Chapter 6

Universal Source Coding

In the previous chapter we saw, from the souce coding theorem, that it is possible to reach a compression ratio equal to the entropy of the source. Futhermore, from Kraft inequality we can conclude that it is not possible to achieve more compression if we want to be able to decode it uniquely. That means the entropy can be used as a lower bound of the compression ratio of a sequence. With Huffman's algorithm it is possible to find an optimal code, which is no further away from the optimal bound than 1 bit. However, in this analysis we have made some assumtions. Firstly, for this construiction to be a practical implementation, the source symbols are considered independently and identically distributed (i.i.d.). Considering for example normal text or images this would require that idividual letters or pixels are independent of their neighbours. In reallity most sources are much more complex than this. A second assumption is that the statistics of the source is known. Again, in for exampple texts or images, the distribution of letters or pixels is dependent of the language or content. So, it turns out that none of these assumtions are very realistic. In this chapter we will look at algorithms that can performe compression for sources when the statistics is unknown. These type of algorithms are called universal source coding algorithms.

6.1 Dictionary coding

The class of universal according techniques include different versions of dictionary coding algorithms. The most obvious way is to use a predefined dictionary for the encoding. For example a version of the probabilities for the letters in English language, shown in Table 3.1, can be used to form a Huffman code. This will work quite good for normal English texts. However, if a text in another language is applied, the algorithm will not work as good any more. It gets even worse if it is used on e.g. a list of names, a program code or even an image, which has a completely different distribution of the characters.

For a text where the statistics is relatively known it is also possible to make a code construction inspired by Shannon's source coding theorem. Assume that a text takes letters from the ASCII table, i.e. 256 letters. Then we block the letters in groups of four and sort out the $2^{10} = 1024$ most likly 4 tuples. These should be encoded with as few bits as possible. Assumin that they are about equaly likely they have the same length, 10 bits. To differ it from the case when they are not in the list the codeword is prefixed by a 1. That gives a length of 11 bits. The 4 tuples that are not in the list should then be encoded with $\log(256^4) = 32$ bits with a prefix of a 0, giving 33 bits. If the probability that a 4 tuple is in the list of priobable tuples is p, we get the average codeword length to

$$L = p11 + (1-p)33 = 33 - 22p$$

For this scheme to have any compression at all this length should be smaller than the uncoded case 32 bits. This gives that

$$p = \frac{1}{22} \approx 0.0455$$

Here we should notice that even if the number 1024 sounds relativly high it is only a fraction, i.e. 2^{-22}, of all possible 4 tuples and that the probability p is about $2^{-4.5}$. It is required good knowledge of the text statistics to find tuples that sum up to this number. Furthermore, for this to be worthwile we would not only like to have a result slightly better that the uncoded case, which will require even more precision in the dictonary.

6.2 Adaptive Coding

It is only in very specific cases when it is possible to know statiscs of the source in such details allowing the code to work from a fixed dictionary. Instead it is often more efficient to use an adaptive dictionary that is built from the data that is to be compressed. Therer is a vast variaty of this can be done. Here we will first describe how the source text it self can be used to find the statistics, then a more advanced version of it will be described. Among the adaptive methods there is a class of algorithm known as LZ codes, based on the work by Lempel and Ziv. These algorithms will be described in the following two sections.

6.2.1 Two sweep algorithm

A direct generalization of the fixed dictionary encoding is to have a two pass algorithm. First the text is processed ones, where the distribution of the letters is derived. Then, for example a Huffman code for this distribution can be used to generate generate the code sequence. This requires that the source text is finite since the first pass must finish. The procedure can give a fairly good compression rate but there is a starting step since the dictionary must be submitted along with the code string.

After the first pass of the algorithm the estimted distribution might not be all correct. The compression loss made by assuming the wrong distribution can be derived by assuming that $p(x)$ is the true distribution for the random variable X, and $q(x)$ the estimted. The optimal codeword length for the source is $L^{(opt)} = E_p[-\log p(x)] = H(X)$. When using $q(x)$ instead we get an average length of $L = E_p[-\log q(x)]$. The difference between the optimal length and the achieved is

$$\begin{aligned} L - L^{(opt)} &= -\sum_x p(x)\log q(x) + \sum_x p(x)\log p(x) \\ &= \sum_x p(x)\log\frac{p(x)}{q(x)} = D\big(p(x)||q(x)\big) \end{aligned}$$

We see here that the relative entropy comes back here as the penalty paid for having mismatch in the distribution of the code and the source.

