Compression

The goal of compression to find a method which "given an input string" will produce a "compressed" string which:

- is shorter than the input string
- contains the same information. This means: from the compressed string one should be able to recover exactly the original input string

... (some information omitted)

The Limit to compression:
Now that we have an algorithm to construct an optimal code (Huffman), we ask is there a natural lower bound to compression. From experience we know that one cannot compress a string to infinitely small length. Using gzip we cannot compress a jpeg file because jpeg is already compressed. Also if we run gzip on the same file two times one after another, the second time gzip will not be able to compress the file further? So, on what depends if a file can be compressed? Is there a lower bound to compression and what is it?

Entropy: the lower bound to compression
Of course, because we proved that Huffman algorithm is optimal, it will provide us with a lower bound. However, it cannot give us a simple mathematical formula for the average number of bits required. Such a formula exists and is given by the so called entropy.
The entropy was introduced by Shannon as the only **measure of information in a probability distribution** that satisfies certain natural requirements. The entropy also provides a lower bound for compression. It is important to remember that

- **requires a probabilistic model** such as coin flipping (called the source) . The entropy is a mapping (function) from probability distributional to real numbers.
- the lower bound is **on average**, meaning the average expected code length expressed in number of bits per symbols is >= entropy of the source.

We now define entropy, but before we can speak of entropy we must have a source, such as a die, a coin, or a bag of words, together with the probabilities of possible outcomes.

We take the example of Bob's and Alice's language. We view this language as a source that emits symbols (which are A, B, C, D, E) (another way to say it: roll a die, draw a word according to the probability distribution).

Words	A	B	C	D	E
Prob	0.25	0.25	0.2	0.15	0.15

So, the entropy of that language is
H = Prob(A) * log(1 / Prob(A)) + Prob(B) * log(1 / Prob(B)) + ... + Prob(E) * log(1 / Prob(E))

Putting the numerical values we have
H = 0.25 * log(1 / 0.25) + 0.25 * log(1 / 0.25) + 0.2 * log(1 / 0.2) + 0.15 * log(1/0.15) + 0.15 * log(1/0.15)
We interpret this as: the number of bits on average to communicate a symbol from the language with five letters {A, B, C, D, E} is **....** Compare with number with the strategy where every symbol was represented by 3 bits (this is strategy 2 above).

We take the base of the log to be 2. This is important because it allows us to interpret the results as the lower bound of the **average number of bits per symbol required to transmit one symbol** from the language using the communication channel.

In general, if Bob's and Alice's language has n words, which are numbered as 1, 2, ..., n and
p(1) = probability of word 1,
p(2) = probability of word 2
...
p(n) = probability of word n

Then the entropy of the language (or the source is)
H(source) = p(1) * log(1/p(1)) + p(2) * log(1/p(2)) + ... + p(n) * log(1/p(n))
Notice that because log(1/ a) = log(a ^ (-1)) = - log(a), the formula for the entropy can be written as

H(source) = - p(1) * log(p(1)) – p(2) * log(p(2)) - .... - p(n) * log(p(n))

Interpretation of the entropy formula:
Interpretation 1: **log(1/p(i)) is the length of the code for word i in the optimal code. It can also be interpreted as the information contained in the symbol i.**

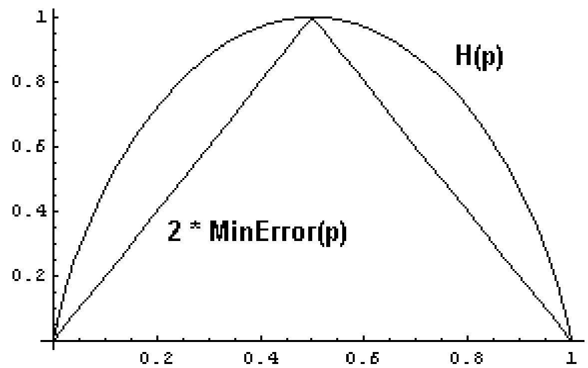
If we use length(1) = log(1/ p(1)) bits to encode the word numbered 1,
and we use length(2) = log(1/ p(2)) bits to encode the word numbered 2,
...
and we use length(n) = log(1/ p(n)) bits to encode the word numbered n,

(where length(i) = length of code word i)
then the on average to encode any symbol we will use

p(1) * length(1) + p(2) * length(2) + ... + p(n) * length(n) =
p(1) * log(1/p(1)) + p(2) * log(1/p(2)) + ... + p(n) * log(1/p(n)) = H(language)
bits to encode a symbol generated by the language.

