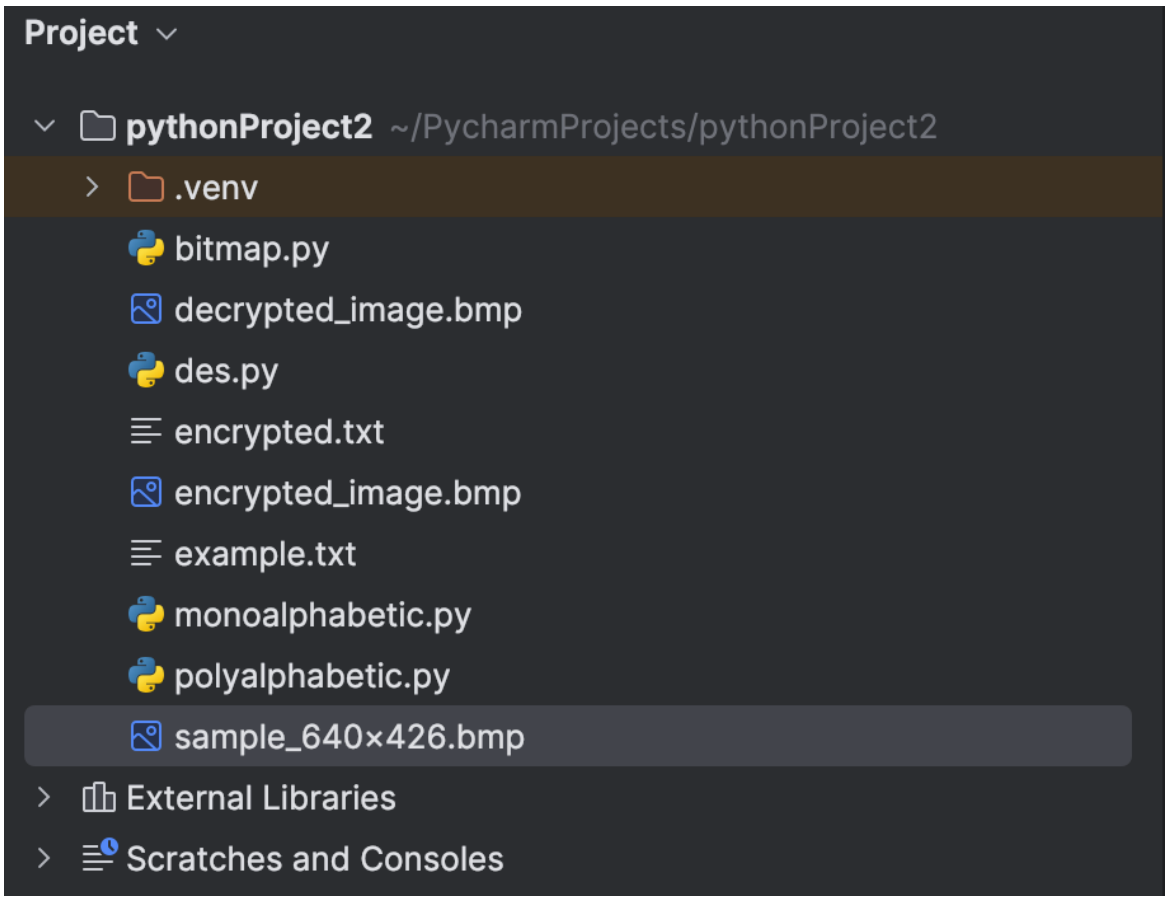
Encryption is done and the bitmap image is saved. The dimensions of the loaded encrypted image are printed and the loaded encrypted image is displayed on the screen.




The decryption process is performed in the same way and the decrypted text appears with its dimensions.



You can also see that encrypted and decrypted images are saved in the file path with .bmp extension.




Since the encryption method used is the same (DES), cracking the encrypted bitmap image is the same as my answer in the previous part.

# CONCLUSION

In this academic report, I explored various encryption techniques, each with unique strengths and vulnerabilities. Through these investigations, it becomes evident that each encryption method has its own unique strengths and weaknesses, which are crucial in understanding their practical applications and limitations.

The first part of the report focused on the Monoalphabetic Substitution Cipher, highlighting its simplicity and susceptibility to frequency analysis attacks. Through practical implementation, we demonstrated how frequency analysis can effectively break this cipher, emphasizing the need for more robust encryption methods.

In the second part, we delved into the Polyalphabetic Substitution Cipher, which improves security by using multiple substitution alphabets. This method significantly mitigates the vulnerabilities seen in monoalphabetic ciphers, particularly against frequency analysis. However, I also showed that brute-force attacks remain a feasible threat, though the complexity increases with the length of the key.

The third part introduced the Data Encryption Standard (DES), a widely recognized symmetric key algorithm. I implemented DES for both text and bitmap images, illustrating its encryption and decryption processes. Despite its historical significance and widespread use, DES's 56-bit key length makes it vulnerable to brute-force attacks with modern computing power, highlighting the evolution towards more secure encryption standards like AES.

As a continuation of the previous section, I applied DES to encrypt and decrypt bitmap images, demonstrating the versatility and practical application of cryptographic techniques in securing digital media. This task reinforced the importance of encryption in maintaining data confidentiality and integrity in various formats.

Overall, the exercises underscored the critical balance between encryption complexity and computational feasibility. While simpler ciphers like the Monoalphabetic Substitution are easily broken, more complex methods like Polyalphabetic Substitution and DES offer greater security but at the cost of increased computational requirements. These insights are crucial for developing and selecting appropriate cryptographic methods in software engineering and cybersecurity.

This report effectively documents the implementation processes, results, and analysis, providing a thorough understanding of each encryption technique's capabilities and limitations.