Compression Algorithm

Edoardo Takanen

1 Genetic Code

Inside the DNA, a string full of nucleotides has all the information to build all the proteins needed by the system.
A sequence is made of nucleotides triplets, also called codons. Based on where we start reading this string, we get different codons.
For example, the string AAATGAACG, if read from the first position, contains the codons AAA, TGA, and ACG; if read from the second position, it contains the codons AAT and GAA; and if read from the third position, it contains the codons ATG and AAC.
This means that the same compressed sequence, can have different meanings based on the index of start. This got me thinking about a new way of compressing data, but also of encrypting.

2 The Structure

The program is divided in three parts:

1. The encrypter
2. The indexes-passcode creator
3. The decrypter

The first two will generate two separated files that will be read by the decrypter in the end.
I decided to separate the short sequence from the indexes, so that it can also work as an encryption system, since the short sequence is unreadable and meaningless without the indexes.

3 The Encrypter

The encrypter will generate the shortest sequence of bytes that contains all the characters used in the original file. If we take all the bits from some text, the result will be pretty much as long as the input. That is why I thought about reading these bits with another representation, like octal or hexadecimal, let me do an example:
We have a string:

"Hi how are you?"

This is the string in bits:

01001000 01101001 00100000 01101000 01101111 01110111 00100000 01100001 01110010 01100101 00100000 01111001 01101111 01110101 00111111

Since an octal digit represents 3 bits, and 8 is not divisible by 3, we just join these bits and partition it in pieces of 3 bits:

010 010 000 110 100 100 100 000 011 010 000 110 111 101 110 111 001 000 000 110 000 101 110 010 011 001 010 010 000 001 111 001 011 011 110 111 010 100 111 111

This in octal will be:

2206444032067567100605623122017133672477

While if we wanted to read it with hexadecimal representation, that would be easier, since an hex digit represents 4 bits, resulting in:

486920686f772061726520796f753f

You can clearly see more repeating characters in the last two sequences, this means we can shorten this sequence.
After writing the algorithm, I actually realized that using hexadecimal, the output is shorter than using octal.

4 The indexes-passcode Creator

This part is for creating the file where all the indexes are stored. I had to find a way to store these efficiently, we cannot have 8 bits for each index, that would be the same as storing each character individually. So I had to find a way to use less than 8 bits for each index.

4.1 Shortcuts method

The first method I came up with is the "Shortcuts" way.
Let's say we have these indexes:

```
1 indexes = [8, 5, 2, 4, 3, 4, 2, 1, 3]
```

The shortcuts will be a list of unique indexes

```
unique = list(set(indexes))
# [8, 5, 2, 4, 3, 1]
```

So the new indices will be

```
indices = []
for i in indexes:
    indices.append(no_duplicates.index(i))

# [0, 1, 2, 3, 4, 3, 2, 5, 4]
```

Why doing this? In case of big strings, we have a lot of large integers, since we have to give each index the same amount of bits, this will result in so much bits used. With this method, we can even give 9 bits for each shortcut, representing the real index, and then use just 6 bits for the new indices. The final file will look like this:

```
# represeting a binary file

# FIRST PART OF THE FILE: legend
# 4 bits for each integer
    0100 # indicates 4 bits for shortcuts
    0011 # indicates 3 bits for indices

# DELIMITER

# SECOND PART: shortcuts
    1000 # 8
    0101 # 5
    0010 # 2
    0100 # 4
    0011 # 3
    0001 # 1

# DELIMITER

# THIRD PART: indices
    000 # 0 -> 8
    001 # 1 -> 5
    010 # 2 -> 2
    011 # 3 -> 4
    100 # 4 -> 3
    011 # 3 -> 4
    010 # 2 -> 2
    101 # 5 -> 1
    100 # 4 -> 3
```

The delimiter is just a sequence that is needed to split the file in 3 parts for the decryption, in this case it is just **0x0F0F0F0F0F0F0F0F**

4.2 Huffman method

One day I was searching for good compression algorithms, and I came across this one. With this method we can have a variable-length code table for encoding the characters.

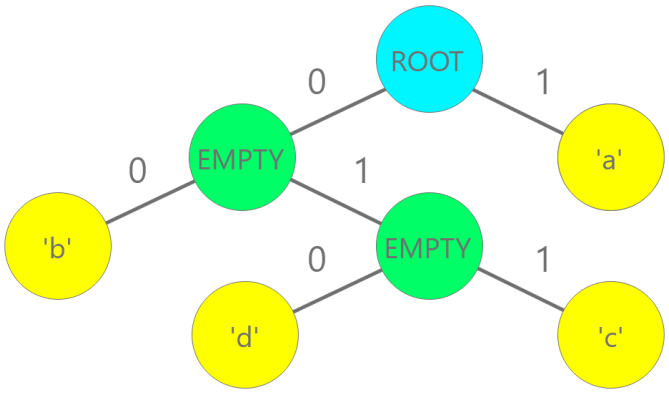

Figure 1: Huffman table example

The technique works by creating a binary tree of nodes. A node can be a character or an empty node with children. After creating the table, each character will have its own variable-length code, which is determined by its position. Every time we look at the right child, we add 1, and every time we look at the left child, we add a 0.
If we look at the example tree in 1 we end up with these codes:

Char	code (bits)
'a'	1
'b'	00
'c'	011
'd'	010

Usually, the most used character are the first children in the tree, so that they have the shortest codes.

The last thing I had to figure out was how I could write the table in the file. I have to save a dictionary with all the frequencies of the indexes:

```
freqs = Counter(indexes)

# frequencies example
# {
#     1: 5 | index 1 occurs 5 times
#     2: 2
#     4: 6
#     7: 5
# }
```

You can notice the frequencies of each index are not progressive, for example we jump from frequency 2 to frequency 5.
For optimization, I decided to "scale" this dictionary, and just keep the relative order, as in a ranking.

```
freqs = scale_dict(freqs)

# "scaling" the dictionary
# {
#     1: 2
#     2: 1
#     4: 3
#     7: 2
# }
```

In such a way that I can write the table like this

```
# 8 bits for index and 1 bit for relative frequency
00000010 0 # frequency = 1
00000001 1 # frequency = 2; occurs one more time than the previous
00000111 0 # frequency = 2; occurs the same times as the previous
00000100 1 # frequency = 3; occurs one more time than the previous
```

The final file will look like this:

```
# represeting a binary file

# FIRST PART OF THE FILE: legend
# 1 byte that indicates the number of added bits at the end of the
    file
    00000000

# DELIMITER

# SECOND PART: table
# 9 bits each
# 8 bits for the index, 1 is the relative frequency
    00000010 0 # 2: 1
    00000001 1 # 1: 2
    00000111 0 # 7: 2
    00000100 1 # 4: 3

# DELIMITER

# THIRD PART: indices
```

4.3 Comparing the two methods

File	Bytes	Bits	KB
input	11,755	94,040	11.76
encrypted	35	280	0.04
indexes	8,872	70,976	8.87
total	8,907	71,256	8.91
efficiency	2,848	22,784	2.85
size shortcuts	7 bits		
size indexes	6 bits		

File	Bytes	Bits	KB
input	11,755	94,040	11.76
encrypted	35	280	0.04
indexes	6,920	55,360	6.92
total	6,955	55,640	6.96
efficiency	4,800	38,400	4.80
size indexes	6847 bytes		
size table	49 bytes		

I tested these two methods with the same text file, sized 11.76 KB.
I can clearly say that the huffman method is way more efficient than the shortcuts one.