THE MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

INTERNATIONAL INFORMATION TECHNOLOGIES UNIVERSITY

Faculty of Information technologies

Department of Computer Systems, Software Engineering and Telecommunications

Project report

Course: Information theory

Students: Arsen Yerbol, Yersain Makazhanov

Group: CSSE - 1608K

Accepted : senior-lecturer L.A. Kozina

Almaty 2019

CONTENT

INTRODUCTION 3
PART 1 4
PART 2 6
PART 3 8
PART 4 9
PART 5 11
PART 6 12
CONCLUSION 14
REFERENCES 15
APPENDIX 16

INTRODUCTION

This project is done specially for “Information Theory” course by 3rd year students at International Information Technologies University.

The aim was to write a code of Communication System divided into 6 parts. Project required the knowledge of one programming language(data structures, basic functions) and encoding algorithms(Shannon-Fano, Huffman, Hamming(7, 4)). The last was given as a requisite of a course itself.

During the half of semester we developed a whole structured project as shown in **Figure 1**.

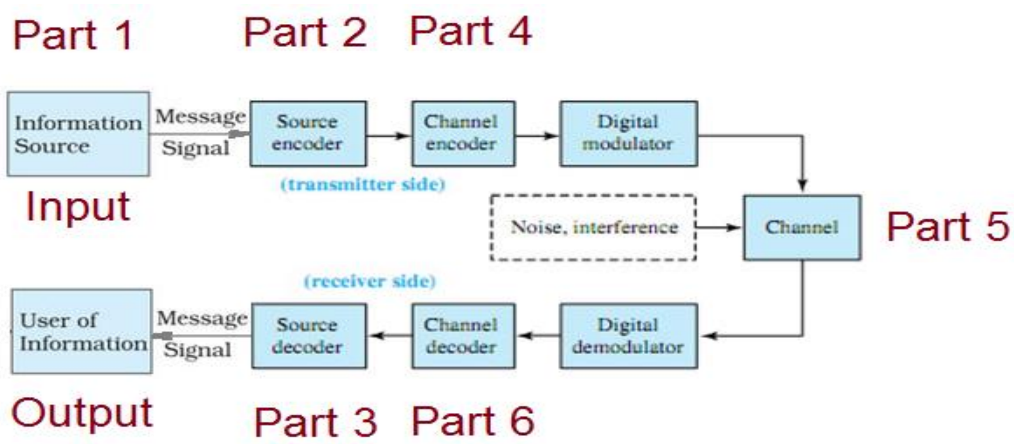

Figure – 1 – Communication system.

We chose **Python** programming language, because all team members were familiar and confident with it. As an IDE we chose **jupyter notebook** due to the same reason.

PART 1

The first part of a project required “Text.txt” file as an input and probabilities of each unique symbol as an output.

Example: A – 0,014, B – 0,012.

Figure 2 shows the beginning of a project. Firstly, we imported dependencies that we need.

Numpy for arrrays and random for error generating.

We made a closer look on text. Then created a dictionary with all unique symbols in it, and finally counted probabilities of each.

PART 1

```
In [1]: import numpy as np
        import random

In [2]: f = open("text.txt", "r")

In [3]: text = f.read()[:-1]

In [4]: text

Out[4]: 'In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exa
        m. The professor, Robert M.Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman,
        unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon
        the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the stud
        ent outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffm
        an avoided the major flaw of the suboptimal Shannon-Fano coding by building the tree from the bottom up instead of from t
        he top down.'

In [5]: keys = list(set(text))

In [6]: values = np.zeros((len(keys),), dtype = int)

In [7]: d = dict(zip(keys, values))

In [8]: for char in text:
            d[char]+=1

In [9]: for key in d:
            d[key] = d[key] / len(text)

In [10]: d = sorted(d.items(), key=lambda x: x[1], reverse = True)
```

