

---

## Chapter 5

# DICTIONARY TECHNIQUES

Yeuan-Kuen Lee  
[ MCU, CSIE ]



---

## Outline

- 5.1 Overview
- 5.2 Introduction
- 5.3 Static Dictionary
- 5.4 Adaptive Dictionary
- 5.6 Summary



## 5.1 Overview

source → independent symbols

- ✓ In the previous two chapters we looked at coding techniques that assume a source that generate a sequence of **independent symbols**.

source → decorrelation → independent symbols

- ✓ As most sources are correlated to start with, the coding step is generally preceded by a **decorrelation step**.
- ✓ In this chapter we will look at techniques that incorporate the structure in the data in order to increase the amount of compression.



## 5.1 Overview

- ✓ These techniques - both **static** and **adaptive** (or **dynamic**) - build a **list** of commonly occurring patterns and encode these patterns by transmitting their **index** in the list.
- ✓ They are most useful with **sources** that generate a **relatively small number of patterns quite frequently**, such as text source and computer commands.
- ✓ We discuss two applications in these areas: the **UNIX compress** command and the **V4.2 bis specifications**.
- ✓ A discussion of the **Graphics Interchange Format (GIF)** highlights the limitations of this approach.

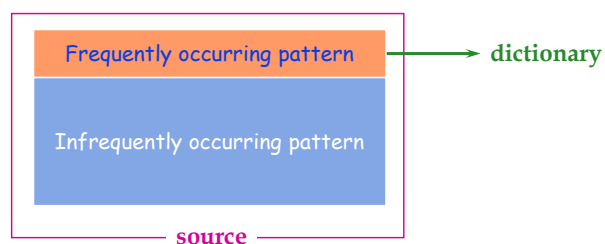


## 5.2 Introduction

- ✓ In many applications, the output of the source consists of **recurring patterns**. A classic example is a text source in which certain patterns or words recur constantly.  
ex. **Limpopo**
- ✓ A very reasonable approach to encoding such source is to keep a **list**, or **dictionary**, of frequently occurring patterns.
- ✓ When these patterns appear in the source output, they are encoded with a **reference** to the dictionary.
- ✓ If the pattern does not appear in the dictionary, then it can be encoded using some other, **less efficient**, method.



## 5.2 Introduction

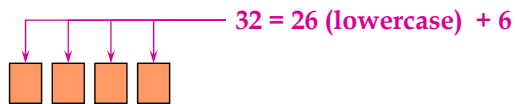


- ✓ In effect we are splitting the input into two classes, **frequently occurring patterns** and **infrequently occurring patterns**.
- ✓ For this technique to be effective, the **class of frequently occurring patterns**, and hence **the size of the dictionary**, must be **much smaller** than the number of all possible patterns.



## 5.2 Introduction

**Example** Four-character words  
Three-character words + punctuation mark



Comma ,  
Period .  
Exclamation mark !  
Question mark ?  
Semicolon ;  
Colon :

$$32^4 = 2^{20} = 1,048,576$$

- ✓ Treating all  $32^4$  four-character patterns as equally likely, we have a code that assigns **20 bits** to each four-character pattern.



## 5.2 Introduction

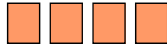
**Example** 

- ✓ Let us put the **256 most likely** four-character patterns into a dictionary.  
The **transmission scheme** works as follows:
  - ✓ Whenever we want to send a pattern that exists in the dictionary, we will send a **1-bit flag**, say, a **0**, followed by an **8-bit index** corresponding to the entry in the dictionary.
  - ✓ If the pattern is not in the dictionary, we will send a **1** followed by the **20-bit encoding** of the pattern.



## 5.2 Introduction

Example



- ✓ If the pattern we encounter is not in the dictionary, we will actually use more bits than in the original scheme, **21** instead of 20.
- ✓ But if it is in the dictionary, we will send only **9** bits.
- ✓ The utility of this scheme will **depend on the percentage** of the words we encounter that are in the dictionary.
- ✓ If the **probability** of encountering a pattern from the dictionary is  $p$ , then the **average number of bits per pattern  $R$**  is given by

$$R = 9p + 21(1 - p) = 21 - 12p.$$



## 5.2 Introduction

Example



- ✓ For this scheme to be useful,  $R$  should have a value less than 20.  
 $R = 21 - 12p < 20$
- ✓ This happens when  $p \geq 0.084$ .  
This does not seem like a very large number.  
However, note that if all patterns were occurring in an equally likely manner, the probability of encountering a pattern from the dictionary would be less than **0.00025 !!!**

$$256 / 1048576 = 0.000244140625.$$



## 5.2 Introduction

---

- ✓ We do not simply want a coding scheme that performs slightly better than the simple-minded approach of coding each pattern as equally likely;  
we would like to **improve the performance as much as possible**.
- ✓ In order for this to happen, **p should be as large as possible**.
  - ✓ This means that we should **carefully select patterns** that are most likely to occur as entries in the dictionary.
  - ✓ To do this, we have to have a **pretty good idea** about the structure of the source output.



## 5.2 Introduction

---

- ✓ If we do not have information of this sort available to us **prior to the encoding** of a particular source output, we need to acquire this information somehow **when we are encoding**.
- ✓ If we feel we have sufficient prior knowledge, we can use a **static approach**;  
if not, we can take an **adaptive approach**.



