Coding algorithms

Enrico Maria Di Mauro
matr. 062270**1706**
Allegra Cuzzocrea
matr. 062270**1707**

Acronyms

RV → Random Variable

PMF → Probability Mass Function

Notation

X is a RV

x are values assumed by X

$p(\cdot)$ is the PMF of a RV

$p(\cdot, \cdot)$ is the joint PMF of a RV

SUMMARY pt.1

SUMMARY pt.2

Lempel-Ziv coding

Lempel-Ziv encoding example

Lempel-Ziv decoding example

Lempel-Ziv coding considerations

Implementation structure

Examples

Huffman Vs Lempel-Ziv

Source code

A source code for a discrete RV X is a mapping from the range of X to the set of finite-length binary strings

C: source code

$C(x)$: codeword corresponding to x

$l(x)$: length of $C(x)$

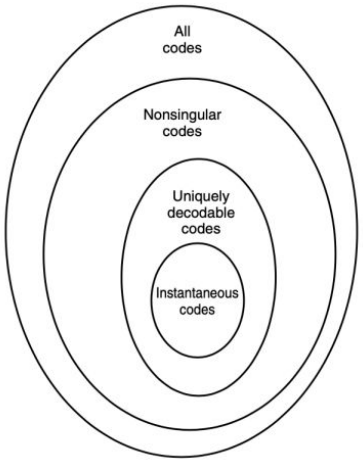
$L(C)$: expected length of C (statistical average of RV $l(X)$)

C^*: extension of C (concatenation of codewords)

Source codes taxonomy

A code is called:

- **Nonsingular** if $x \neq x' \implies C(x) \neq C(x')$
- **Uniquely decodable** if its extension C* is nonsingular
- **Prefix** or **instantaneous** if no codeword is the prefix of any other codeword



Shannon source coding theorem

Let X be a discrete source with PMF $p(x)$ and let L_{min} be the minimum expected length over all instantaneous codes for X. Then

$$H(X) \leq L_{min} < H(X) + 1$$

In addition:

- $L_{min} \geq H(\mathrm{X})$ is valid also for any uniquely decodable code for X
- $L_{min} = H(\mathrm{X}) \Leftrightarrow p(x)$ is an integer power of 2 for all x

If one considers the n-th extension of the source, the above theorem gives the following result with L_{min} be the per-letter minimum expected length:

$$H(X) \leq L_{min} < H(X) + \frac{1}{n}$$

Huffman coding

Huffman Coding is a lossless technique of compressing data

Given a set of symbols, Huffman **encoding algorithm** is done with the following steps:

1. Calculate the probability (or frequency) of each symbol
2. Sort the probabilities
3. Sum the two smallest probabilities
4. Repeat steps 2. and 3. until the number of remaining probabilities is one
5. Assign 0 and 1 values to the left (right) and right (left) links respectively of the obtained tree

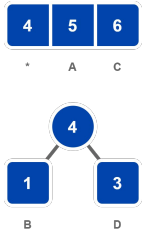
Given a code and the corresponding Huffman tree, Huffman **decoding algorithm** is done by traversing the tree using the code bits to find the symbols

Huffman encoding example

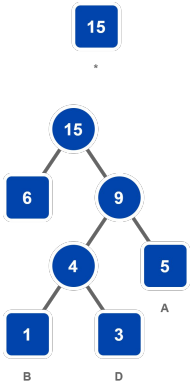
1.

2.

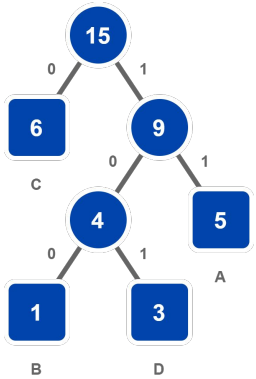
3.

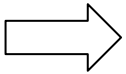


4.



5.

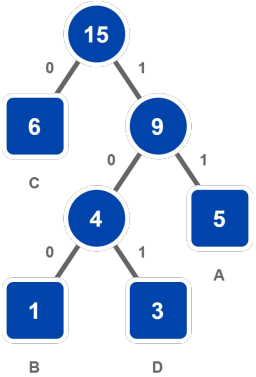


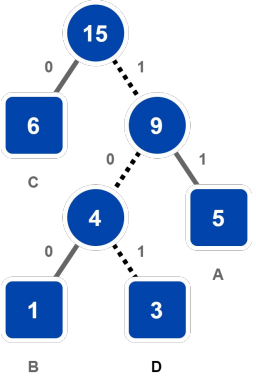


Symbol	Codeword
A	11
B	100
C	0
D	101

Huffman decoding example

Code: 101





Source: D

Huffman coding considerations

Huffman code is not unique for some reasons:

- More symbols can have the same probability
- It is possible to invert the links assigned values
- It is possible to assign values in different ways for each couple of links

All Huffman codes have the same minimum expected length

The longest codewords are associated with the least probable symbols and vice versa

Data structures

Heap

Used to obtain nodes with minimum frequencies

This data structure guarantees $O(\log(n))$ complexity for push and pop operations