6.2.2 Adaptive Huffman

There is also an adaptive variant of the Huffman code where the code tree is built during the same pass as the text is encoded. Then the code tree and the distribution depends on the symbols seen so far in the process. The problem here is to update the tree in a efficient way after each symbol in the text, both at the encoder and the decoder side.

To be done.

6.3 LZ77 (Lempel-Ziv 1977)

There is another class of algorithms based on the ideas of Abraham Lempel and Jacob Ziv, who published the paper *A Universal Algorithm for Sequential Data Compression* in 1977, [23]. The algorithm described is often referred to as LZ77. They also published a variant of the algorithm, based on the same observations, in 1978 called *Compression of Individual Sequences via Variable-Rate Coding*, [24], often refered to as LZ78. The basic idea is to process the text once and to build an adaptive dictoionary during the process. In the first version from 1977 a window of previous symbols are used and in LZ78 a dictionary covering the repetitions of the past in the text is used.

These two publications are the origins of two families of algorithms. They are very spread and used in priactical applications covering text compression (e.g. deflate, zip, gz, 7z) image compression (e.g. png and tiff). We will in this chapter have a look at one improved version of LZ77, called LZSS, and one improved version of LZ78, called LZW.

The base of the LZ77 algorithm is a sliding window technique with two buffers, one search buffer of length S and and one look ahead buffer of length B. If the character x_n at time n should be encoded, the search buffer consists of the S previos characters, $(x_{n-S}, \dots, x_{n-1})$. The look ahead buffer consists of the next B characters, including the present, $(x_n, \dots, x_{n+B-1})$. In Figure 6.1 a schematic over the buffers is shown. The idea is that texts often contains repetitions, so there should be a not too low probability that x_n and a couple of its successors can also be found in the closest previous characters. In this way we can use the search buffer as an adaptive dictionary. The algorithm searches the dictionary for the longest match of x_n and its sucessors in the look ahead buffer. If there is such a match it will give an index j to the start of the match and a length l of the match. The index j is the offset between the match in the search buffer and in the look ahead buffer, i.e. the number of positions from x_n to the start of the match. In the case when there is no match, set $j, l = 0, 0$. To get on after a missing match the next character in the string should also be attached to the codeword. In Algorithm 3 the procedure is described.

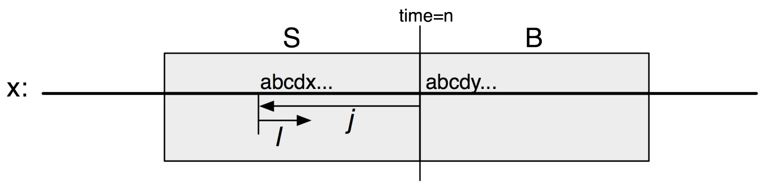


Figure 6.1: The search buffer and the look ahead buffer in LZ77.

```
Algorithm 3 (LZ77)
Initialization:
    Initialize the seach buffer with the first $S$ characters of the text, and set $n = S + 1$.

Continue until end of text
    Find all offsets of $x_n$ in the search buffer.
    If number offsets > 0 [i.e. there are at least one match]
        Find the offset with the longest match. This gives an index $j$ and a length $l$.
        Set the codeword to $(j, l, c)$
        where $c$ is the next character after the match.
        Set $n \leftarrow n + l + 1$.
    Else [i.e. there are no matches]
        Set the codeword to $(0, 0, c)$
        where $c = x_n$.
        Set $n \leftarrow n + 1$.
    Update search buffer and look ahead buffer
```

~~The two numbers j and l are bounded by the buffer lengths, $j \le S$ and $l \le B$. So, to~~ express these in binary form we need $\lceil \log(S + 1) \rceil$ and $\lceil \log(B + 1) \rceil$, respectively. Here we have assumed that number zero should also be possible to set. If we use the normal ASCII table to express the characters in binary form we need eight bits each. Hence, a binary representation of a codeword requires

$$\ell\big((j, l, c)\big) = \lceil \log(S + 1) \rceil + \lceil \log(B + 1) \rceil + 8$$

To see how the algorithm works we will have a look at an example.