A question that should be raised here is: is it possible to have codes whose code words will have exactly those lengths. The answer is yes. Because log(1/ p(i)) need not be an integer, but our code words must have integer lengths, we need to round up **ceil(log(1/ p(i)))** . A procedure that will construct a code with exactly those code word lengths,known as the Shannon-Fano code, exists. We will not study this procedure because we already have an optimal procedure: the Huffman algorithm.

Relationship between Entropy and Prediction Error Rate
(Compressibility and Predictability)



Suppose we have a coin with P(heads) = p. We try to guess what the next coin flip will be: H or T.
If p = P(heads) > ½ the best we can do is always predict heads. Because the coin is bound to show p% heads our error rate is 1 – p.
If p = P(heads) < ½ then the best we can do is predict always tails. Then our error is p.
If p = ½ no matter what we do our error will be 0.5
So, the error for the best possible strategy is min(p, 1 – p). In the above graph the entropy and 2 * MinError = 2 * min(p, 1 – p) is plotted. We see that the error rate and the entropy are correlated.

High error => high entropy
low error => low entropy

In general: **P(error) >= [H(X) – 1] / Log[m]** where m = number of different outcomes of X.
(The above is known as Fano's inequality)

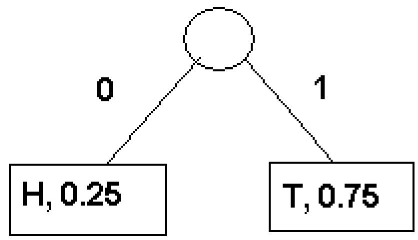
This example shows:
compressibility(being measured by the entropy) implies predictability(low entropy => low error rate)
predictability implies compressibility (low error rate => good compression)

Compressing the outcome of a coin flipping experiment.
Suppose we have flipped a coin with P(heads) = 1/4 exactly n times. As a result we have a string of 0's and 1's. Let us try to compress such a string. For example suppose we have the string (which was actually generated by a computer program):

1 0 0 0 1 0 0 0 0 0 1 0

(We write 1 for head, 0 for tail).
We know how to do that: apply Huffman code.

Outcome	H	T
Prob	0.25	0.75



Huffman code:
encode H with 0
encode T with 1

Let's encode our string:

original:	1	0	0	0	1	0	0	0	0	0	1	0
encoded:	0	1	1	1	0	1	1	1	1	1	0	1

We could not reduce the length of that string because we spend 1 bit for each symbol in the input alphabet {H, T}. But if we measure the entropy H(0.25) = 0.811278 we obtain that we should use not 1 bit per symbol but 0.8 bits (on average).