Binary tree

Used as Huffman tree in order to encode and decode symbols

Each node has a left and right child and links are associated to 0 and 1 values

The complexity of the encoding algorithm implementation is $O(n\log(n))$

Implementation structure

Two main functions

```
def huffmanEncode(data):
    """Method that encodes data

    Args:
        data: data containing all symbols

    Returns:
        outputEnc: output encoded string
        nodes[0]: Huffman tree root
        symCode: symbols and Huffman codes dictionary
    """
```

- Create the Huffman tree
- Calculate the codewords for symbols
- Encode the data

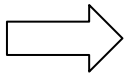
```
def huffmanDecode(outputEnc, huffmanTree):
    """Method that decodes data

    Args:
        outputEnc: output encoded string
        huffmanTree: Huffman tree root

    Returns:
        outputDec: output decoded string
    """
```

- Read the code bit by bit
- Traverse the tree
- Decode to obtain data again

Examples

qwerty.txt (32 characters)

qwerty.txt
qwertyuiopasdfghjklzxcvbnm123456

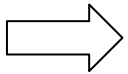


```
Space usage before compression: 256 bits (32 Bytes)
Space usage after compression: 160 bits (20.0 Bytes)
Compression gain: 96 bits (12.0 Bytes)
Compression percentage: 37.5 %
Symbols frequencies: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Entropy: 5.0
Codes mean length: 5.0
Difference: 0.0
```

long.txt (762013 characters)

long.txt
Data compression is a reduction in the number

How compression works



```
Space usage before compression: 6096104 bits (762013 Bytes)
Space usage after compression: 3232693 bits (404086.625 Bytes)
Compression gain: 2863411 bits (357926.375 Bytes)
Compression percentage: 46.97116388 %
Symbols frequencies: [606, 24262, 35988, 71551, 29485, 111991, 64880, 14722, 54917, 54704, 18779, 33885, 54155, 51759, 2, 25297, 37601, 9
388, 6938, 9062, 15056, 1522, 1061, 4402, 10368, 978, 3450, 661, 1093, 1609, 1300, 671, 665, 252, 323, 1272, 896, 1996, 1996, 643, 836, 3
16, 284, 136, 76, 116, 63]
Entropy: 4.20897646
Codes mean length: 4.24230689
Difference: 0.03333043
```

Examples

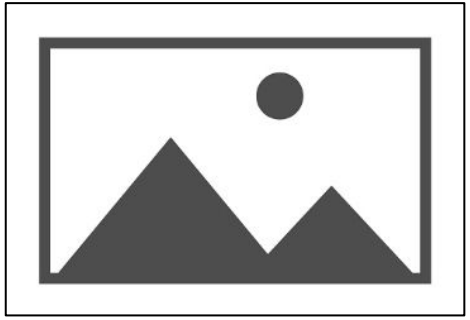
slide.txt (20 characters)

slide.txt

11111222223333444555

```
Space usage before compression: 160 bits (20 Bytes)
Space usage after compression: 46 bits (5.75 Bytes)
Compression gain: 114 bits (14.25 Bytes)
Compression percentage: 71.25 %
Symbols frequencies: [5, 5, 4, 3, 3]
Entropy: 2.2854753
Codes mean length: 2.3
Difference: 0.0145247
```

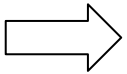
img.jpg



```
Space usage before compression: 147904 bits (18488 Bytes)
Space usage after compression: 138816 bits (17352.0 Bytes)
Compression gain: 9088 bits (1136.0 Bytes)
Compression percentage: 6.14452618 %
Symbols frequencies: [567, 47, 95, 1218, 45, 38, 44, 92, 108, 46, 121, 111, 282, 74, 131, 57, 31, 57, 42, 82, 28, 47, 43, 135, 152, 124,
89, 87, 157, 43, 127, 151, 94, 118, 57, 41, 91, 77, 99, 59, 97, 92, 145, 92, 61, 37, 68, 161, 57, 74, 80, 82, 32, 131, 77, 41, 33, 60, 54
, 44, 42, 36, 38, 96, 88, 51, 69, 39, 89, 34, 64, 70, 71, 67, 37, 91, 44, 52, 63, 139, 31, 73, 30, 119, 52, 27, 36, 38, 48, 43, 62, 47, 4
1, 47, 57, 59, 78, 23, 60, 66, 60, 39, 51, 40, 140, 287, 102, 41, 36, 42, 32, 73, 39, 28, 52, 192, 322, 58, 55, 36, 258, 47, 32, 342, 68,
 44, 42, 62, 59, 13, 65, 37, 55, 37, 61, 59, 254, 46, 57, 23, 31, 107, 28, 115, 35, 57, 47, 31, 60, 28, 43, 36, 30, 99, 37, 32, 38, 42, 7
6, 37, 34, 28, 30, 44, 39, 61, 73, 53, 48, 58, 44, 45, 62, 32, 26, 24, 33, 58, 26, 50, 59, 103, 62, 50, 41, 30, 26, 36, 47, 61, 345, 58,
41, 63, 57, 56, 53, 27, 49, 17, 38, 42, 32, 96, 44, 43, 35, 62, 31, 53, 33, 35, 32, 53, 59, 42, 35, 230, 35, 47, 44, 56, 68, 103, 43, 67,
 23, 51, 44, 50, 55, 44, 29, 30, 27, 47, 27, 47, 66, 27, 94, 129, 21, 28, 41, 27, 33, 184, 41, 36, 66, 35, 53, 55, 86, 48]
Entropy: 7.48435822
Codes mean length: 7.50843791
Difference: 0.02407969
```