## 5.3 Static Dictionary

- ✓ Choosing a static dictionary technique is most appropriate when considerable **prior knowledge** about the source is available.
- ✓ This technique is especially suitable for use in **specific applications**.
  - ✓ To compress the student records at a university : "Name" , "Student ID" , "Sophomore" , "credits"...
- ✓ **Application-specific, or data-specific static-dictionary-based coding scheme**.
  - ✓ Work well only for the applications and data
- ✓ A static dictionary technique that is **less specific** to a single application is **digram coding**.



### 5.3.1 Digram Coding

- ✓ In **digram coding**, the dictionary consists of **all letters** of the source alphabet follow by as many **pairs of letters**, called **digram**, as can be accommodated by the dictionary.  
**Example:**
  - ✓ To construct a dictionary of size **256**
  - ✓ All printable ASCII characters : **95**
  - ✓ The remaining **161** entries would be the most frequently used pairs of characters.
- ✓ The digram encoder reads a two-character input and searches the dictionary to see if this input exists in the dictionary.



## 5.3.1 Digram Coding

### Encoding

- ✓ The digram encoder **reads a two-character input** and searches the dictionary to see if this input exists in the dictionary.
- ✓ If it does, the corresponding index is encoded and transmitted.
- ✓ If it does not, the first character of the pair is encoded
- ✓ The second character in the pair then becomes the first character of the next digram.
- ✓ The encoder reads another character to complete the digram, and the search procedure is repeated.



## 5.3.1 Digram Coding

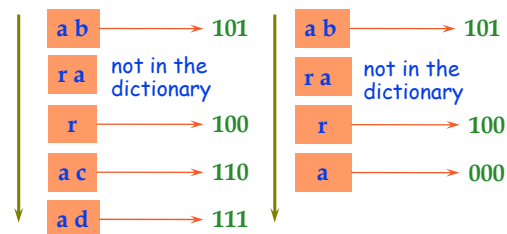
### Example 5.3.1

- ✓ Suppose we have a five-letter alphabet  $A = \{ a, b, c, d, r \}$ .
- ✓ Based on knowledge about the source, we build the dictionary shown in **Table 5.1**.

Code	Entry
000	a
001	b
010	c
011	d
100	r
101	ab
110	ac
111	ad

- ✓ Suppose we wish to encode the sequence:

**a b r a c a d a b r a**



**101 100 110 111 101 100 000**



## 5.3.1 Digram Coding

Table 5.2 Thirty most frequently occurring pairs of characters in a 41,364-character-long **Latex** document.

Pair	Count	Pair	Count	Pair	Count
ep	1128	en	392	dp	272
pt	838	on	385	po	266
pp	823	np	353	io	257
th	817	ti	322	co	256
he	712	pi	317	re	247
in	512	ar	314	p\$	246
sp	494	at	313	rp	239
er	433	pw	309	di	230
pa	425	te	296	ic	229
tp	401	ps	295	ct	226



## 5.3.1 Digram Coding

Table 5.3 Thirty most frequently occurring pairs of characters in a collection of **C programs** containing 64,983 characters.

Pair	Count	Pair	Count	Pair	Count
pp	5728	,p	554	or	374
nlp	1471	nl nl	506	rp	373
;nl	1133	pf	505	en	371
in	985	ep	500	er	358
nt	739	p*	444	ri	357
=p	687	st	442	at	352
pi	662	le	440	pr	351
tp	615	ut	440	te	349
p=	612	f (	416	an	348
);	558	ar	381	lo	347



## 5.4 Adaptive Dictionary

- ✓ Most **adaptive-dictionary-based techniques** have their roots in two landmark papers by **Jacob Ziv** and **Abraham Lempel** in 1977 and 1978.
- ✓ **LZ77 Family ( LZ1 )**
- ✓ **LZ78 Family ( LZ2 )**



### 5.4.1 The LZ77 Approach

- ✓ In the LZ77 approach, the **dictionary** is simply a portion of the previously encoded sequence.
- ✓ The **encoder** examines the input sequence through a **sliding window**.
- ✓ The window consists of two parts, a **search buffer** that contains a portion of the recently encoded sequence, and a **look-ahead buffer** that contains the next portion of the sequence to be encoded.



## 5.4.1 The LZ77 Approach

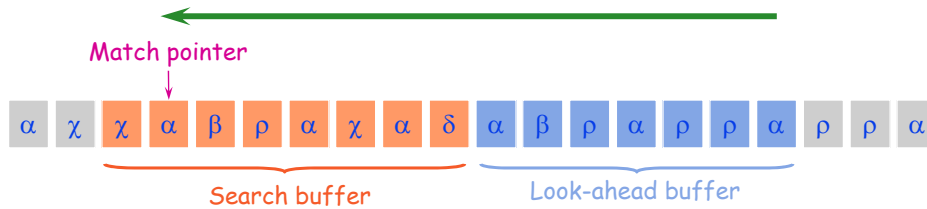


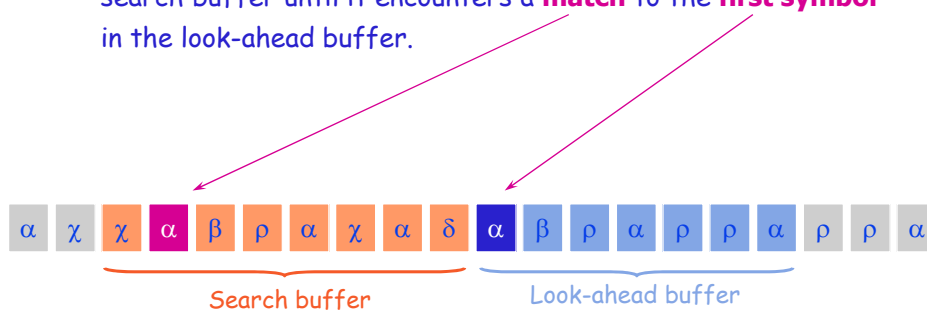
Figure 5.1 Encoding using LZ77 approach.