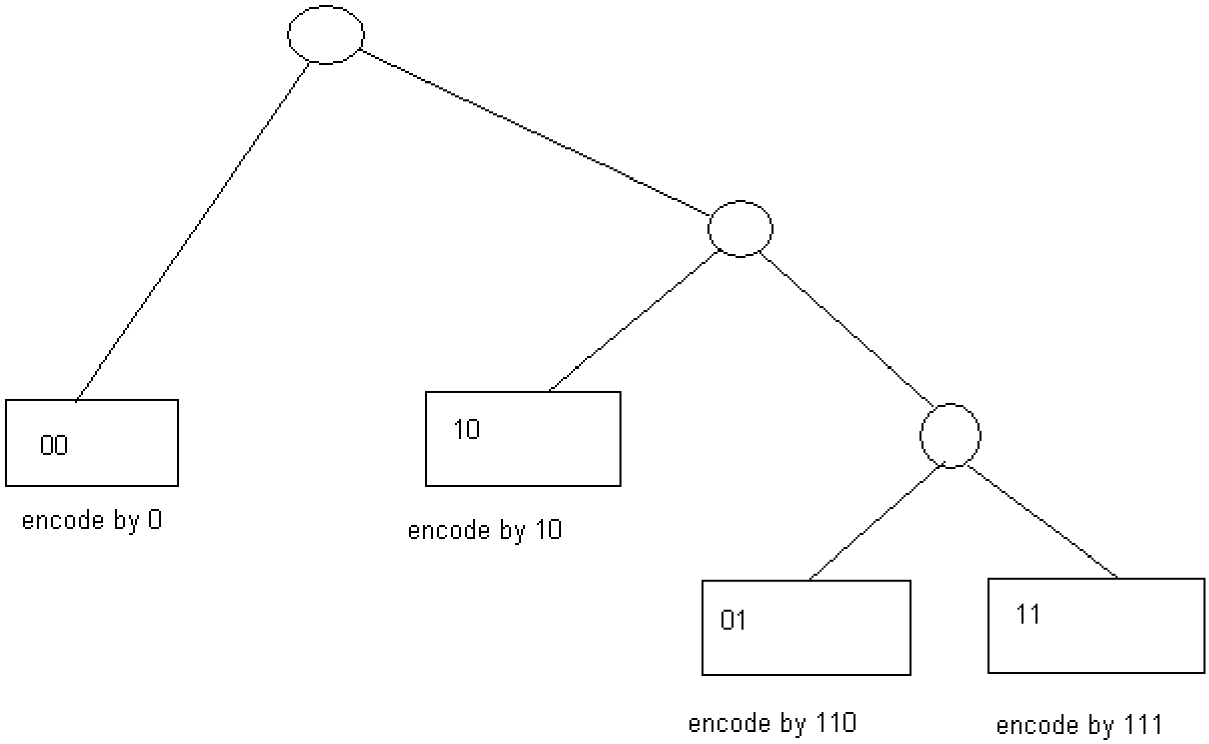
So, instead of encoding the outcome of each coin flip separately let us encode the outcome of each every two coin flips separately. We do the following:

Step 1: Split the input string into groups each group being equal to 2 bits:
10 00 10 00 00 10

Step 2: Treat each of the possibilities {00, 01, 10, 11} as a separate letter. For each of those letters compute its probability. For example Prob(01) = Prob(0) * Prob(1) = Prob(tail) * Prob(head) = 0.75 * 0.25 = because the coin flips are independent. We make a table of probabilities:

Outcome	00	01	10	11
Prob	0.75 * 0.75 = 0.5625	0.75 * 0.25 = 0.1875	0.25 * 0.75 = 0.1875	0.25 * 0.25 = 0.0625

Step 3: Run Huffman code on the table above.



Huffman code gives us the following code:
00 => 0, 01 => 110, 10 => 10, 11 => 111

Encode the string: 10 00 10 00 00 10

10 0 10 0 0 10
for which we use 9 instead of 12 bits.

This technique is known as **block-coding**.

Justification:
We need two facts:

H(X1, X2) = H(X1) + H(X2) when X1 and X2 are independent
H(X1, ..., Xn) = n H(X1) when X1, X2, ..., Xn are independent and have the same distribution (known as i.i.d.: independent and identically distributed)

For Huffman code:
H(X) <= HuffmanCodeAvgLengthInBits(X) < H(X) + 1

We apply this inequalities in the following way.
Let Y(n) be the random variable obtained by considering n consequtive coin flips.
Then
H(Y(n)) <= HuffmanCodeAvgLengthInBits(Y(n)) < H(Y(n)) + 1 (*)

But because the coin flips and independent: H(Y(n)) = n H(Y(1)) (**)
HuffmanCodeAvgLengthInBits(Y(n)) = If I input n bits, an average huffman will compress then to this many bits.
So, HuffmanCodeAvgLengthInBits(Y(n)) / n is the compression ratio (***).
So, from (*), (**), (***), the compression ratio is

H(p) <= compression ratio <= H(p) + 1/n, where p = Prob(head of coin) and n = length of block

Conclusion: if we use longer and longer blocks Huffman code approaches the entropy limit.