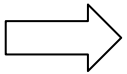
Examples

black.jpg



```
Space usage before compression: 4152720 bits (519090 Bytes)
Space usage after compression: 1172808 bits (146601.0 Bytes)
Compression gain: 2979912 bits (372489.0 Bytes)
Compression percentage: 71.75807663 %
Symbols frequencies: [13, 3, 2, 46, 3, 5, 4, 4, 88, 2, 1, 1, 5, 4, 4, 3, 3, 1, 3, 9, 5, 3, 1, 3, 1, 3, 4, 3, 1, 2, 129603, 4, 3, 4, 1, 4,
 1, 5, 3, 1, 3, 3, 1, 3, 3, 1, 129604, 2, 84, 1, 8, 5, 1, 13, 3, 6, 2, 8, 12, 4, 5, 4, 2, 4, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 129602, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 2, 2, 2, 2
, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 1, 3, 129600]
Entropy: 2.02014832
Codes mean length: 2.25935387
Difference: 0.23920555
```

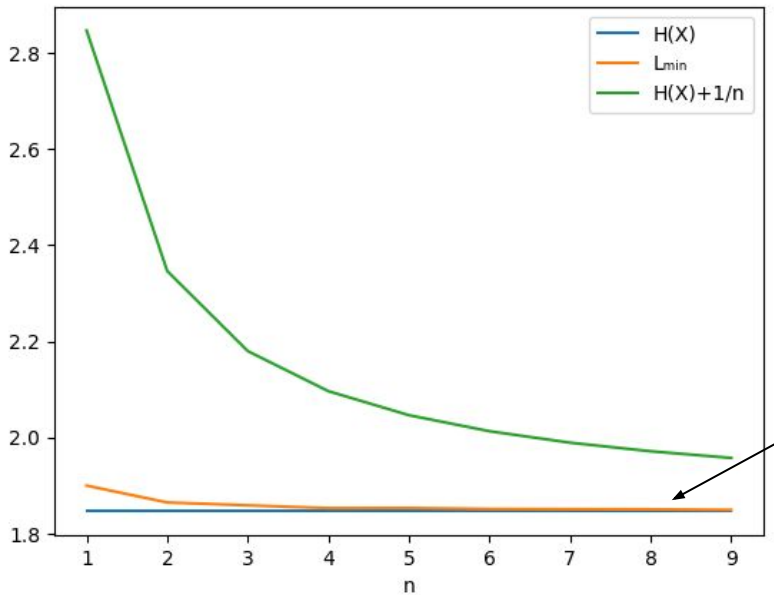
rainbow.png



```
Space usage before compression: 160864 bits (20108 Bytes)
Space usage after compression: 138253 bits (17281.625 Bytes)
Compression gain: 22611 bits (2826.375 Bytes)
Compression percentage: 14.05597275 %
Symbols frequencies: [34, 36, 67, 28, 64, 23, 51, 26, 28, 15, 16, 35, 41, 74, 10, 23, 44, 15, 31, 88, 33, 58, 11, 35, 60, 37, 62, 22, 62,
 36, 112, 29, 54, 64, 26, 71, 14, 40, 58, 40, 74, 60, 17, 67, 35, 53, 75, 70, 36, 45, 49, 82, 78, 83, 50, 75, 61, 66, 1088, 69, 78, 46, 3
5, 32, 63, 33, 17, 46, 1079, 20, 51, 62, 54, 22, 13, 46, 9, 70, 54, 44, 62, 70, 31, 40, 53, 71, 63, 55, 57, 65, 39, 58, 44, 43, 65, 57, 3
8, 54, 87, 39, 56, 54, 15, 50, 1096, 45, 49, 25, 55, 45, 91, 52, 30, 47, 61, 66, 57, 85, 49, 56, 61, 31, 45, 90, 60, 73, 31, 67, 44, 28,
45, 70, 94, 73, 54, 711, 69, 55, 76, 56, 92, 30, 37, 52, 48, 20, 51, 47, 66, 52, 9, 50, 122, 63, 48, 59, 63, 26, 64, 79, 43, 70, 25, 38,
29, 63, 12, 906, 53, 77, 58, 52, 40, 46, 24, 60, 61, 22, 30, 82, 923, 41, 58, 42, 84, 63, 56, 65, 17, 30, 62, 56, 90, 18, 76, 17, 1320, 4
4, 74, 56, 58, 31, 69, 65, 44, 25, 62, 62, 72, 42, 51, 36, 63, 91, 37, 27, 51, 37, 80, 41, 18, 64, 37, 62, 48, 46, 33, 35, 52, 40, 15, 49
, 62, 26, 34, 42, 62, 961, 24, 21, 50, 30, 26, 27, 51, 51, 22, 27, 20, 16, 36, 20, 68, 6, 24, 10]
Entropy: 6.84285562
Codes mean length: 6.87552218
Difference: 0.03266656
```