- ✓ In Figure 5.1, the search buffer contains 8 symbols, while the look-ahead buffer contains 7 symbols. In practice, the size of the buffers are significantly larger; however, for purpose of explanation, we will keep the buffer sizes small.



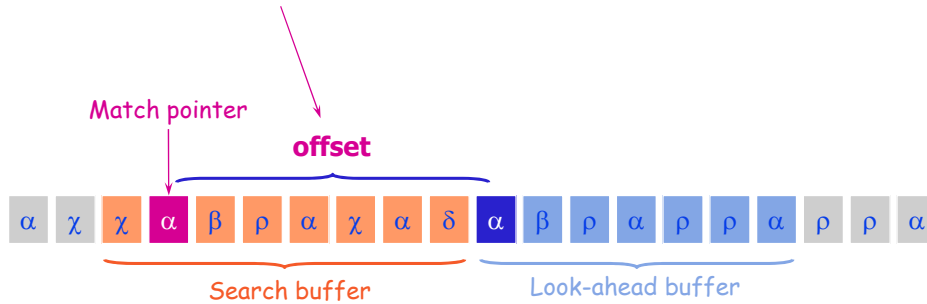
## 5.4.1 The LZ77 Approach

- ✓ To encode the sequence in the sequence in the look-ahead buffer, the encoder moves a search pointer back through the search buffer until it encounters a **match** to the **first symbol** in the look-ahead buffer.



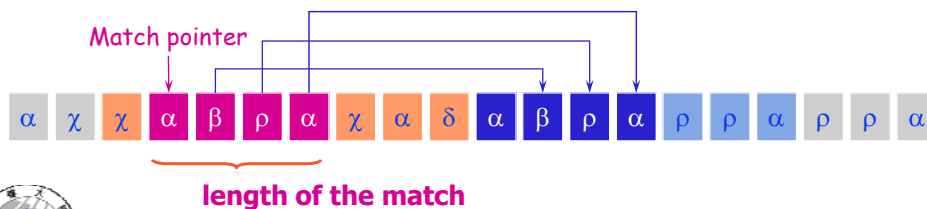
## 5.4.1 The LZ77 Approach

- ✓ The distance of the pointer from the look-ahead buffer is called the **offset**.



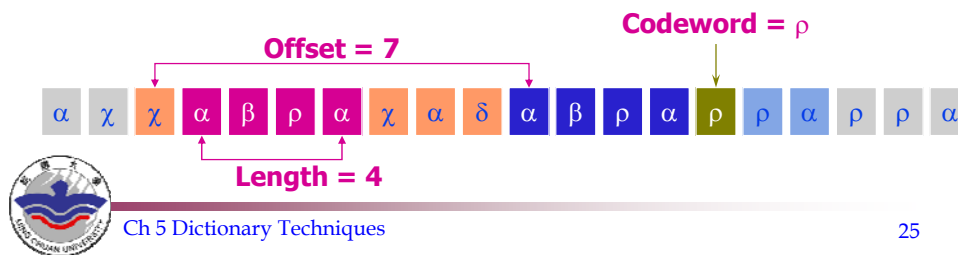
## 5.4.1 The LZ77 Approach

- ✓ The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer.
- ✓ The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called **the length of the match**.



## 5.4.1 The LZ77 Approach

- ✓ The encoder searches the **search buffer** for **the longest match**.
- ✓ Once the longest match has been found, the **encoder** encodes it with a **triple**  $\langle o, l, c \rangle$ ,  
where **o** is the offset,  
**l** is the length of the match,  
**c** is the **codeword** corresponding to the symbol in the look-ahead buffer that follows the match.



## 5.4.1 The LZ77 Approach

- ✓ The reason for sending the third element in the triple is to take care of the situation where **no match** for the symbol in the look-ahead buffer can be found in the search buffer.
- ✓ In this case,  $o = l = 0$ ,  $c =$  **the code for the symbol itself**.
- ✓ If the size of the search buffer is  $S$ ,  
the size of the window (search and look-ahead buffers) is  $W$ ,  
the size of the source alphabet is  $A$ ,  
then the number of bits needed to code the triple using fixed-length codes is  $\lceil \log_2 S \rceil + \lceil \log_2 W \rceil + \lceil \log_2 A \rceil$ .
- ✓ **Notice:** the second term is  $\lceil \log_2 W \rceil$ , **not**  $\lceil \log_2 S \rceil$ .  
**Reason:** the length of the match can actually exceed the length of the search buffer.