Figure – 2 – Code for part1.

```
In [11]: d

Out[11]: [(' ', 0.17185385656292287),
          ('e', 0.08795669824086604),
          ('o', 0.07713125845737483),
          ('t', 0.06765899864682003),
          ('n', 0.06359945872801083),
          ('a', 0.056833558863328824),
          ('i', 0.05548037889039242),
          ('r', 0.044654939106901215),
          ('h', 0.03924221921515562),
          ('f', 0.03924221921515562),
          ('d', 0.037889039242219216),
          ('s', 0.035182679296346414),
          ('m', 0.02706359945872801),
          ('u', 0.023004059539918808),
          ('p', 0.02029769959404601),
          ('c', 0.016238159675236806),
          ('l', 0.016238159675236806),
          ('b', 0.013531799729364006),
          ('y', 0.012178619756427604),
          ('w', 0.012178619756427604),
          ('g', 0.012178619756427604),
          ('v', 0.010825439783491205),
          ('.', 0.009472259810554804),
          (',', 0.009472259810554804),
          ('H', 0.0040595399188092015),
          ('I', 0.0040595399188092015),
          ('S', 0.0027063599458728013),
          ('T', 0.0027063599458728013),
          ('k', 0.0027063599458728013),
          ('-', 0.0027063599458728013),
          ('M', 0.0027063599458728013),
          ('1', 0.0027063599458728013),
          ('q', 0.0027063599458728013),
          ('F', 0.0027063599458728013),
          ('j', 0.0013531799729364006),
          ('5', 0.0013531799729364006),
          ('D', 0.0013531799729364006),
          ('C', 0.0013531799729364006),
          ('A', 0.0013531799729364006),
          ('9', 0.0013531799729364006),
          ('R', 0.0013531799729364006),
          ('x', 0.0013531799729364006)]
```

Figure – 3 – Output of dictionary.

PART 2

The goal of 2nd part was to encode the input text to the sequence of a binary digits according to our probabilities.

Here we chose Huffman encoding, because it seemed more interesting than Shannon-Fano and of course, it is more reliable.

We developed our own algorithm of Huffman encoding without using of Trees and Nodes. We started giving 0s and 1s just from the beginning and did not wait until we get all symbols combined.

For example, lets say that we have A – 0.5 and B – 0.3. Our algorithm in such case, firstly gives 1 to A and 0 to B. The main advantage of such algorithm is that we are winning in terms of memory, because it is not using hashes and Trees.

As a result we got encoded text in binary digits.

PART 2

```
In [12]: d2 = dict(d.copy())

In [13]: i = len(d) - 2

In [14]: for x in d2.keys():
             d2[x] = ''

In [15]: while i!=-1:
             first = d[i][0]+d[i+1][0]
             second = d[i][1]+d[i+1][1]
             a = (first, second)
             d.append(a)
             if d[i][1]>d[i+1][1]:
                 for key, value in d2.items():
                     if key in d[i][0]:
                         d2[key] += '1'
                     elif key in d[i+1][0]:
                         d2[key] += '0'
             else:
                 for key, value in d2.items():
                     if key in d[i][0]:
                         d2[key] += '0'
                     elif key in d[i+1][0]:
                         d2[key] += '1'
             d = dict(d)
             d = sorted(d.items(), key=lambda x: x[1], reverse = True)
             del d[-1]
             del d[-1]
             i-=1
         print(d)

    [(' pDCj5RxA9.hfdos,HIctnalbmiywgvruk-STqFM1e', 1.0)]
```

Figure – 4 – Huffman.

```
In [16]: d2

Out[16]: {' ': '111',
 'e': '000',
 'o': '1101',
 't': '1001',
 'n': '0001',
 'a': '1110',
 'i': '1010',
 'r': '0100',
 'h': '01011',
 'f': '10011',
 'd': '00011',
 's': '10101',
 'm': '00110',
 'u': '11100',
 'p': '011011',
 'c': '000101',
 'l': '110110',
 'b': '010110',
 'y': '010010',
 'w': '110010',
 'g': '100010',
 'v': '000010',
 '.': '0111011',
 ',': '1100101',
 'H': '00100101',
 'I': '10100101',
 'S': '01001100',
 'T': '11001100',
 'k': '00001100',
 '-': '10001100',
 'M': '01101100',
 '1': '11101100',
 'q': '00101100',
 'F': '10101100',
 'j': '0101111011',
 '5': '1101111011',
 'D': '0001111011',
 'C': '1001111011',
 'A': '0111111011',
 '9': '1111111011',
 'R': '0011111011',
 'x': '1011111011'}
```