Benchmarks and considerations

PMF = {'A':0.3, 'B':0.2, 'C':0.4, 'D': 0.1}



Assumption: the outputs of the source are independent

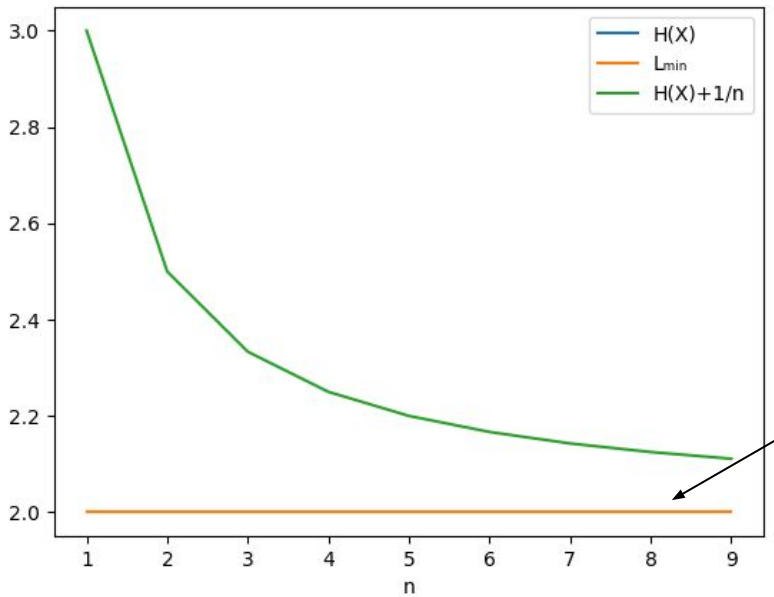
$$p(x_1, x_2, \ldots, x_n) = p(x_1)p(x_2)\ldots p(x_3)$$

As n increases L_{min} converges to entropy

$$H(X) \leq L_{min} < H(X) + \frac{1}{n}$$

Benchmarks and considerations

PMF = {'A':0.25, 'B':0.25, 'C':0.25, 'D': 0.25}



Assumption: the outputs of the source are independent

$$p(x_1, x_2, \ldots, x_n) = p(x_1)p(x_2)\ldots p(x_3)$$

Per-letter minimum expected length coincides with entropy since PMF is uniform

$$H(X) \leq L_{min} < H(X) + \frac{1}{n}$$

Benchmarks and considerations

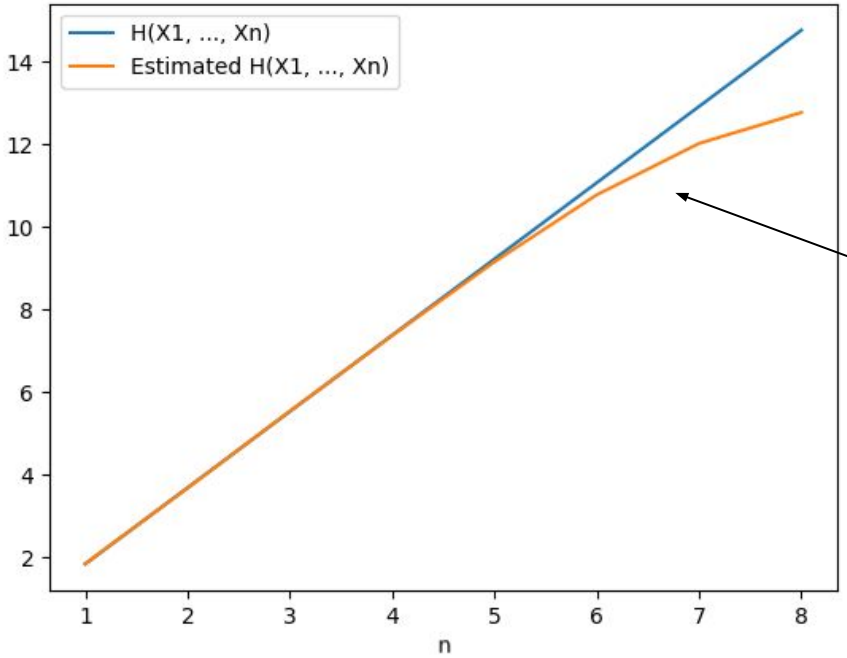
PMF = {'A':0.25, 'B':0.25, 'C':0.25, 'D': 0.25}

```
Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 200000 bits (25000.0 Bytes)
Compression gain: 600000 bits (75000.0 Bytes)
Compression ratio: 75.0 %
Entropy: 2.0
Codes mean length: 2.0
Difference: 0.0
```

L_{min}=$H(X)$ since PMF is uniform

Benchmarks and considerations

$H(X_1, \ldots, X_n)$ Vs Estimated $H(X_1, \ldots, X_n)$



For n>5 estimated entropy moves further and further away from the entropy

Benchmarks and considerations

- data.txt has been generated using PMF_1
- Huffman algorithm has been tested with the following inputs:
  - data.txt and **PMF_1**
  - data.txt and **PMF_2**

PMF_1 = {'A':0.3, 'B':0.3, 'C':0.4, 'D': 0}

```
Encoding and Decoding with PMF₁

Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 190262 bits (23782.75 Bytes)
Compression gain: 609738 bits (76217.25 Bytes)
Compression ratio: 76.21725 %
Entropy: 1.570950594454669
Codes mean length: 1.9
Difference: 0.32904941
```