## 5.4.1 The LZ77 Approach

- ✓ In the following example, we will look at **three different possibilities** that may encountered during the coding process:
  - 1. There is no match for the next character to be encoded in the window.**
  - 2. There is a match.**
  - 3. The match string extends inside the look-ahead buffer.**



## 5.4.1 The LZ77 Approach

### Example 5.4.1 Encoding

- ✓ Suppose the sequence to be encoded is  
**... c a b r a c a d a b r a r r a d ...**
- ✓ Suppose the length of the window is **13**,  
the size of the look-ahead buffer is **6**,  
and the current condition is as follows:  

c	a	b	r	a	c	a	d	a	b	r	r	a	r	a	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

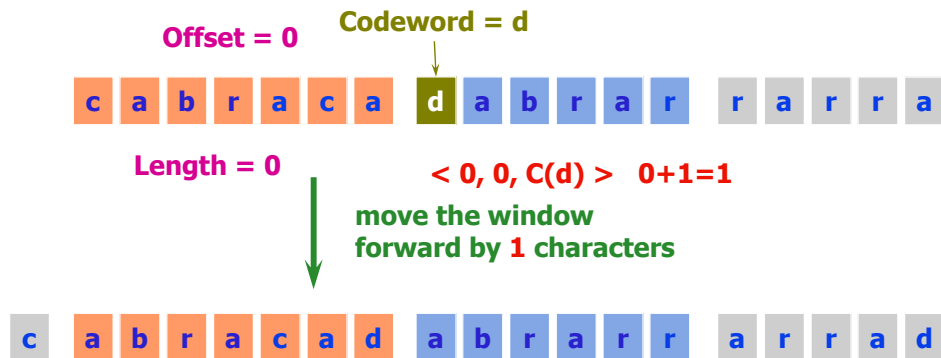
with **d a b r a r** in the look-ahead buffer.
- ✓ We look back in the encoded portion of the window to find a match for **d**.
- ✓ As we can see, there is no match, so we transmit the triple **< 0, 0, C(d) >**.



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

Encoding



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

Encoding

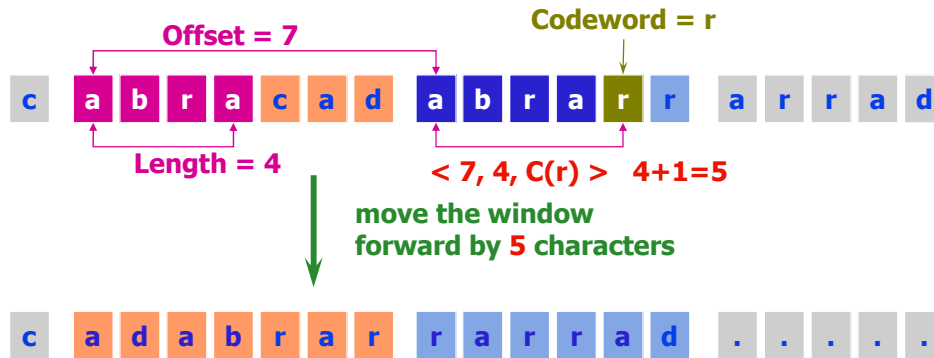
- ✓ Now, the contents of the buffer are  
a b r a c a d a b r a r r
- ✓ We find a match to **a** at an offset of **2**.  
The length of this match is **1**.
- ✓ Looking further back, we have another match for **a** at the offset of **4**; again the length of the match is **1**.
- ✓ Looking back even further in the window, we have a third match for **a** at the offset of **7**.  
However, this time the length of the match is **4**.
- ✓ So, we encode the string **a b r a** with the triple  $\langle 7, 4, C(r) \rangle$ , and **move the window forward by 5 characters**.



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

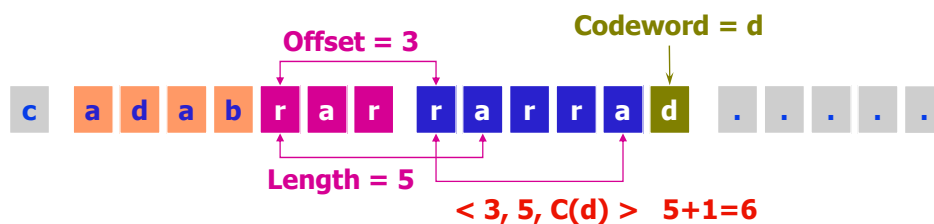
Encoding



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

Encoding



- ✓ Looking back in the window, we find a match to r at an offset of 1 and a match length of 1, and a second match at an offset of 3 with a match length of what at first appears to be 3.
- ✓ It turns out we can use a match length of 5 instead of 3.
- ✓ So, we encode the string rarrad with the triple  $\langle 3, 5, C(d) \rangle$ .



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

Decoding

c a b r a c a

- ✓ Let us assume that we have decoded the sequence **c a b r a c a** and we receive the triples  
 $\langle 0, 0, C(d) \rangle \langle 7, 4, C(r) \rangle \langle 3, 5, C(d) \rangle$ .
- ✓ The first triple is easy to decode; there was no match within the previous decoded string, and the next symbol is **d**.

c a b r a c a d

c a b r a c a d Search buffer



## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

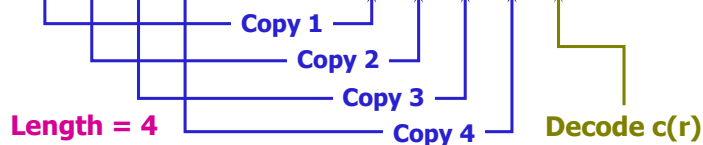
Decoding

$\langle 7, 4, C(r) \rangle$

Move back 7

c a b r a c a d

c a b r a c a d a b r a r



- ✓ The first element of the next triple tells the decoder to move the copy pointer back **7** characters, and copy **4** characters from that point. And then decodes the next symbol, i.e., **r**.