Why compression is possible: the asymptotic equipartition property
The simple answers: very few strings carry 99% of the total probability (compare to Zipf's law).
If we looks at all possible strings of length n we will see that there are 2^n of them.
However, only 2^(H(p) * n) carry let's say 99% of the total probability, where H(p) is the entropy of the coin that generated those strings.

Question: when is the entropy highest?
When all the outcomes have the same probability.
If there are m possible outcomes then the entropy is log(m) [binary log].

Question: when is the entropy lowest?
When there is no uncertainty, for example P(head) = 0.

Lempel-Ziv: an example of an universal compression algorithm

We illustrate Lempel-Ziv on couple of examples:

Example 1 for Lempel-Ziv:
Encoding of usa

Step 1: Convert each letter to its ASCII code
01110101 01110011 01100001

Step 2: Parse the binary string into unique substrings
0, 1, 11, 01, 010, 111, 00, 110, 1100, 001

Step 3: Replace each substring by its (back-ref, suffix-bit) pair
Note: if a substring is of length 1, by definition we will write 0 for back-ref
(0, 0), (0, 1), (1, 1), (3, 1), (1, 0), (3, 1), (6, 0), (5, 0), (1, 0), (3, 1)

Step 4: encode each back-ref by its binary representation
Note: the length of the bit string used to encode the back-ref depends on the consecutive number of the bit string, see below
back-ref 0 is at position 0 and therefore will be encoded by 0 bits
0 ==> _
back-ref 0 is at position 1 and therefore will be encoded by 1 bits
0 ==> 0

back-ref 1 is at position 2 and therefore will be encoded by 2 bits
1 ==> 01
back-ref 3 is at position 3 and therefore will be encoded by 2 bits
3 ==> 11
back-ref 1 is at position 4 and therefore will be encoded by 3 bits
1 ==> 001
back-ref 3 is at position 5 and therefore will be encoded by 3 bits
3 ==> 011
back-ref 6 is at position 6 and therefore will be encoded by 3 bits
6 ==> 110
back-ref 5 is at position 7 and therefore will be encoded by 3 bits
5 ==> 101
back-ref 1 is at position 8 and therefore will be encoded by 4 bits
1 ==> 0001
back-ref 3 is at position 9 and therefore will be encoded by 4 bits
3 ==> 0011

Summary of Step 4:
(0, 0), (0, 1), (1, 1), (3, 1), (1, 0), (3, 1), (6, 0), (5, 0), (1, 0), (3, 1)
(_, 0), (0, 1), (01, 1), (11, 1), (001, 0), (011, 1), (110, 0), (101, 0), (0001, 0), (0011, 1),

Result:
001011111001001111100101000010001111

Example 2 for Lempel-Ziv:
Decode 001011111001001111100101000010001111

Input: 001011111001001111100101000010001111

Step 1
Split input string into blocks where:

the size of block 1 is 1 bit(s) ***[one time 1 bit]***
the size of block 2 is 2 bit(s) ***[1 = 2^0 time 2 bits]***
the size of block 3 is 3 bit(s) ***[2 = 2^1 times 3 bits]***
the size of block 4 is 3 bit(s) ...
the size of block 5 is 4 bit(s) ***[4 = 2^2 times 4 bits]***
the size of block 6 is 4 bit(s)
the size of block 7 is 4 bit(s)
the size of block 8 is 4 bit(s)
the size of block 9 is 5 bit(s) ***[8 = 2^3 times 5 bits]***
the size of block 10 is 5 bit(s) ...

If we continue ,we will have ***[16 = 2^4 times 6 bits]***

0, 01, 011, 111, 0010, 0111, 1100, 1010, 00010, 00111

Step 2

Split each block into two pieces: back-ref string and a suffix bit.

The last bit is the suffix bit, everything before is the backref string

Convert the backref string from binary to decimal

(_, 0), (0, 1), (01, 1), (11, 1), (001, 0), ...