PMF_2 = {'A':0.25, 'B':0.25, 'C':0.2, 'D': 0.3}

```
Encoding and Decoding with PMF₂
D(PMF₁||PMF₂): 0.5578206435002763

Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 200000 bits (25000.0 Bytes)
Compression gain: 600000 bits (75000.0 Bytes)
Compression ratio: 75.0 %
Entropy: 1.9854752972273344
Codes mean length: 2.0
Difference: 0.0145247
```

Since divergence of PMF_1 from PMF_2 is high, Huffman algorithm has worse performances with PMF_2 although it nearly reaches the optimal value for L_{min}. It means that it is the best it can do

Lempel-Ziv coding

Lempel-Ziv Coding is a lossless technique of compressing data

- **LZ77**: the first algorithm was published in 1977 but it was thought to be too complex to implement
- **LZ78**: another version of the algorithm was published in 1978 and this time it had success
- **LZW**: in 1984 Terry Welch published an improved version of LZ78

Lempel-Ziv coding

Given a binary sequence, Lempel-Ziv **encoding algorithm** is done with the following steps:

1. Parse the source data stream into segments that are the shortest subsequences not encountered previously
2. Encode each segment concatenating its prefix address and its extra bit

Since codewords have a fixed length and the decoder is informed of it, Lempel-Ziv **decoding algorithm** decodes each codeword by concatenating its corresponding root subsequence and its last bit

Lempel-Ziv encoding example

Fixed length

1.

1, 0, 10, 01, 011, 11, 101, 010, 1010, 0110

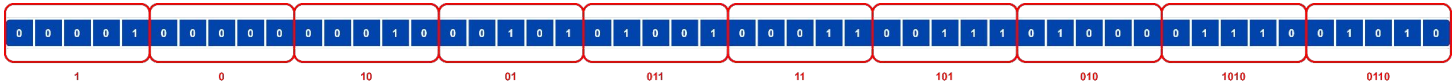
1 2 3 4 5 6 7 8 9 10

10 phrases of different lengths

2.

Address	Phrase	Codeword
1	1	0000 1
2	0	0000 0
3	10	0001 0
4	01	0010 1
5	011	0100 1
6	11	0001 1
7	101	0011 1
8	010	0100 0
9	1010	0111 0
10	0110	0101 0

Lempel-Ziv decoding example



The decoder knows the length of each codeword and it therefore knows that each phrase is encoded by 4 bits of address and the extra bit

Codeword No.	Codeword	Decoded phrase
1	0000 1	1
2	0000 0	0
3	0001 0	10
4	0010 1	01
5	0100 1	011
6	0001 1	11
7	0011 1	101
8	0100 0	010
9	0111 0	1010
10	0101 0	0110

Lempel-Ziv one reading coding

Given a binary sequence, Lempel-Ziv **one reading encoding algorithm** is done with the following step:

1. Parse the source data stream into segments that are the shortest subsequences not encountered previously. As soon as a segment is identified encode it using k+2 bits and use this length for next segments 2^k times

   k is an integer starting from '0'

   In particular:
   - encode the first segment with its only bit
   - encode each next segment concatenating its prefix address using k+1 bits and its extra bit

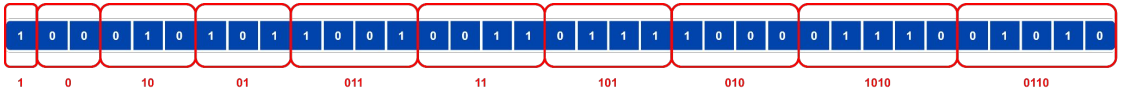
Since each codeword has its fixed length and the decoder knows it, Lempel-Ziv **one reading decoding algorithm** decodes each codeword by concatenating its corresponding root subsequence and its last bit

Lempel-Ziv one reading encoding example

1.

Address	Phrase		Codeword
1	1	⇨	1
2	0	⇨	0 0
3	10	⇨	01 0
4	01	⇨	10 1
5	011	⇨	100 1
6	11	⇨	001 1
7	101	⇨	011 1
8	010	⇨	100 0
9	1010	⇨	0111 0
10	0110	⇨	0101 0

Lempel-Ziv one reading decoding example



The decoder knows the length of each codeword and it therefore knows that each phrase is encoded by k+1 bits of address and the extra bit

Codeword No.	Codeword	Decoded phrase
1	1	1
2	0 0	0
3	01 0	10
4	10 1	01
5	100 1	011
6	001 1	11
7	011 1	101
8	100 0	010
9	0111 0	1010
10	0101 0	0110

Lempel-Ziv coding considerations

LZ algorithm is:

- **universal**: it does not need the statistical structure of the source
- **asymptotically optimal**: if the source is stationary and ergodic and its size is sufficiently big it has the same performances of an algorithm that knows the structure of the source

LZ algorithm also works when the source has gradually changing statistics

To encode, two readings of the entire sequence are needed but this is avoided in more efficient variations of the algorithm

Implementation structure

LZ algorithm has been implemented through **one reading** of the source

```
def lzEncode(data):
```

As soon as it identifies a new phrase, it:

- memorizes the phrase address in a dictionary
- encodes the phrase
- adds the encoded phrase to the encoded output string