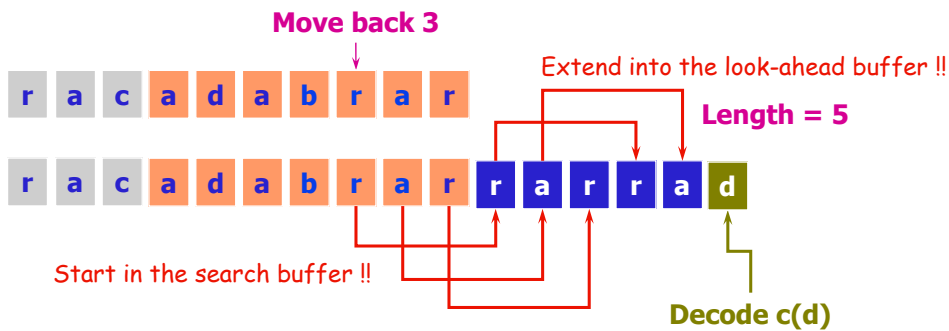


## 5.4.1 The LZ77 Approach

Example 5.4.1 (Conti.)

Decoding

< 3, 5, C(d) >



- ✓ We move back 3 characters, and copy 5 characters from that point. And then decodes the next symbol, i.e., d.



## 5.4.1 The LZ77 Approach

- ✓ **LZ77 scheme** is a very simple adaptive scheme :
  - ✓ requires no prior knowledge of the source
  - ✓ seems to require no assumptions about the characteristics of the source.
- ✓ The authors the LZ77 showed that asymptotically the **performance** of this algorithm **approached the best** that could be obtained by using a scheme that **had full knowledge** about the statistics of the source.
- ✓ By using the **recent portions** of the sequence, there is an **assumption** of sorts being used here - that is, that **patterns recur "close" together**.  
( **LZ78** removed this assumption )



## 5.4.1 The LZ77 Approach

### Variations on the LZ77 Theme

- ✓ A number of ways to make the **LZ77 scheme** more efficient:
  - 1. Triples** are encoded using **variable-length codes**.  
If we were willing to accept more complexity.  
**Adaptive, semi-adaptive (two-pass algorithm).**  
**PKZip, Zip, Lharc, PNG, gzip, and ARJ.**
  - 2. Varying the size of the search and look-ahead buffers.**  
To make the search buffer large requires the development of more effective search strategy.  
Such strategies can be implemented more effectively if the contents of the **search buffer** are stored in a manner conducive to fast searches.



## 5.4.1 The LZ77 Approach

### Variations on the LZ77 Theme (Conti.)

- 3. To eliminate the situation where we use a triple to encode a single character.**
  - ✓ Use of a triple is highly inefficient, especially if a large number of characters occurs infrequently.
  - ✓ The modification to get rid of this inefficiency is simply the addition of a **flag bit**, to indicate whether what follows is the codeword for a single symbol.
  - ✓ By using this flag bit we also get rid of the necessary for the **third element of the triple**.  
All we need to do is to send a pair of values corresponding to the offset and length of match.
  - ✓ This modification is referred to as **LZSS**.



## 5.4.2 The LZ78 Approach

- ✓ The **LZ77 scheme** implicitly assumes that like patterns will occur close together.
- ✓ It makes use of this structure by using the recent past of the sequence as the dictionary for encoding.
- ✓ However, this means that any pattern that recurs over a period longer than that covered by the coder window will not be captured.
- ✓ The **worst-case situation** would be where the sequence to be encoded was periodic with a period longer than the search buffer.



## 5.4.2 The LZ78 Approach

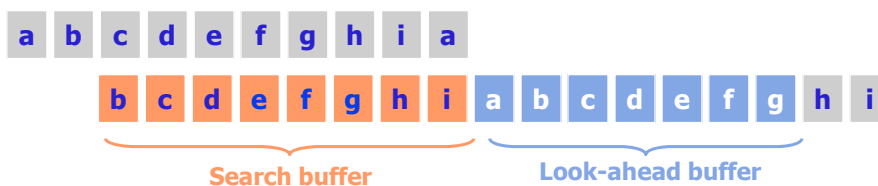


Figure 5.5 the Achilles heel of LZ77.

- ✓ This is a periodic sequence with a period of **9**.
- ✓ If the search buffer had been just one symbol longer, this sequence could have been significantly compressed.
- ✓ As it stands, none of the new symbols will have a match in the search buffer and will have to be represented by separate codewords.