Figure – 5 – Binary representation.

```
In [17]: for key in d2.keys():
             d2[key] = d2[key][::-1]

In [18]: output = ''
         for word in text:
             for char in word:
                 output+=d2[char]

In [19]: output

Out[19]: '1010010110001110011011111011111111101111011001101111010011111110111100001110100000101110001111101111110110111011111010010
         0001111100111001011000111100011101111000110001111101001011010111100110110101001010011001111101011000110011011001001100011
         1100101011011100001111001110100001011001001001011110100001101101111010110101011000111100100010101111010011000001000011101 0
         0010101010000000100011110011101000011110100011010101101011010000000111101111001111011111110010000010011001111101100111110 1
         1000000101111011001011101111111100101011000011101101111100011011111010111011001101110111001100111101000011111011000101011
         1100100010101101011011001010100111111101111100101101101000000101001111001101101101110001101010111100010111010011110111 10
         1011010101010100011000000110001110111111100100000100110011111011001111101100000010111101110001111001110100001111101100010
         1011011010011011000011001111011110011111100101011000110000101100001000111110011101000011101100101110101100111100011001110
         0101011010000101000100010011110110100101100001110010010010111101000101111000000110111011110100100001111100110010110001 11
         1000101001111100111100000111011010011011000111100110111111110110001010110100000000111011110000100101111010001011110000001010
         1111010011000000100001111001110100001110110010111010110011110001100111001010110100001010001000100110100111110100110111101 0
         1111011101101010110011110011111001101111101000101010100000000111001111101101110111100011000111101011001011100101001111 1010
         1100100111110000100100101100001000111111001101100101111001110100001111100101011000011101101111101001111010000100011111010
         0001111101001011001111001111101101011100011110011101000011101011100000001111111011110011110011110101010110000100011110111
         1111100100100000011010000111000100010100001001000110001101011011001010010001100011101101001011000011100100100101111001001
         0000000111011110001100011100110100001110101101000001100000110110100101111101100010101101000000001100011110011101001011010 1
         1110110000010011101010111100011110011101000011101100101110101100111100011001110010101101000010100010001001110111011110100
         1011000111110001011010110000100011111010110111010011111100111010000111101011001001111100000001000100111110110011110011100 0
         0101110001111101001011010111111101100010101111001000101011010110110010101001111101001111010101111111010011111000111010011 1
         0110010001100000001100011101001101011001110101110101100011001101100100110001111001010110111000111100111010000101100100100
         1011101011000010000000100010011011001011111011110010110110111001111100000001110011001011010011110001000101110001111001101 1
         1111100000000100000000110111011110110111011111111010101010110001010110110111001011110100010111100000001101110111101001000011
         1110011100101100011110001110111010000101101011100000001100011110011101000011101100011111011110101011001011111001011011 0111
         0100111111011110011111001110100001111010100111011010101111011010010101011000111011011111001100101101001111000100010111000
         0011000100110101011110001011111101000101111000010110000100011110110100100101110110100011101010110111100001011000010001111
         1001110100001111001001000000011111001001010110110011110011101000011101101010111001100110110110011100111110110111010110001
         0101100100001111100011110111100111111001001010110110011110011101000011110011011110110111110001011010011100011 01110'
```

Figure – 6 – Binary representation of text.

PART 3

The aim of this part was to decode back sequence of binary digits into initial text. It was pretty easy, we were adding each digit until it matches the first value in our dictionary. You can see the output on Figure N.

```
In [22]: char = ''
         sentence = ''
         for i in output:
             char+=i
             if char in d2.values():
                 sentence+=list(d2.keys())[list(d2.values()).index(char)]
                 char = ''
```

```
In [23]: sentence
```

```
Out[23]: 'In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exa
m. The professor, Robert M.Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman,
unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon
the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the stud
ent outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. Huffm
an avoided the major flaw of the suboptimal Shannon-Fano coding by building the tree from the bottom up instead of from t
he top down.'
```

Figure – 7 – Decoding text back.

PART 4

In this part we encoded our sequence of binary digits from part 2 with the use of Hamming(7, 4) code. We divided the sequence into blocks by 4 and the left symbols saved in a variable, because it is out of encoding (if the length of sequence cannot be divided by 4 with the mod 0).