In the end, the remaining bits are added to the encoded output string without being encoded

```
def lzDecode(outputEnc, lenEnd):
```

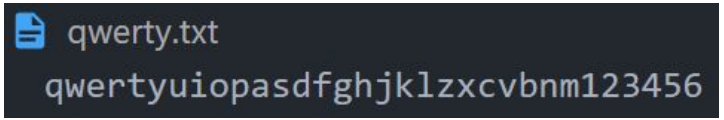
For each codeword read, it:

- decodes the codeword
- adds the decoded codeword to the decoded output string
- memorizes the decoding obtained in a dictionary

When the codewords are finished it adds the last bits to the decoded output string

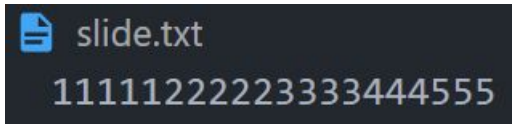
Examples

qwerty.txt (32 characters)




```
Encoding time: 0.0
256
Encoded input: 347
Decoding time: 0.0
Original input: 256
Space usage before compression: 256 bits (32 Bytes)
Space usage after compression: 347 bits (43 Bytes)
Compression gain: -91 bits (-11 Bytes)
Compression ratio: -35.546875 %
```

slide.txt (20 characters)

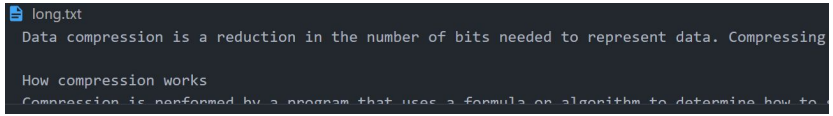


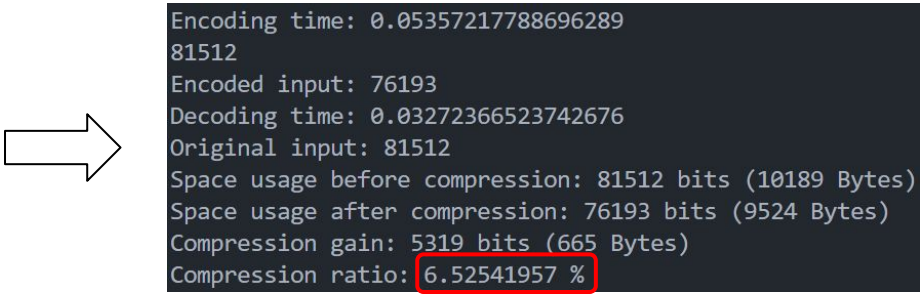

```
Encoding time: 0.0
160
Encoded input: 199
Decoding time: 0.0
Original input: 160
Space usage before compression: 160 bits (20 Bytes)
Space usage after compression: 199 bits (24 Bytes)
Compression gain: -39 bits (-4 Bytes)
Compression ratio: -24.375 %
```

For short files the algorithm has bad performances in terms of **compression ratio**.
In fact, in the two showed cases it expands the source instead of compressing it

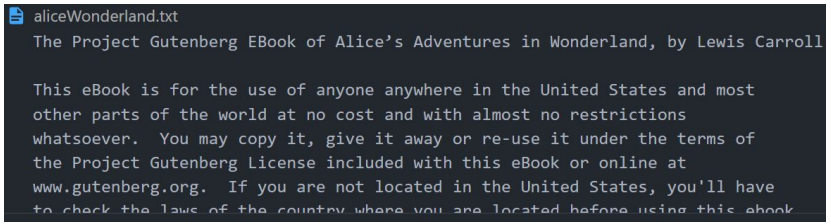
Examples

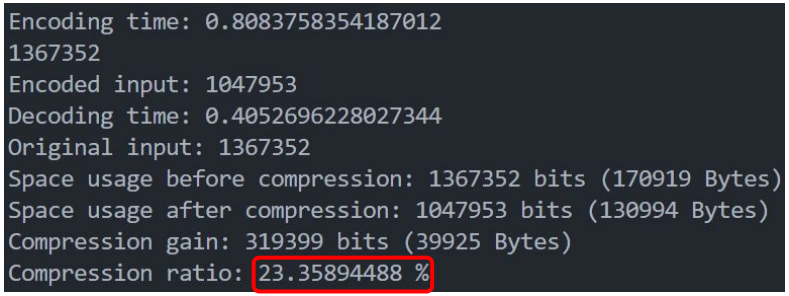
long.txt (10189 characters)





aliceWonderland.txt (164409 characters)