## 5.4.2 The LZ78 Approach

- ✓ Although **Figure 5.5** is an extreme situation, there are less drastic circumstances in which **the finite view of the past would be a drawback**.
- ✓ The **LZ78 algorithm** solves this problem by
  1. **dropping the reliance on the search buffer** and
  2. **keeping an explicit dictionary**.
- ✓ This **dictionary**:
  1. **built at both the encoder and decoder,**
  2. **built in an identical manner.**



## 5.4.2 The LZ78 Approach

- ✓ The input are coded as a **double  $\langle i, c \rangle$** , with **i** being an **index** corresponding to the dictionary entry that was the longest match to the input, and **c** being the **code** for the character in the input following the matched portion of the input.
- ✓ As in the case of **LZ77**, the index value of **0** is used in the case of **no match**. This **double** then becomes the **newest entry** in the dictionary. Thus, each new entry into the dictionary is **one new symbol concatenated with an existing dictionary entry**.



## 5.4.2 The LZ78 Approach

### Example 5.4.2

- ✓ Let us encode the following sequence

**wabba** **bwabba** **bwabba** **bwabba** **bwwoo** **bwwoo** **bwwoo**

where **b** stands for space.

- ✓ Initially, the dictionary is empty, so the first few symbols encountered are encoded with the index value set to **0**.

The first three encoder outputs are

**<0, C(w)>**, **<0, C(a)>**, **<0, C(b)>**.

- ✓ The dictionary looks like **Table 5.4**.

#### The initial dictionary

<u>Index</u>	<u>entry</u>
<b>1</b>	<b>w</b>
<b>2</b>	<b>a</b>
<b>3</b>	<b>b</b>



## 5.4.2 The LZ78 Approach

### Example 5.4.2 (Conti.)

**wabba** **bwabba** **bwabba** **bwabba** **bwwoo** **bwwoo** **bwwoo**

- ✓ The fourth symbol is a **b**, which is the third entry in the dictionary.
  - ✓ If we append the next symbol, we would get the pattern **ba**, which is not in the dictionary,
  - ✓ so we encode these two symbols as **<3, C(a)>**, and
  - ✓ add the pattern **ba** as the fourth entry in the dictionary.



## 5.4.2 The LZ78 Approach

**TABLE 5.5** Development of dictionary.

Encoder	dictionary	Encoder	dictionary		
Output	Index	entry	Output	Index	entry
<0,C(w)>	1	w	<9,C(b)>	11	wabb
<0,C(a)>	2	a	<8,C(w)>	12	a <b>w</b>
<0,C(b)>	3	b	<0,C(o)>	13	o
<3,C(a)>	4	ba	<13,C( <b>b</b> )>	14	o <b>b</b>
<0,C( <b>b</b> )>	5	<b>b</b>	<1,C(o)>	15	wo
<1,C(a)>	6	wa	<14,C(w)>	16	o <b>w</b>
<3,C(b)>	7	bb	<13,C(o)>	17	oo
<2,C( <b>b</b> )>	8	a <b>b</b>	.	.	.
<6,C(b)>	9	wab	.	.	.
<4,C( <b>b</b> )>	10	ba <b>b</b>	.	.	.



wabba**w**abba**w**abba**w**abba**w**oo**w**oo**w**oo

45

## 5.4.2 The LZ78 Approach

### Example 5.4.2 (Conti.)

- ✓ **Notice** that the entries in the dictionary generally keep getting longer, and if this particular sentence was repeated often.

wabba**w**abba**w**abba**w**abba**w**oo**w**oo**w**oo

- ✓ As it in the song, after a while the entire sentence would be an entry in the dictionary.



## 5.4.2 The LZ78 Approach

- ✓ While the LZ78 algorithm has the ability to capture patterns and hold them indefinitely, it also has a rather serious **drawback**.  
**The dictionary keeps growing without bound.**
- ✓ In a practice situation, we would have to
  - ✓ **stop** the growth of the dictionary at some stage, and then
  - ✓ either **prune it back** or **treat the encoding as a fixed dictionary scheme.**



## 5.4.2 The LZ78 Approach

### Variations on the LZ78 Theme – The LZW algorithm

- ✓ The most well-known modification is proposed by **Terry Welch** known as **LZW**.
- ✓ **Welch** proposed a technique for removing the necessity of encoding the second element of the pair **< i, c >**.  
That is, the encoder would only send the index of the dictionary.
- ✓ In order to do this, the **dictionary** has to be primed with all the letters of all the source alphabet.



## 5.4.2 The LZ78 Approach

### The LZW algorithm (Conti.)

- ✓ The **input** of the encoder is **accumulated** in a pattern **p** as long as **p** is contained in the dictionary.
- ✓ If the addition of another letter **a** results in a pattern **p\*a** (\* denotes concatenation) that is not in the dictionary, then
  1. the index of **p** is transmitted to the receiver,
  2. the pattern **p\*a** is added to the dictionary, and we
  3. start another pattern with the letter **a**.



## 5.4.2 The LZ78 Approach

### Example 5.4.3

### Encoding

- ✓ Let us use the same sequence  
**wabba****bwabba****bwabba****bwabba****bw****woo****bw****woo****bw****woo**  
 Assuming that the alphabet for the source is { **bwawo** },  
 the **LZW dictionary** initially looks like **Table 5.6**.