Then we calculated the sets of parity bits with Formula 1 and added them into our blocks.

Finally we concatenated all blocks and left symbols to get the output.

$$r1 = i1 \text{ XOR } i2 \text{ XOR } i3 \quad (1)$$

$$r2 = i2 \text{ XOR } i3 \text{ XOR } i4 \quad (2)$$

$$r3 = i1 \text{ XOR } i2 \text{ XOR } i4 \quad (3)$$

where r1,r2,r3 – parity bits , i1,i2,i3,i4 – data bits.

```
In [28]: trios = []
         for i in range(len(blocks_4)):
             t_1 = int(blocks_4[i][0]) ^ int(blocks_4[i][1]) ^ int(blocks_4[i][2])
             t_2 = int(blocks_4[i][1]) ^ int(blocks_4[i][2]) ^ int(blocks_4[i][3])
             t_3 = int(blocks_4[i][0]) ^ int(blocks_4[i][1]) ^ int(blocks_4[i][3])
             t = (str(t_1)+str(t_2)+str(t_3))
             blocks_4[i]+=t

In [29]: blocks_4

Out[29]: ['1010011',
          '0101100',
          '1000101',
          '1110100',
          '0110001',
          '1111111',
          '1011000',
          '1111111',
          '1110100',
          '1111111',
          '0110001',
          '0110001',
          '1111111',
          '0100111',
          '1111111',
          '1110100',
          '1111111',
          '0000000',
          '1110100',
```

Figure – 8 – Appending parity bits.

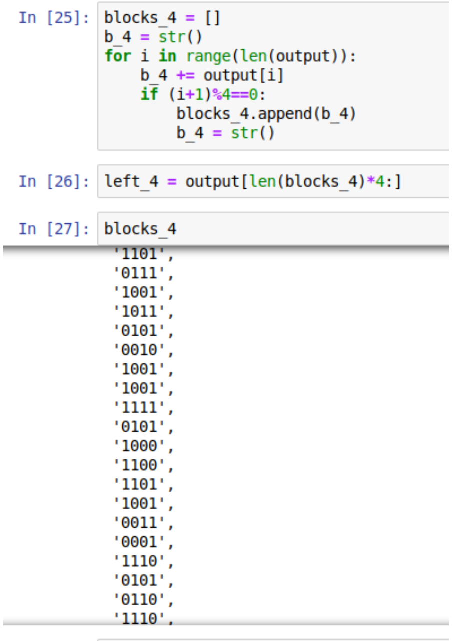

Figure – 9 – Dividing into blocks.

Figure – 10 – Binary representation of text with parity bits.

PART 5

Here we added 1 error to each block. It was required to make a random index and put the reverse number on the place of that index in block. The output looks like this.

```
In [33]: for i in range(len(blocks_5)):
             index = random.randint(0, 6)
             part = list(blocks_5[i])
             if part[index]=='0':
                 part[index] = '1'
             else:
                 part[index] = '0'
             blocks_5[i] = ''.join(part)

In [34]: blocks_5

Out[34]: ['1010111',
          '0100100',
          '1010101',
          '1111100',
          '0110000',
          '1111101',
          '1011001',
          '1111101',
          '1100100',
          '1011111',
          '0110000',
          '0110011',
          '1111011',
          '0100110',
          '0111111',
          '1110101',
          '1011111',
          '0010000',
          '0110100',
```

Figure – 11 – Adding errors.

```
In [35]: output_5 = str()
         for i in blocks_5:
             output_5+=i
         output_5+=left_5

In [36]: output_5

Out[36]: '1010111010010010101011111100011000011111011011001111110111001001011111011000001100111111011010011001111111110101101111100
```

Figure – 12 – Binary representation of text with errors.