Example 39 [LZ77] To encode the text string

$$T = \text{can␣you␣can␣cans␣as␣a␣canner␣can␣can␣cans?}$$

using the LZ77 algorithm with $S = 15$ and $B = 7$ we use a table to show the results. In the string '␣' denotes a space. In the following table the search buffer and the look ahead buffer is shown for each codeword generated.

Step	S buffer	B buffer	Codeword
1	can␣you␣can␣can	s␣as␣a␣	(0,0,s)
2	an␣you␣can␣cans	␣as␣a␣c	(5,1,a)
3	␣you␣can␣cans␣a	s␣a␣can	(3,3,␣)
4	␣can␣cans␣as␣a␣	canner␣	(10,3,n)
5	␣cans␣as␣a␣cann	er␣can␣	(0,0,e)
6	cans␣as␣a␣canne	r␣can␣c	(0,0,r)
7	ans␣as␣a␣canner	␣can␣ca	(7,4,␣)
8	s␣a␣canner␣can␣	can␣cans	(4,7,s)
9	er␣can␣can␣cans	?	(0,0,?)

In the table we have denoted a mathch of the first letter from the B buffer with a ◦, and underlined the match length for each match. The longest match is also underlined in the B buffer. The initialization of the S buffer is done with the 16 first first letters. We will go through the algorithm step by step:

1. To start, the first character of the B buffer is 's'. This does not exists in the S buffer so the codeword becomes (0,0,s).
2. In the second step there are three matches of the first symbol, '␣', in the B buffer. All three of them give a match of length 1, so we can choose any of the codewords (5,1,a), (9,1,a) or (13,1,a). In our case we have chosen the first match from the right.
3. there is only one match of 's'. This gives a match length of 3, and the codeword is (3,3,␣).
4. There are two 'c' in the S buffer, on position 10 and 14. Both have a match length of 3. We have again chosen the codeword with the shortest back track, (10,3,n).
5. 'e' cannot be found in the S buffer, giving the codeword (0,0,e).
6. 'r' cannot be found in the S buffer, giving the codeword (0,0,r).
7. There are three '␣' in the S buffer, on 7, 9 and 12. The first has the longest match, four, giving the codeword (7,4,␣).
8. In the S buffer there are two 'c', at 4 and 11. The match at 11 has length 3 and the match at 4 has length 7, hence the codeword is (4,7,s). Notice that the match continues in to the B buffer. This is not a problem sinceat decoding the S buffer is known at each step, and the match is built up from left to right.
9. There is no '?' in the S buffer s the codeword is (0,0,?).

That is, we get the encoded sequence:

$$C = (0,0,s)(5,1,a)(3,3,␣)(10,3,n)(0,0,e)(0,0,r)(7,4,␣)(4,7,s)(0,0,?)$$

Here we have not showed the initialization of the S buffer, but these 15 characters must also be included in the code sequence. If we assume the characters are encoded by the

ASCII table using eight bits each, the uncoded string requires $42 \cdot 8 = 336$ bits. The requires for the initialization $15 \cdot 8 = 120$ bits. Each codeword needs $\lceil \log 16 \rceil + \lceil \log 8 \rceil + 8 = 15$ bits and there are 9 codewords. In total the encoded sequence can be represented with $120 + 9 \cdot 15 = 255$ bits. We see that even this small string, with relativly small buffers, gives a compression with rate $R = 0.7589$.

Decoding can now be done with the initialization sequence of the S buffer and the codewords. The next table show how this is preformed. In the S buffer the match from the codeword is marked.

Step	Codeword	S buffer	New symbols
1	(0,0,s)	can␣you␣can␣can	s
2	(5,1,a)	an␣you␣can␣cans	␣a
3	(3,3,␣)	␣you␣can␣cans␣a	s␣a␣
4	(10,3,n)	␣can␣cans␣as␣a␣	cann
5	(0,0,e)	␣cans␣as␣a␣cann	e
6	(0,0,r)	cans␣as␣a␣canne	r
7	(7,4,␣)	ans␣as␣a␣canner	␣can␣
8	(4,7,s)	s␣a␣canner␣can␣	can␣cans
9	(0,0,?)	er␣can␣can␣cans	?