(_, 0), (0, 1), (1, 1), (3, 1), (1, 0), (3, 1), (6, 0), (5, 0), (1, 0), (3, 1),

Step 3

Process the list of pairs from left to right

If the first part of the pair is 0 or _, simply write the suffix bit (which is the second part of the pair)

If the first part of the pair is n > 0, then find the string that was corresponds to the n-th pair before the current one

Consider pair (0, 0),

Simply print the suffix bit (0, 0) => 0

Consider pair (0, 1),

Simply print the suffix bit (0, 1) => 1

Consider pair (1, 1),

Obtain the bit string that appeared 1 steps before

It is 1

Append the suffix bit to obtain (1, 1) => 11

Consider pair (3, 1),

Obtain the bit string that appeared 3 steps before

It is 0

Append the suffix bit to obtain (3, 1) => 01

Consider pair (1, 0),

Obtain the bit string that appeared 1 steps before

It is 01

Append the suffix bit to obtain (1, 0) => 010

Consider pair (3, 1),

Obtain the bit string that appeared 3 steps before

It is 11

Append the suffix bit to obtain (3, 1) => 111

Consider pair (6, 0),

Obtain the bit string that appeared 6 steps before

It is 0

Append the suffix bit to obtain (6, 0) => 00

Consider pair (5, 0),

Obtain the bit string that appeared 5 steps before

It is 11

Append the suffix bit to obtain (5, 0) => 110

Consider pair (1, 0),

Obtain the bit string that appeared 1 steps before

It is 110

Append the suffix bit to obtain (1, 0) => 1100

Consider pair (3, 1),

Obtain the bit string that appeared 3 steps before

It is 00

Append the suffix bit to obtain (3, 1) => 001

Step 4: Merge blocks into a bit string
011101010111001101100001

Step 5: split the bit string into groups, each group being 8 bits; convert each group to its ASCII code
01110101 = 117 ==> u
01110011 = 115 ==> s
01100001 = 97 ==> a

Result: usa

Example 2: Encode the word *information*

Step 1: Convert each letter to its ASCII code
01101001 01101110 01100110 01101111 01110010 01101101 01100001 01110100 01101001 01101111 01101110

Step 2: Parse the binary string into unique substrings
0, 1, 10, 100, 101, 1011, 1001, 10011, 00, 11, 01, 111, 011, 10010, 0110, 110, 10110, 000, 10111, 010, 001, 1010, 0101, 101111, 01101, 110,

Step 3: Replace each substring by its (back-ref, suffix-bit) pair
Note: if a substring is of length 1, by definition we will write 0 for back-ref
(0, 0), (0, 1), (1, 0), (1, 0), (2, 1), (1, 1), (3, 1), (1, 1), (8, 0), (8, 1), (10, 1), (2, 1), (2, 1), (7, 0), (2, 0), (6, 0), (11, 0), (9, 0), (13, 1), (9, 0), (12, 1), (17, 0),
(3, 1), (5, 1), (10, 1), (16, 0),

Step 4: encode each back-ref by its binary representation
Note: the length of the bit string used to encode the back-ref depends on the consecutive number of the bit string, see below

back-ref 0 is at position 0 and therefore will be encoded by 0 bits
0 ==>

back-ref 0 is at position 1 and therefore will be encoded by 1 bits
0 ==> 0

back-ref 1 is at position 2 and therefore will be encoded by 2 bits
1 ==> 01

back-ref 1 is at position 3 and therefore will be encoded by 2 bits
1 ==> 01

back-ref 2 is at position 4 and therefore will be encoded by 3 bits
2 ==> 010

back-ref 1 is at position 5 and therefore will be encoded by 3 bits
1 ==> 001

back-ref 3 is at position 6 and therefore will be encoded by 3 bits
3 ==> 011

back-ref 1 is at position 7 and therefore will be encoded by 3 bits
1 ==> 001

back-ref 8 is at position 8 and therefore will be encoded by 4 bits
8 ==> 1000

back-ref 8 is at position 9 and therefore will be encoded by 4 bits
8 ==> 1000

back-ref 10 is at position 10 and therefore will be encoded by 4 bits
10 ==> 1010