PART 6

Final part of a project had a purpose of decoding back everything. We firstly calculated error syndromes with Formula 2. Then with the use of that syndromes we came to the first blocks of 4 digits without parity bits and errors. Furthermore, we combined all blocks and left digits to generate a sequence of binary digits as it was in part 2. Finally we decoded it with the algorithm wrote in part 3. As a result we got our initial text.

$$S1 = r1 \text{ XOR } i1 \text{ XOR } i2 \text{ XOR } i3 \quad (4)$$

$$S2 = r2 \text{ XOR } i2 \text{ XOR } i3 \text{ XOR } i4 \quad (5)$$

$$S3 = r3 \text{ XOR } i1 \text{ XOR } i2 \text{ XOR } i4 \quad (6)$$

where S1-3 – error syndromes, r1,r2,r3 – parity bits, i1,i2,i3,i4 – data bits.

```
In [39]: syndromes = []
         for i in range(len(blocks_6)):
             s_1 = int(blocks_6[i][0]) ^ int(blocks_6[i][1]) ^ int(blocks_6[i][2]) ^ int(blocks_6[i][4])
             s_2 = int(blocks_6[i][1]) ^ int(blocks_6[i][2]) ^ int(blocks_6[i][3]) ^ int(blocks_6[i][5])
             s_3 = int(blocks_6[i][0]) ^ int(blocks_6[i][1]) ^ int(blocks_6[i][3]) ^ int(blocks_6[i][6])
             s = (str(s_1)+str(s_2)+str(s_3))
             syndromes.append(s)

In [40]: syndromes

Out[40]: ['100',
          '011',
          '110',
          '011',
          '001',
          '010',
          '001',
          '010',
          '110',
          '111',
          '001',
          '010',
          '100',
          '001',
          '101',
          '001',
          '111',
          '110',
          '101',
```

Figure – 13 – Calculating error syndromes.

```
In [41]: vals = [6, 5, 3, 4, 0, 2, 1]
         keys = ['001', '010', '011', '100', '101', '110', '111']
         check = dict(zip(keys, vals))

In [42]: check

Out[42]: {'001': 6, '010': 5, '011': 3, '100': 4, '101': 0, '110': 2, '111': 1}

In [43]: for i in range(len(blocks_6)):
             part_6 = list(blocks_6[i])
             if part_6[check[syndromes[i]]] == '0':
                 part_6[check[syndromes[i]]] = '1'
             else:
                 part_6[check[syndromes[i]]] = '0'
             blocks_6[i] = ''.join(part_6)

In [44]: for i in range(len(blocks_6)):
             blocks_6[i] = blocks_6[i][:4]

In [45]: blocks_6

Out[45]: ['1010',
          '0101',
          '1000',
          '1110',
          '0110',
          '1111',
          '1011',
          '1111',
          '1110',
          '1111',
          '0110',
          '0110',
          '1111',
          '0100',
          '1111',
          '1110',
          '1111',
          '0000',
          '1110',
```

Figure – 14 – Finding errors.

Figure – 15 – Binary representation of text without errors.

Figure – 16 – Decoded text.

CONCLUSION

During this project we gained the knowledge of encoding algorithms practically. We dealt with Huffman and Hamming encodings and realized them in Python language.

We faced several troubles on 2nd part where we were asked to write an algorithm of Huffman encoding. However, after some research we developed our own and maybe state-of-art algorithm without using of Trees and Nodes. The solution for other parts came up really fast in one breath.

Project also revealed some forgotten functions and features of programming language. Moreover, it required real participation and interaction with the course material, which made it more interesting. Researching, critical thinking and team work were the main three objectives of a project for us. These skills worked and fit our personalities the best and we came up with solution without any difficulty.

REFERENCES

- «Information Theory» course, L. Kozina, N. Duzbayev, International Information Technologies University
- Information Measures: Information and its Description in Science and Engineering, **Arndt C.**
- Elements Of Information Theory, **Thomas Cover.**

APPENDIX

This Project is done by

Yersain Makazhanov

Arsen Yerbol

PART 1

```
import numpy as np

import random


f = open("text.txt", "r")


text = f.read()


keys = list(set(text))


values = np.zeros((len(keys),), dtype = int)


d = dict(zip(keys, values))


for char in text:

    d[char]+=1


for key in d:

    d[key] = d[key] / len(text)
```

```
d = sorted(d.items(), key=lambda x: x[1], reverse = True)
```