6.3.1 LZSS

There are a variaty of improvments that can be done for the algorithm to have a better compression ratio. Here we will describe one such improvement, published in [LZSS]. Often, this version of the algorithm is called LZSS, after the inventors Storer and Szymanski in 1982. It is based on the observation that in each codeword there is also an uncoded character. This is only necessary in the case when there is no match. If each codeword is preceded by a binary flag, stating if there is a match or not, the effect of the uncoded character can be reduced. So the modified algorithm is described in Algorithm 4.

Algorithm 4 (LZSS)

```
Initialization:
    Initialize the seach buffer with the first $S$ characters of the text, and set $n = S + 1$.

Continue until end of text
    Find all offsets of $x_n$ in the search buffer.
    If number offsets > 0 [i.e. there are at least one match]
        Find the offset with the longest match. This gives an index $j$ and a length $l$.
        Set the codeword to $(0, j, l)$
        where $c$ is the next character after the match.
        Set $n \leftarrow n + l$.
    Else [i.e. there are no matches]
        Set the codeword to $(1, c)$
        where $c = x_n$.
    *Set* $n \leftarrow n + 1$.
  *Update search buffer and look ahead buffer*
```

To illustrate the algorithm we use the same text as we had for the LZ77 algorithm.

Example 40 [LZSS] To encode the text string

$T =$ can␣you␣can␣cans␣as␣a␣canner␣can␣can␣cans?

using the LZSS algorithm with $S = 15$ and $B = 7$ we use a table to show the results. In the string '␣' denotes a space. In the following table the search buffer and the look ahead buffer is shown for each codeword generated.

Step	S buffer	B buffer	Codeword
1	can␣you␣can␣can	s␣as␣a␣	(1,s)
2	an␣you␣can␣cans	␣as␣a␣c	(0,5,1)
3	n␣you␣can␣cans␣	as␣a␣ca	(0,4,1)
4	␣you␣can␣cans␣a	s␣a␣can	(0,3,3)
5	u␣can␣cans␣as␣a	␣canner	(0,10,4)
6	n␣cans␣as␣a␣can	␣ner␣can	(0,1,1)
7	␣cans␣as␣a␣cann	er␣can␣	(1,e)
8	cans␣as␣a␣canne	r␣can␣c	(1,r)
9	ans␣as␣a␣canner	␣can␣ca	(0,7,4)
10	as␣a␣canner␣can	␣can␣ca	(0,4,7)
11	nner␣can␣can␣ca	ns	(0,4,1)
12	ner␣can␣can␣can␣	s	(1,s)
13	er␣can␣can␣cans	?	(1,?)

The encoding follows the same principles as for LZ77. One thing that is shown in this example that eas not present in Example 39 is seen in step 10. The match is of length 7 but it is limited by the length of the look ahead buffer. If the buffer would be longer, so would the match. The choise of lengths on the buffers comes from a balance between having enough matches in the dictionary and lengths in the match, and the number of bits used to describe the codewords.

The codeword for the case when there is a match can now be represented with $1 + \lceil \log 16 \rceil + \lceil \log 8 \rceil = 8$ bits. In the case when there is now mathch we need a binary flag in front of the character, resulting in the codeword length $1 + 8 = 9$ bits. There are in total 8 codewords of length 8 and 5 of length 9, resulting in totally $15 \cdot 8 + 8 \cdot 8 + 5 \cdot 9 = 229$ bits. The uncoded sequence gives, as before, 336 bits. The compression rate becomes $R = 0.6815$.

To decode the the code sequence we can use the following table.

Step	Codeword	S buffer	New symbols
1	(1,s)	can␣you␣can␣can	s
2	(0,5,1)	an␣you␣can␣cans	␣
3	(0,4,1)	n␣you␣can␣cans␣	a
4	(0,3,3)	␣you␣can␣cans␣a	s␣a
5	(0,10,4)	u␣can␣cans␣as␣a	␣can
6	(0,1,1)	n␣cans␣as␣a␣can	n
7	(1,e)	␣cans␣as␣a␣cann	e
8	(1,r)	cans␣as␣a␣canne	r
9	(0,7,4)	ans␣as␣a␣canner	␣can
10	(0,4,7)	as␣a␣canner␣can	␣can␣ca
11	(0,4,1)	nner␣can␣can␣ca	n
12	(1,s)	ner␣can␣can␣can␣	s
13	(1,?)	er␣can␣can␣cans	?