**w** : in the dictionary  
**wa** : **not** in the dictionary  
       add **wa** to the dictionary  
       send **5**  
**a** : in the dictionary  
**ab** : **not** in the dictionary  
       add **ab** to the dictionary  
       send **2**

The initial dictionary	
Index	entry
1	<b>b</b>
2	<b>a</b>
3	<b>w</b>
4	<b>o</b>
5	<b>wo</b>



wabba**ʔ**wabba**ʔ**wabba**ʔ**wabba**ʔ**woo**ʔ**woo**ʔ**woo

## 5.4.2 The LZ78 Approach

**The LZW dictionary**

Index	entry
1	<b>ʔ</b>
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb
9	ba
10	a <b>ʔ</b>
11	<b>ʔ</b> w
12	w...

**Table 5.7**

**b** : in the dictionary  
**bb** : **not** in the dictionary  
 add **bb** to the dictionary, send **3**  
**b** : in the dictionary  
**ba** : **not** in the dictionary  
 add **ba** to the dictionary, send **3**  
**a** : in the dictionary  
**aʔ** : **not** in the dictionary  
 add **aʔ** to the dictionary, send **2**  
**ʔ** : in the dictionary  
**ʔw** : **not** in the dictionary  
 add **ʔw** to the dictionary, send **1**  
**w** : in the dictionary  
 ...



wabba**ʔ**wabba**ʔ**wabba**ʔ**wabba**ʔ**woo**ʔ**woo**ʔ**woo

## 5.4.2 The LZ78 Approach

**Table 5.8**

**The LZW dictionary**

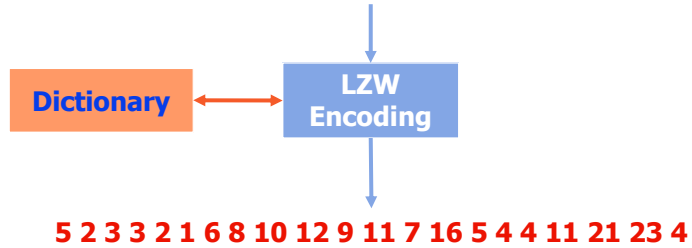
Index	entry	Index	entry
1	<b>ʔ</b>	14	a <b>ʔ</b> w
2	a	15	wabb
3	b	16	ba <b>ʔ</b>
4	o	17	<b>ʔ</b> wa
5	w	18	abb
6	wa	19	ba <b>ʔ</b> w
7	ab	20	wo
8	bb	21	oo
9	ba	22	o <b>ʔ</b>
10	a <b>ʔ</b>	23	<b>ʔ</b> wo
11	<b>ʔ</b> w	24	oo <b>ʔ</b>
12	wab	25	<b>ʔ</b> woo
13	bba		



## 5.4.2 The LZ78 Approach

### Example 5.4.3 (Conti.)

wabba**bw**wabba**bw**wabba**bw**wabba**bw**oo**bw**oo**bw**oo

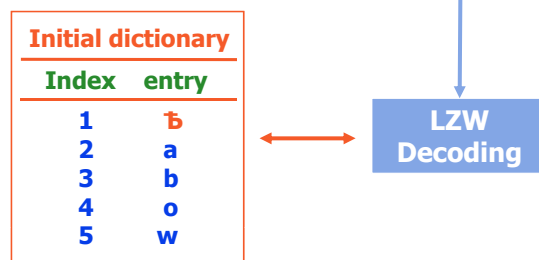


## 5.4.2 The LZ78 Approach

### Example 5.4.4

### Decoding

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

Initial dictionary	
Index	entry
1	␣
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb

- 5 → w, in the dictionary (buffer w)  
 2 → a, (buffer wa) not in the dictionary, add wa to dictionary (index 6) (buffer a) in the dictionary  
 3 → b, (buffer ab) not in the dictionary, add ab to dictionary (index 7) (buffer b) in the dictionary  
 3 → b, (buffer bb) not in the dictionary, add bb to dictionary (index 8) (buffer b) in the dictionary



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

LZW dictionary	
Index	entry
1	␣
2	a
3	b
4	o
5	w
6	wa
7	ab
8	bb
9	ba
10	a␣
11	␣w

- 2 → a, (buffer ba) not in the dictionary, add ba to dictionary (index 9) (buffer a) in the dictionary  
 1 → ␣, (buffer a␣) not in the dictionary, add a␣ to dictionary (index 10) (buffer ␣) in the dictionary  
 6 → wa, (buffer ␣w) not in the dictionary, add ␣w to dictionary (index 11) (buffer wa) in the dictionary, so we decode the next input, which is 8



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

LZW dictionary			
Index	entry	Index	entry
1	␣	12	wab
2	a	13	bba
3	b	14	a␣w
4	o		
5	w		
6	wa		
7	ab		
8	bb		
9	ba		
10	a␣		
11	␣w		

8 → bb,

(buffer wab) not in the dictionary,  
add wab to dictionary (index 12)

(buffer bb) in the dictionary

10 → a␣,

(buffer bba) not in the dictionary,  
add bba to dictionary (index 13)

(buffer a␣) in the dictionary

12 → wab,

(buffer a␣w) not in the dictionary,  
add a␣w to dictionary (index 14)

(buffer wab) in the dictionary



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

LZW dictionary			
Index	entry	Index	entry
1	␣	12	wab
2	a	13	bba
3	b	14	a␣w
4	o	15	wabb
5	w	16	ba␣
6	wa	17	␣wa
7	ab		
8	bb		
9	ba		
10	a␣		
11	␣w		

9 → ba,

(buffer wabb) not in the dictionary,  
add wabb to dictionary (index 15)