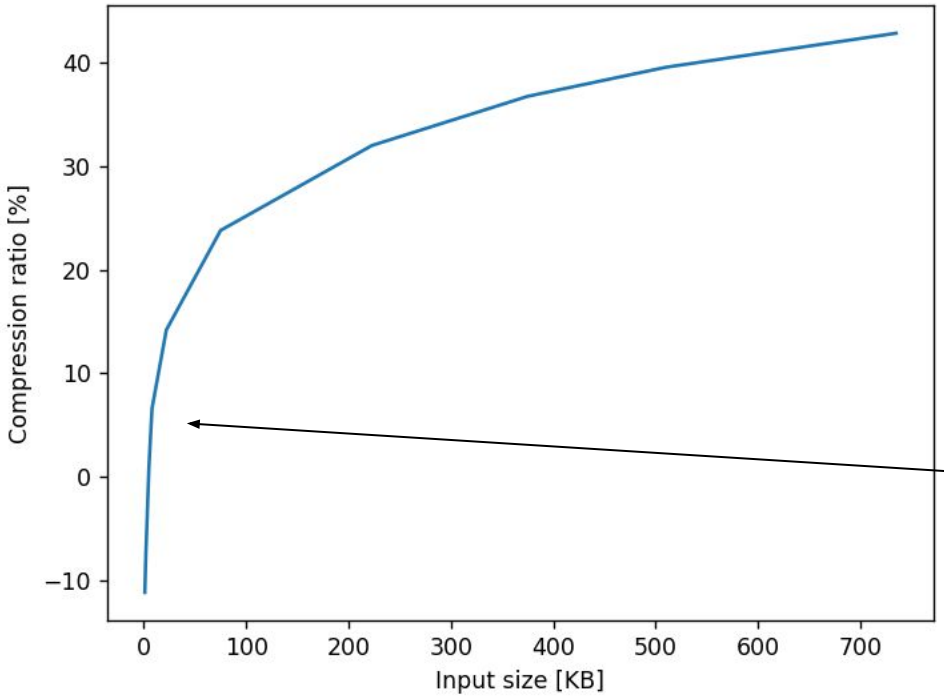




Performances in terms of **compression ratio** start to increase

Examples

Compression ratio as the input size of a random text file increases



It can be noted that **compression ratio** increases as the number of characters increases

For small input sizes the increase is high

Examples

lorem700.txt (521852 characters)

```
Encoding time: 2.6303327083587646
4174816
2524392
Encoded input: 2524392
Space usage before compression: 4174816 bits (521852 Bytes)
Space usage after compression: 2524392 bits (315549 Bytes)
Compression gain: 1650424 bits (206303 Bytes)
Compression ratio: 39.53285606 %
```

a.txt (521852 characters)

```
Encoding time: 2.5824685096740723
4174816
106990
Encoded input: 106990
Space usage before compression: 4174816 bits (521852 Bytes)
Space usage after compression: 106990 bits (13373 Bytes)
Compression gain: 4067826 bits (508479 Bytes)
Compression ratio: 97.43725232 %
```

Lempel-Ziv algorithm performs a compression ratio of 39% on a random text file, while it performs 97% on a deterministic text file of the same size

Examples

PMF = {'A':0.25, 'B':0.2, 'C':0.1, 'D': 0.09, 'E': 0.2, 'F': 0.16}

new.txt (520000 characters)

```
Encoding time: 3.0747828483581543
Encoded input: 2109750
Space usage before compression: 4160000 bits (520000 Bytes)
Space usage after compression: 2109750 bits (263718 Bytes)
Compression gain: 2050250 bits (256282 Bytes)
Compression ratio: 49.28485577 %
```

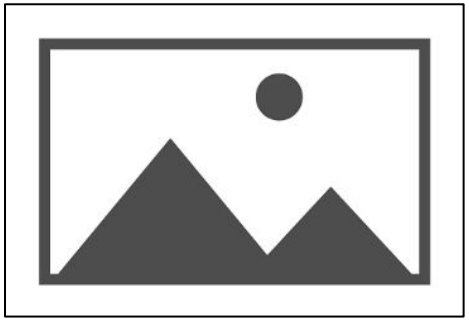
a.txt (520000 characters)

```
Encoding time: 3.488454818725586
Encoded input: 106267
Space usage before compression: 4160000 bits (520000 Bytes)
Space usage after compression: 106267 bits (13283 Bytes)
Compression gain: 4053733 bits (506717 Bytes)
Compression ratio: 97.44550481 %
```

Lempel-Ziv algorithm performs a compression ratio of about 49% on a random text file, while it performs about 97% on a deterministic text file of the same size

Examples

img.jpg



```
Encoding time: 0.07953882217407227
147904
Encoded input: 151136
Decoding time: 0.05314040184020996
Original input: 147904
Space usage before compression: 147904 bits (18488 Bytes)
Space usage after compression: 151136 bits (18892 Bytes)
Compression gain: -3232 bits (-404 Bytes)
Compression ratio: -2.18520121 %
```

cat.jpg

```
Encoding time: 0.22141718864440918
279512
Encoded input: 317305
Decoding time: 0.1400465965270996
Original input: 279512
Space usage before compression: 279512 bits (34939 Bytes)
Space usage after compression: 317305 bits (39663 Bytes)
Compression gain: -37793 bits (-4724 Bytes)
Compression ratio: -13.52106529 %
```

Examples

black.jpg

```
Encoding time: 2.724907636642456
4152720
Encoded input: 231924
Decoding time: 1.0226917266845703
Original input: 4152720
Space usage before compression: 4152720 bits (519090 Bytes)
Space usage after compression: 231924 bits (28990 Bytes)
Compression gain: 3920796 bits (490100 Bytes)
Compression ratio: 94.41513032 %
```

rainbow.png

```
Encoding time: 0.09612751007080078
160864
Encoded input: 130757
Decoding time: 0.05297493934631348
Original input: 160864
Space usage before compression: 160864 bits (20108 Bytes)
Space usage after compression: 130757 bits (16344 Bytes)
Compression gain: 30107 bits (3764 Bytes)
Compression ratio: 18.71580963 %
```