PART 2

```
d2 = dict(d.copy())


i = len(d) – 2


for x in d2.keys():
    d2[x] = ''


while i!=-1:
    first = d[i][0]+d[i+1][0]
    second = d[i][1]+d[i+1][1]
    a = (first, second)
    d.append(a)
    if d[i][1]>d[i+1][1]:
        for key, value in d2.items():
            if key in d[i][0]:
                d2[key] += '1'
            elif key in d[i+1][0]:
                d2[key] += '0'
    else:
        for key, value in d2.items():
            if key in d[i][0]:
```

```
            d2[key] += '0'
        elif key in d[i+1][0]:
            d2[key] += '1'
    d = dict(d)
    d = sorted(d.items(), key=lambda x: x[1], reverse = True)
    del d[-1]
    del d[-1]
    i-=1
print(d)


for key in d2.keys():
    d2[key] = d2[key][::-1]


output = ''
for word in text:
    for char in word:
        output+=d2[char]
```

PART 3

```
char = ''
sentence = ''
for i in output:
    char+=i
    if char in d2.values():
        sentence+=list(d2.keys())[list(d2.values()).index(char)]
```

```
        char = ''
```

PART 4

```
blocks_4 = []
b_4 = str()
for i in range(len(output)):
    b_4 += output[i]
    if (i+1)%4==0:
        blocks_4.append(b_4)
        b_4 = str()

left_4 = output[len(blocks_4)*4:]

trios = []
for i in range(len(blocks_4)):
    t_1 = int(blocks_4[i][0]) ^ int(blocks_4[i][1]) ^ int(blocks_4[i][2])
    t_2 = int(blocks_4[i][1]) ^ int(blocks_4[i][2]) ^ int(blocks_4[i][3])
    t_3 = int(blocks_4[i][0]) ^ int(blocks_4[i][1]) ^ int(blocks_4[i][3])
    t = (str(t_1)+str(t_2)+str(t_3))
    blocks_4[i]+=t

output_4 = str()
for i in blocks_4:
    output_4+=i
output_4+=left_4
```

PART 5

```
blocks_5 = []

b_5 = str()

for i in range(len(output_4)):

    b_5 += output_4[i]

    if (i+1)%7==0:

        blocks_5.append(b_5)

        b_5 = str()

left_5 = output_4[len(blocks_5)*7:]


for i in range(len(blocks_5)):

    index = random.randint(0, 6)

    part = list(blocks_5[i])

    if part[index]=='0':

        part[index] = '1'

    else:

        part[index] = '0'

    blocks_5[i] = ''.join(part)


output_5 = str()

for i in blocks_5:

    output_5+=i

output_5+=left_5
```

PART 6

```
blocks_6 = []
b_6 = str()
for i in range(len(output_5)):
    b_6 += output_5[i]
    if (i+1)%7==0:
        blocks_6.append(b_6)
        b_6 = str()
left_6 = output_5[len(blocks_6)*7:]


syndromes = []
for i in range(len(blocks_6)):
    s_1 = int(blocks_6[i][0]) ^ int(blocks_6[i][1]) ^ int(blocks_6[i][2]) ^ int(blocks_6[i][4])
    s_2 = int(blocks_6[i][1]) ^ int(blocks_6[i][2]) ^ int(blocks_6[i][3]) ^ int(blocks_6[i][5])
    s_3 = int(blocks_6[i][0]) ^ int(blocks_6[i][1]) ^ int(blocks_6[i][3]) ^ int(blocks_6[i][6])
    s = (str(s_1)+str(s_2)+str(s_3))
    syndromes.append(s)


vals = [6, 5, 3, 4, 0, 2, 1]
keys = ['001', '010', '011', '100', '101', '110', '111']
check = dict(zip(keys, vals))
```

```
for i in range(len(blocks_6)):
    part_6 = list(blocks_6[i])
    if part_6[check[syndromes[i]]] == '0':
        part_6[check[syndromes[i]]] = '1'
    else:
        part_6[check[syndromes[i]]] = '0'
    blocks_6[i] = ''.join(part_6)

for i in range(len(blocks_6)):
    blocks_6[i] = blocks_6[i][:4]

output_6 = str()
for i in blocks_6:
    output_6 += i
output_6+=left_6

char = ''
sentence = ''
for i in output_6:
    char+=i
    if char in d2.values():
        sentence+=list(d2.keys())[list(d2.values()).index(char)]
        char = ''
```