back-ref 2 is at position 11 and therefore will be encoded by 4 bits
2 ==> 0010

back-ref 2 is at position 12 and therefore will be encoded by 4 bits
2 ==> 0010

back-ref 7 is at position 13 and therefore will be encoded by 4 bits
7 ==> 0111

back-ref 2 is at position 14 and therefore will be encoded by 4 bits
2 ==> 0010

back-ref 6 is at position 15 and therefore will be encoded by 4 bits
    6 ==> 0110
back-ref 11 is at position 16 and therefore will be encoded by 5 bits
    11 ==> 01011
back-ref 9 is at position 17 and therefore will be encoded by 5 bits
    9 ==> 01001
back-ref 13 is at position 18 and therefore will be encoded by 5 bits
    13 ==> 01101
back-ref 9 is at position 19 and therefore will be encoded by 5 bits
    9 ==> 01001
back-ref 12 is at position 20 and therefore will be encoded by 5 bits
    12 ==> 01100
back-ref 17 is at position 21 and therefore will be encoded by 5 bits
    17 ==> 10001
back-ref 3 is at position 22 and therefore will be encoded by 5 bits
    3 ==> 00011
back-ref 5 is at position 23 and therefore will be encoded by 5 bits
    5 ==> 00101
back-ref 10 is at position 24 and therefore will be encoded by 5 bits
    10 ==> 01010
back-ref 16 is at position 25 and therefore will be encoded by 5 bits
    16 ==> 10000

Summary of Step 4:

(_, 0), (0, 1), (01, 0), (01, 0), (010, 1), (001, 1), (011, 1), (001, 1), (1000, 0), (1000, 1), (1010, 1), (0010, 1), (0010, 1), (0111, 0), (0010, 0), (0110, 0), (01011, 0), (01001
, 0), (01101, 1), (01001, 0), (01100, 1), (10001, 0), (00011, 1), (00101, 1), (01010, 1), (10000, 0),

Result:
00101001001010011011100111000010001101010010100101011100010001100010110010010011011
010010011001100010000111001011010101100000

Decode the string:

00101001001010011011100111000010001101010010100101011100010001100010110010010011011
010010011001100010000111001011010101100000
Input:
00101001001010011011100111000010001101010010100101011100010001100010110010010011011
010010011001100010000111001011010101100000

Step 1
Split input string into blocks where:
    the size of block 1 is 1 bit(s)
    the size of block 2 is 2 bit(s)
    the size of block 3 is 3 bit(s)
    the size of block 4 is 3 bit(s)
    the size of block 5 is 4 bit(s)

the size of block 6 is 4 bit(s)
the size of block 7 is 4 bit(s)
the size of block 8 is 4 bit(s)
the size of block 9 is 5 bit(s)
the size of block 10 is 5 bit(s)

0, 01, 010, 010, 0101, 0011, 0111, 0011, 10000, 10001, 10101, 00101, 00101, 01110, 00100, 01100, 010110, 010010, 011011, 010010, 011001, 100010, 000111, 001011, 010101, 100000,

Step 2

Split each block into two pieces: backref string and a suffix bit.

The last bit is the suffix bit, everything before is the backref string

Convert the backref string from binary to decimal

(0, 0), (0, 1), (1, 0), (1, 0), (2, 1), (1, 1), (3, 1), (1, 1), (8, 0), (8, 1), (10, 1), (2, 1), (2, 1), (7, 0), (2, 0), (6, 0), (11, 0), (9, 0), (13, 1), (9, 0), (12, 1), (17, 0),
(3, 1), (5, 1), (10, 1), (16, 0),

Step 3

Process the list of pairs from left to right

If the first part of the pair is 0, simply write the suffix bit (which is the second part of the pair)

If the first part of the pair is n > 0, then find the string that was corresponds to the n-th pair before the current one

Consider pair (0, 0),
Simply print the suffix bit (0, 0) => 0
Consider pair (0, 1),
Simply print the suffix bit (0, 1) => 1
Consider pair (1, 0),
Obtain the bit string that appeared 1 steps before
It is 1
Append the suffix bit to obtain (1, 0) => 10
Consider pair (1, 0),
Obtain the bit string that appeared 1 steps before
It is 10
Append the suffix bit to obtain (1, 0) => 100
Consider pair (2, 1),
Obtain the bit string that appeared 2 steps before
It is 10
Append the suffix bit to obtain (2, 1) => 101
Consider pair (1, 1),
Obtain the bit string that appeared 1 steps before
It is 101
Append the suffix bit to obtain (1, 1) => 1011
Consider pair (3, 1),
Obtain the bit string that appeared 3 steps before
It is 100
Append the suffix bit to obtain (3, 1) => 1001
Consider pair (1, 1),
Obtain the bit string that appeared 1 steps before
It is 1001
Append the suffix bit to obtain (1, 1) => 10011