There are other improvments that canbe done to the algorithm also, but in this text we will not consider more versions of LZ77. However, there is another type of improvment also that is common to use. It is based on using a two pass procedure, where the first processing consits of LZ77 or LZSS, and the second processing is a Huffman code of the codewords. This will give some more compression to the system.

6.4 LZ78

In 1978 Lempel and Ziv published a followup article on the fuirs one [24]. In this they give an alternative version of a compression algorithm, often called LZ78. It is based on the same idea that a dictionary is built during a one pass sweep of the text. The first algorithm, LZ77, can get into problems when the repetions in the text are longer than the search buffer. Then it might actually lead to an expansion of the text. The problem is that the dictionary is built with a sliding window techinque, and that not all the previous text is represented. In LZ78 a dictionary is built from all the past of the file. The dictionary can be represented in a tree which makes it efficient from a memory and search point of view. In the dictionary past strings are stored and indexed. When a new match is found the dictionary is expanded by this matching string concatenated with the following symbol. In that way we build a dictionary with matches of different lengths. To see how this works we first set up the algorithm in Algorithm 5.

Algorithm 5 (LZ78)

```
Initialize
    The dictioary contains the empty symbol at index 0
    $n = 1$ [The first symbol]
    $Ind = 1$ [Next index in the dictionary]

Continue until end of text
    Find $x_n$ in the dictionary.
    If there is a match
        Find the longest match $x_n \dots x_{n+l-1}$ in the dictionary.
        Set the codeword $(Ind_m, x_{n+l})$, where $Ind_m$ is the index of the match
        Add $x_n \dots x_{n+l-1} x_{n+l}$ with index $Ind$ to the dictionary
        Set $n \leftarrow n + l + 1$.
    Else [$x_n$ not on dictionary]
        Set the codeword to $(0, x_n)$
        Add $x : n$ with index $Ind$ to the dictionary
    $Ind \leftarrow Ind + 1$
```

At step Ind the match ondex of the codeword is in $[0, Ind-1]$, and it is needed $\lceil \log Ind \rceil$ bits to describe it. If the alphabet size is $|\mathcal{A}|$ it is neede in total $L_{Ind} = \lceil \log Ind \rceil + \lceil \log |\mathcal{A}| \rceil$ bits to describe the codeword at step Ind.

In the next example we use the same text string as before, for the LZ78 algorithm.

Example 41 Encode the text string

$$T = \text{can␣you␣can␣cans␣as␣a␣canner␣can␣can␣cans?}$$

using the LZSS algorithm. First we show the text again with the encoded parts underlined and marked with index of the codeword. Then., in the following table, the corresponding codewords and the dictionary is shown. Also, the binary representation of the codeword is shown to the right in the table. For the binary representation we have assumed the ASCII table with eigth bits for each character.

$$T=\underbrace{\text{c}}_{1}\underbrace{\text{a}}_{2}\underbrace{\text{n}}_{3}\underbrace{\text{␣}}_{4}\underbrace{\text{y}}_{5}\underbrace{\text{o}}_{6}\underbrace{\text{u}}_{7}\underbrace{\text{␣c}}_{8}\underbrace{\text{an}}_{9}\underbrace{\text{␣ca}}_{10}\underbrace{\text{ns}}_{11}\underbrace{\text{␣a}}_{12}\underbrace{\text{s␣}}_{13}\underbrace{\text{a␣}}_{14}\underbrace{\text{ca}}_{15}\underbrace{\text{nn}}_{16}\underbrace{\text{e}}_{17}\underbrace{\text{r}}_{18}\underbrace{\text{␣can}}_{19}\underbrace{\text{␣can␣}}_{20}\underbrace{\text{can}}_{21}\underbrace{\text{s?}}_{22}$$