Huffman Vs Lempel-Ziv

PMF = {'A':0.3, 'B':0.35, 'C':0.35, 'D': 0}

Huffman

```
Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 194731 bits (24341.375 Bytes)
Compression gain: 605269 bits (75658.625 Bytes)
Compression ratio: 75.658625 %
Entropy: 1.5812908992306927
Codes mean length: 1.95
Difference: 0.3687091
```

Lempel-Ziv

```
Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 316387 bits (39548 Bytes)
Compression gain: 483613 bits (60452 Bytes)
Compression ratio: 60.451625 %
```

PMF = {'A':0.25, 'B':0.25, 'C':0.25, 'D': 0.25}

Huffman

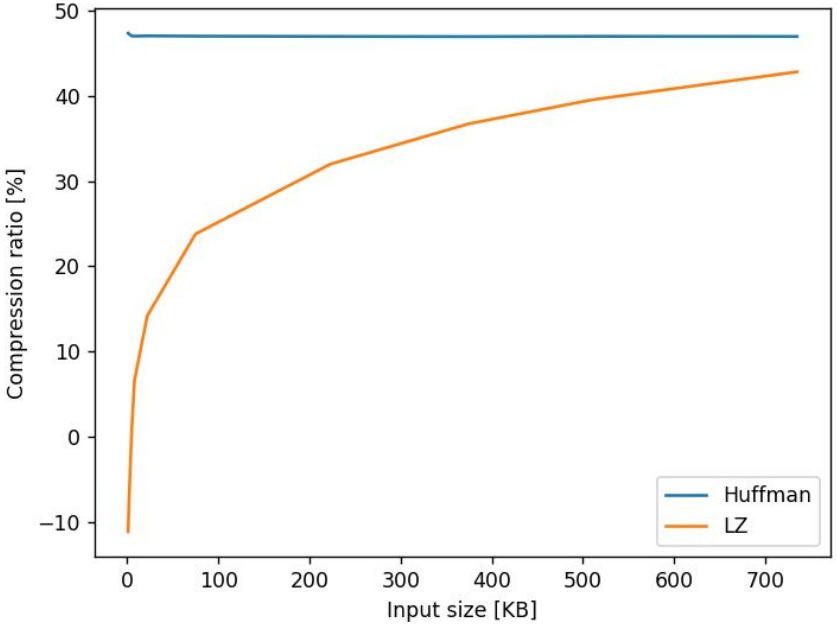
```
Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 200000 bits (25000.0 Bytes)
Compression gain: 600000 bits (75000.0 Bytes)
Compression ratio: 75.0 %
Entropy: 2.0
Codes mean length: 2.0
Difference: 0.0
```

Lempel-Ziv

```
Space usage before compression: 800000 bits (100000 Bytes)
Space usage after compression: 383892 bits (47986 Bytes)
Compression gain: 416108 bits (52014 Bytes)
Compression ratio: 52.0135 %
```

Huffman Vs Lempel-Ziv

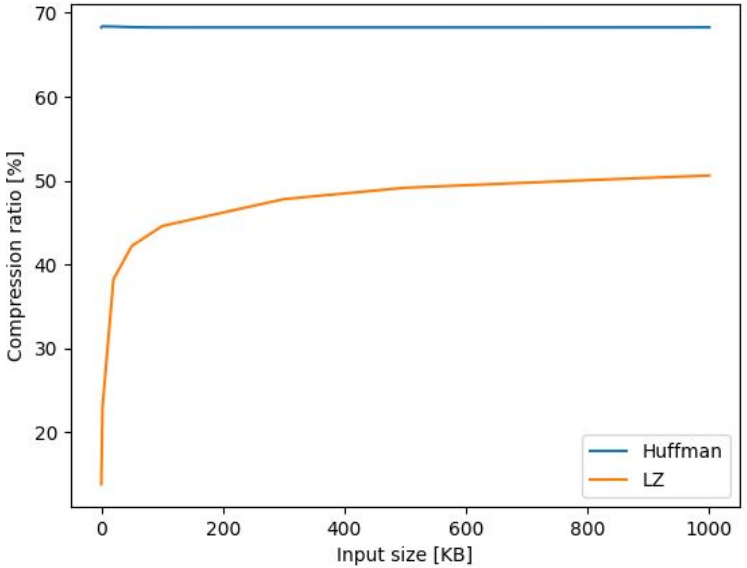
Huffman Vs **Lempel-Ziv** compression ratio as the number of characters of a random text file increases



- **Huffman** algorithm has good performances on small text files but it does not improve as the file dimension increases
- **Lempel-Ziv** algorithm has very bad performances on small text files and it improves as the file dimension increases

Huffman Vs Lempel-Ziv

Huffman Vs **Lempel-Ziv** compression ratio as the number of characters of a random text file increases



- **Huffman** algorithm has good performances on small text files but it does not improve as the file dimension increases
- **Lempel-Ziv** algorithm has very bad performances on small text files and it improves as the file dimension increases

Thank you for the attention

Enrico Maria Di Mauro
matr. 062270**1706**
Allegra Cuzzocrea
matr. 062270**1707**