Consider pair (8, 0),
    Obtain the bit string that appeared 8 steps before
        It is 0
        Append the suffix bit to obtain (8, 0) => 00
Consider pair (8, 1),
    Obtain the bit string that appeared 8 steps before
        It is 1
        Append the suffix bit to obtain (8, 1) => 11
Consider pair (10, 1),
    Obtain the bit string that appeared 10 steps before
        It is 0
        Append the suffix bit to obtain (10, 1) => 01
Consider pair (2, 1),
    Obtain the bit string that appeared 2 steps before
        It is 11
        Append the suffix bit to obtain (2, 1) => 111
Consider pair (2, 1),
    Obtain the bit string that appeared 2 steps before
        It is 01
        Append the suffix bit to obtain (2, 1) => 011
Consider pair (7, 0),
    Obtain the bit string that appeared 7 steps before
        It is 1001
        Append the suffix bit to obtain (7, 0) => 10010
Consider pair (2, 0),
    Obtain the bit string that appeared 2 steps before
        It is 011
        Append the suffix bit to obtain (2, 0) => 0110
Consider pair (6, 0),
    Obtain the bit string that appeared 6 steps before
        It is 11
        Append the suffix bit to obtain (6, 0) => 110
Consider pair (11, 0),
    Obtain the bit string that appeared 11 steps before
        It is 1011
        Append the suffix bit to obtain (11, 0) => 10110
Consider pair (9, 0),
    Obtain the bit string that appeared 9 steps before
        It is 00
        Append the suffix bit to obtain (9, 0) => 000
Consider pair (13, 1),
    Obtain the bit string that appeared 13 steps before
        It is 1011
        Append the suffix bit to obtain (13, 1) => 10111
Consider pair (9, 0),
    Obtain the bit string that appeared 9 steps before
        It is 01
        Append the suffix bit to obtain (9, 0) => 010
Consider pair (12, 1),

Obtain the bit string that appeared 12 steps before
It is 00
Append the suffix bit to obtain (12, 1) => 001
Consider pair (17, 0),
Obtain the bit string that appeared 17 steps before
It is 101
Append the suffix bit to obtain (17, 0) => 1010
Consider pair (3, 1),
Obtain the bit string that appeared 3 steps before
It is 010
Append the suffix bit to obtain (3, 1) => 0101
Consider pair (5, 1),
Obtain the bit string that appeared 5 steps before
It is 10111
Append the suffix bit to obtain (5, 1) => 101111
Consider pair (10, 1),
Obtain the bit string that appeared 10 steps before
It is 0110
Append the suffix bit to obtain (10, 1) => 01101
Consider pair (16, 0),
Obtain the bit string that appeared 16 steps before
It is 11
Append the suffix bit to obtain (16, 0) => 110
Step 4: Merge blocks into a bit string
0110100101101110011001100110111101110010011011010110000101110100011010010110111101101110
Step 5: split the bit string into groups, each group being 8 bits; convert each group to its ASCII code
01101001 = 105 ==> i
01101110 = 110 ==> n
01100110 = 102 ==> f
01101111 = 111 ==> o
01110010 = 114 ==> r
01101101 = 109 ==> m
01100001 = 97 ==> a
01110100 = 116 ==> t
01101001 = 105 ==> i
01101111 = 111 ==> o
01101110 = 110 ==> n

Result: information

Example 3: Encode the word *two*
Encoding of two

Step 1: Convert each letter to its ASCII code
01110100 01110111 01101111

Step 2: Parse the binary string into unique substrings

0, 1, 11, 01, 00, 011, 10, 111, 0110, 1111,

Step 3: Replace each substring by its (back-ref, suffix-bit) pair
Note: if a substring is of length 1, by definition we will write 0 for back-ref
(0, 0), (0, 1), (1, 1), (3, 1), (4, 0), (2, 1), (5, 0), (5, 1), (3, 0), (2, 1),