	Ind	Codeword	Dictionary	bits	Binary
Init	0		–		
	1	0,c	c	0	01100011
	2	0,a	a	1	0 01100001
	3	0,n	n	2	00 01101110
	4	0,␣	␣	2	00 00100000
	5	0,y	y	3	000 01111001
	6	0,o	o	3	000 01101111
	7	0,u	u	3	000 01110101
	8	4,c	␣c	3	100 01100011
	9	2,n	an	4	0010 01101110
	10	8,a	␣ca	4	1000 01100001
	11	3,s	ns	4	0011 01110011
	12	4,a	␣a	4	0100 01100001
	13	0,s	s	4	0000 01110011
	14	12,␣	␣a␣	4	1100 00100000
	15	1,a	ca	4	0001 01100001
	16	3,n	nn	4	0011 01101110
	17	0,e	e	5	00000 01100101
	18	0,r	r	5	00000 01110010
	19	10,n	␣can	5	01010 01101110
	20	19,␣	␣can␣	5	10011 00100000
	21	15,n	can	5	01111 01101110
	22	13,?	s?	5	01101 00111111

In total we need 255 bits to represent the codewords and the uncoded sequence needs 336 bits. That imply a compression rate of $R = 0.7589$. The representation of the dictionary is better done in a tree structure. For this case it is shown in Figure 6.2.

The decoding is done straight forward very similarly. The same dictionary can be built since everything about the sequence up to the current codeword is known. We will not go into details of how this is achieved.

As with the LZ77 algorithm, there are several possible improvments for this algorithm. The most common one was published by Terry Welsh in 1984. It is usually named the LZW algorithm, and is described next. The LZW algorithm is the base in compression standards like the UNIX compress and the image format GIF (Graphics Interchange Format). GIF was very popular until December 28, 1994 when Unisoft, who had a patent for LZW, declared that they would start charging royalties for programs supporting GIF. The reactions on the web was very strong, and resulted the formation of the PNG group. Already in March 7, 1995 the first PNG image was published. PNG, at first standing for *PiNG is Not GIF* but now normally *Portable Network Graphics*, is patent free and based on LZ77. Even if there are patents very close to PNG, the implementation can use work-arounds [7].

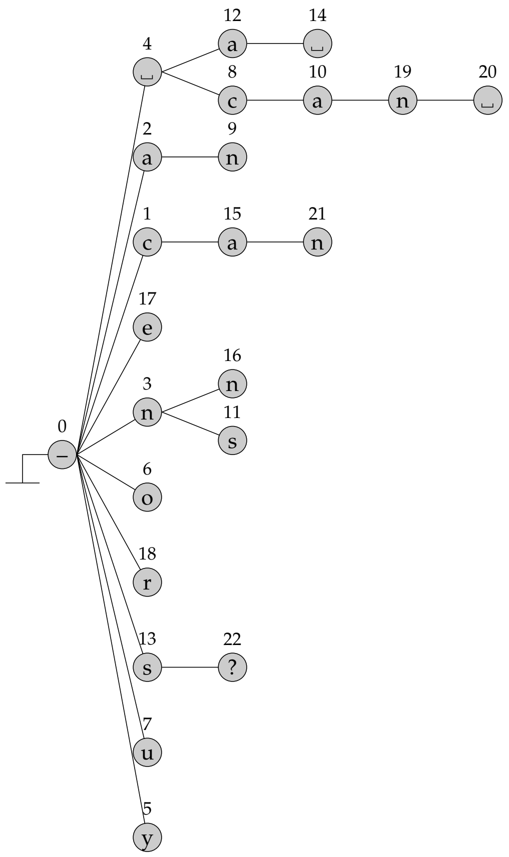


Figure 6.2: The dictionary in a tree form for example 41.

6.4.1 LZW

To cope with the case when the first encoded character is not in the dictionary, it is needed that there is an uncoded character in the codeword. In this way the dictionary will be expanded with that character. However, this is the only reason for having an uncoded character in the codeword and in most cases it is not needed. This is the same problem as in LZ77 that was solved with two different codewords. Here, we can instead initialize the dictionary with the complete aphabet, which is known by both the endoder and decoder side. In this way we no longer have a need for the extra character in the codeword adn can use just the index of the match. We will see that it is still possible to extend the dictionary with the match concatenated with the fiollowing character. This is the base of the improvement by Welch, often denoted LZW [21].