Step 4: encode each back-ref by its binary representation
Note: the length of the bit string used to encode the back-ref depends on the consecutive number of the bit string, see below
back-ref 0 is at position 0 and therefore will be encoded by 0 bits
0 ==>
back-ref 0 is at position 1 and therefore will be encoded by 1 bits
0 ==> 0
back-ref 1 is at position 2 and therefore will be encoded by 2 bits
1 ==> 01
back-ref 3 is at position 3 and therefore will be encoded by 2 bits
3 ==> 11
back-ref 4 is at position 4 and therefore will be encoded by 3 bits
4 ==> 100
back-ref 2 is at position 5 and therefore will be encoded by 3 bits
2 ==> 010
back-ref 5 is at position 6 and therefore will be encoded by 3 bits
5 ==> 101
back-ref 5 is at position 7 and therefore will be encoded by 3 bits
5 ==> 101
back-ref 3 is at position 8 and therefore will be encoded by 4 bits
3 ==> 0011
back-ref 2 is at position 9 and therefore will be encoded by 4 bits
2 ==> 0010

Summary of Step 4:

(_, 0), (0, 1), (01, 1), (11, 1), (100, 0), (010, 1), (101, 0), (101, 1), (0011, 0), (0010, 1),

Result:
001011111100001011010101100110001 01

Decoding 0010111111000010110101011001100010 1
Input: 00101111110000101101010110011000101

Step 1
Split input string into blocks where:
the size of block 1 is 1 bit(s)
the size of block 2 is 2 bit(s)
the size of block 3 is 3 bit(s)
the size of block 4 is 3 bit(s)
the size of block 5 is 4 bit(s)
the size of block 6 is 4 bit(s)

the size of block 7 is 4 bit(s)
the size of block 8 is 4 bit(s)
the size of block 9 is 5 bit(s)
the size of block 10 is 5 bit(s)
0, 01, 011, 111, 1000, 0101, 1010, 1011, 00110, 00101,

Step 2
Split each block into two pieces: backref string and a suffix bit.
The last bit is the suffix bit, everything before is the backref string
Convert the backref string from binary to decimal
(0, 0), (0, 1), (1, 1), (3, 1), (4, 0), (2, 1), (5, 0), (5, 1), (3, 0), (2, 1),

Step 3
Process the list of pairs from left to right
If the first part of the pair is 0, simply write the suffix bit (which is the second part of the pair)
If the first part of the pair is n > 0, then find the string that was corresponds to the n-th pair before the current one
Consider pair (0, 0),
Simply print the suffix bit (0, 0) => 0
Consider pair (0, 1),
Simply print the suffix bit (0, 1) => 1
Consider pair (1, 1),
Obtain the bit string that appeared 1 steps before
It is 1
Append the suffix bit to obtain (1, 1) => 11
Consider pair (3, 1),
Obtain the bit string that appeared 3 steps before
It is 0
Append the suffix bit to obtain (3, 1) => 01
Consider pair (4, 0),
Obtain the bit string that appeared 4 steps before
It is 0
Append the suffix bit to obtain (4, 0) => 00
Consider pair (2, 1),
Obtain the bit string that appeared 2 steps before
It is 01
Append the suffix bit to obtain (2, 1) => 011
Consider pair (5, 0),
Obtain the bit string that appeared 5 steps before
It is 1
Append the suffix bit to obtain (5, 0) => 10
Consider pair (5, 1),
Obtain the bit string that appeared 5 steps before
It is 11
Append the suffix bit to obtain (5, 1) => 111
Consider pair (3, 0),
Obtain the bit string that appeared 3 steps before
It is 011
Append the suffix bit to obtain (3, 0) => 0110

Consider pair (2, 1),

Obtain the bit string that appeared 2 steps before

It is 111

Append the suffix bit to obtain (2, 1) => 1111

Step 4: Merge blocks into a bit string

011101000111011101101111

Step 5: split the bit string into groups, each group being 8 bits; convert each group to its ASCII code

01110100 = 116 ==> t

01110111 = 119 ==> w

01101111 = 111 ==> o

Result: two