Algorithm 6 (LZW)

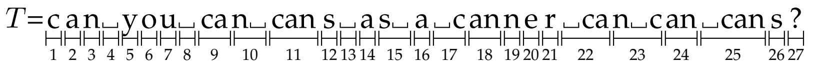
```
Initialize
    Initialize the dictionary with the complete alphabet $\mathcal{A}$.
    $n = 1$ [The first symbol inthe sequence]
    $Ind = |\mathcal{A}| + 1$ [Next index in the dictionary]

Continue until end of text
```

```
    Find $x_n$ in the dictionary.
    Find the longest match $x_n \dots x_{n+l-1}$ in the dictionary.
    Set the codeword $(Ind_m)$, where $Ind_m$ is the index of the match
    Add $x_n \dots x_{n+l-1}x_{n+l}$ with index $Ind$ to the dictionary
    Set $n \leftarrow n + l + 1$.
    $Ind \leftarrow Ind + 1$
```

As in LZ78 the number of bits to describe the index of the codeword depends on the length of the dictionary.

Example 42 We will use the same text string as before to show how the algorithm works. What first needs to be decided is what alphabet to use. Of course we can, for pedagogic resons, say that it is only the present letters, $\mathcal{A}_1 = \{\llcorner\!\lrcorner, ?, a, c, e, n, o, r, s, u, y\}$. That would give a very unfair estimate of the compressionrate since we campare with the uncoded case where the characters are taken from the ASCII table with eight bits. Therefore, we have here chosen to use the ASCII table as our alphabet, which contains 256 characters. To shorten the table of the encoding a bit we only show the entrance of the alphabet that is actually used.

First we show the encoded text again where we mark the parts that are encoded, and which codeword it corresponds to.



Next, the table showing how the codewords are formed and the dictionary grows.

	n	Codeword	Ind	Dictionary	bits	Binary
			32	␣		
			63	?		
			97	a		
			99	c		
			101	e		
			110	n		
			111	o		
			114	r		
			115	s		
			117	u		
Init			121	y		
	1	99	256	ca	8	01100011
	2	97	257	an	9	001100001
	3	110	258	n␣	9	001101110
	4	32	259	␣y	9	000100000
	5	121	260	yo	9	001111001
	6	111	261	ou	9	001101111
	7	117	262	u␣	9	001110101
	8	32	263	␣c	9	000100000
	9	256	264	can	9	100000000
	10	258	265	n␣c	9	100000010
	11	264	266	cans	9	100001000
	12	115	267	s␣	9	001110011
	13	32	268	␣a	9	000100000
	14	97	269	as	9	001100001
	15	267	270	s␣a	9	100001011
	16	97	271	a␣	9	001100001
	17	263	272	␣ca	9	100000111
	18	257	273	ann	9	100000001
	19	110	274	ne	9	001101110
	20	101	275	er	9	001100101
	21	114	276	r␣	9	001110010
	22	272	277	␣can	9	100010000
	23	265	278	n␣ca	9	100001001
	24	257	279	an␣	9	100000001
	25	277	280	␣cans	9	100010101
	26	115	281	s?	9	001110011
	27	63	282	?×	9	000111111

In total we need 242 bits to represent the codewords, giving a compression rate of $R = 0.7202$. Similar to LZ78 the dictionary can be represented in a tree structure, as in Figure 6.3.

What is left is to show how the same disctionary can be built at the receiver side. The codeword specifies the index of the dictionary, and that gives directly the corresponding text. But then the new entrance in the dictionary consists of this text concatenated with the next character of the text. At this moment we do not have this. But directly when we start decoding the next codeword we will get. Hence, at that point we can allways

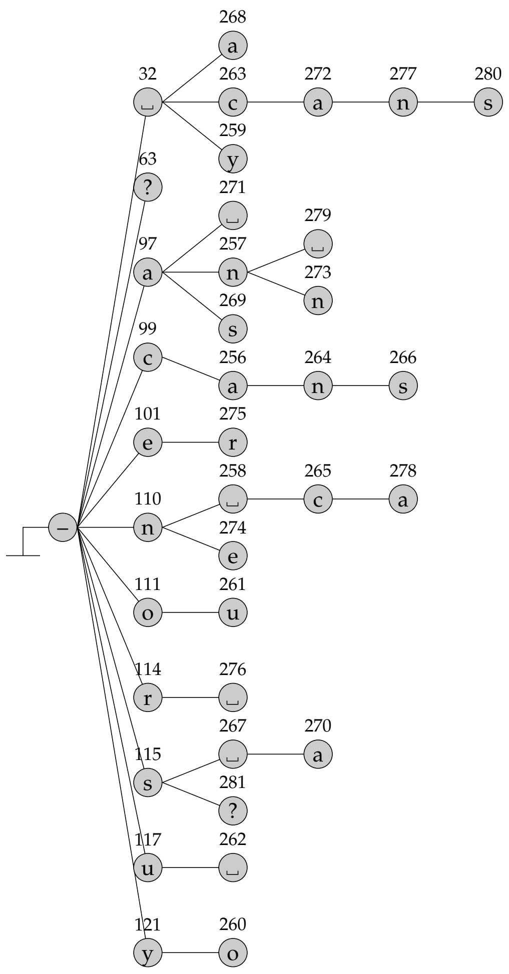


Figure 6.3: The dictionary in a tree form for example 42.

complement the new dictionary and set it in the tree. In the next example we show the first couple of steps decoding the previously encoded sequence.

Example 43 Let us initialize the decoding of LZW with the ASCII table and decode

$$C = (99, 97, 110, 32, 121, 111, 117, 32, 256, 258, 264, 115, 32, 97, 267, 97, 263, 257, 110, 101, 114, 272, 265, 257$$

The initialization is of course done with the whole ASCII table, but the part that we will use is shown in the folloowing table.

Ind	Dictionary
32	␣
63	?
97	a
99	c
101	e
110	n
111	o
114	r
115	s
117	u
121	y

The first codeword, 99, says then that the text starts with ′c′. The dictionary should be expanded with ′c×′, where × denote the next character of the text. With this as a start we fill in a table used for decoding. The first line becomes

n	Codeword	Text	Ind	Dictionary
1	99	c	256	c×

The next codeword is 97, meaning the next character is ′a′. We can now complete the dictionary entrance at index 256 and start with the next line in the table.

n	Codeword	Text	Ind	Dictionary
1	99	c	256	ca
2	97	a	257	a×

Similarly, the dictionary entrance at index 257 can be completed when the next codeword is decoded. This is 110, which gives an ′n′. The table becomes

n	Codeword	Text	Ind	Dictionary
1	99	c	256	ca
2	97	a	257	an
3	110	n	257	n×

This way we can allways fill in all entrances of the dictionary. We are always taking the first character of the next decoded text string and attache to the end of the dictionary entrance. The complete decoding table becomes

n	Codeword	Text	*Ind*	Dictionary
1	99	c	256	ca
2	97	a	257	an
3	110	n	258	n␣
4	32	␣	259	␣y
5	121	y	260	yo
6	111	o	261	ou
7	117	u	262	u␣
8	32	␣	263	␣c
9	256	ca	264	can
10	258	n␣	265	n␣c
11	264	can	266	cans
12	115	s	267	s␣
13	32	␣	268	␣a
14	97	a	269	as
15	267	s␣	270	s␣a
16	97	a	271	a␣
17	263	␣c	272	␣ca
18	257	an	273	ann
19	110	n	274	ne
20	101	e	275	er
21	114	r	276	r␣
22	272	␣ca	277	␣can
23	265	n␣c	278	n␣ca
24	257	an	279	an␣
25	277	␣can	280	␣cans
26	115	s	281	s?
27	63	?	282	?×

6.5 Applications of LZ

The LZ algorithms have become very popular due to its combination of relative simplesness and efficiency and have been incorporated in many standard compressin schemes.

The LZ algotihms are used in many applications. Among others:

- LZ77
  - deflate
  - zip, gzip, 7z
  - PNG
- LZW
  - compress
  - GIF, TIFF

    - V.42 bis

- LZ77: Variable to fix
- LZSS, LZ78, LZW: Variable to variable

6.6 Optimality of LZ

To be done.

6.7 PNG

To be done.