(buffer ba) in the dictionary

11 → ␣w,

(buffer ba␣) not in the dictionary,  
add ba␣ to dictionary (index 16)

(buffer ␣w) in the dictionary

7 → ab,

(buffer ␣wa) not in the dictionary,  
add ␣wa to dictionary (index 17)

(buffer ab) in the dictionary



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

LZW dictionary			
Index	entry	Index	entry
1	␣	12	wab
2	a	13	bba
3	b	14	a␣w
4	o	15	wabb
5	w	16	ba␣
6	wa	17	␣wa
7	ab	18	aab
8	bb	19	ba␣w
9	ba	20	wo
10	a␣		
11	␣w		

16 → ba␣,

(buffer abb) not in the dictionary,  
add abb to dictionary (index 18)

(buffer ba␣) in the dictionary

5 → w,

(buffer ba␣w) not in the dictionary,  
add ba␣w to dictionary (index 19)

(buffer w) in the dictionary

4 → o,

(buffer wo) not in the dictionary,  
add wo to dictionary (index 20)

(buffer o) in the dictionary



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

LZW dictionary			
Index	entry	Index	entry
1	␣	14	a␣w
2	a	15	wabb
3	b	16	ba␣
4	o	17	␣wa
5	w	18	aab
6	wa	19	ba␣w
7	ab	20	wo
8	bb	21	oo
9	ba	22	o␣
10	a␣	23	␣wo
11	␣w		
12	wab		
13	bba		

4 → o,

(buffer oo) not in the dictionary,  
add oo to dictionary (index 21)

(buffer o) in the dictionary

11 → ␣w,

(buffer o␣) not in the dictionary,  
add o␣ to dictionary (index 22)

(buffer ␣w) in the dictionary

21 → oo,

(buffer ␣wo) not in the dictionary,  
add ␣wo to dictionary (index 23)

(buffer oo) in the dictionary



## 5.4.2 The LZ78 Approach

### Example 5.4.4

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 **23 4**

LZW dictionary			
Index	entry	Index	entry
1	␣	14	a␣w
2	a	15	wabb
3	b	16	ba␣
4	o	17	␣wa
5	w	18	aab
6	wa	19	ba␣w
7	ab	20	wo
8	bb	21	oo
9	ba	22	o␣
10	a␣	23	␣wo
11	␣w	24	oo␣
12	wab	25	␣woo
13	bba		

**23** → ␣wo ,

(buffer oo␣) not in the dictionary,  
add oo␣ to dictionary (**index 24**)

(buffer ␣wo) in the dictionary

**4** → o ,

(buffer ␣woo) not in the dictionary,  
add ␣woo to dictionary (**index 25**)

(buffer o) in the dictionary

**Notice that the dictionary being constructed by the decoder is identical to that constructed by the encoder.**

61

## 5.4.2 The LZ78 Approach

- ✓ Suppose we had a source with an alphabet  $A = \{ a b \}$ , and we were to encode the sequence beginning with **ababababababa....**
- ✓ The encoding process is still the same.
- ✓ We begin with the initial dictionary shown in **Table 5.10** and end up with the final dictionary shown in **Table 5.11**.
- ✓ The transmitted sequence is **1 2 3 5 ...**

**Table 5.10**

Initial dictionary	
Index	Entry
1	a
2	b

**Table 5.11**

Final dictionary for abababaab.			
Index	Entry	Index	Entry
1	a	5	aba
2	b	6	abab
3	ab	7	b...
4	ba		



## 5.4.2 The LZ78 Approach

1 2 3 5 4  
 a b a b a b a b a b a ...  
 { 3 5 7 }  
 { 4 6 }



## 5.4.2 The LZ78 Approach

✓ The transmitted sequence is **1 2 3 5 ...**

Initial dictionary	
Index	Entry
1	a
2	b
3	ab
4	ba
5	buffer ab
6	

- 1 → a,  
(buffer a) in the dictionary
- 2 → b,  
(buffer ab) **not** in the dictionary,  
add **ab** to dictionary (**index 3**)  
(buffer b) in the dictionary.
- 3 → ab,  
(buffer ba) **not** in the dictionary,  
add **ba** to dictionary (**index 4**)  
(buffer ab) in the dictionary.
- 5 → ??? **Incomplete entry ???**



## 5.4.2 The LZ78 Approach

- ✓ This situation is actually not as bad as it looks.  
( Of course, if it were, we could not now be studying LZW. )
- ✓ While we may not have a **fifth entry** for the dictionary, we do have the beginnings of the **fifth entry**, which is **ab...**
- ✓ Let us **pretend** that we do indeed have the **fifth entry** and **continue with the decoding process**.
- ✓ If we had a **fifth entry**, the first two letter would be **a** and **b**. Concatenating **a** to the partial new entry, we get the pattern **aba**.
- ✓ This pattern is not contained in the dictionary, so we add this to our dictionary, which is **aba (index 5)**, which now look like like **Table 5.15**.



## 5.4.2 The LZ78 Approach

Table 5.15

Initial dictionary	
Index	Entry
1	a
2	b
3	ab
4	ba
5	aba
6	buffer a..

5 → aba  
(buffer a..) in the dictionary.

Continue on our merry way !!



## 5.4.2 The LZ78 Approach

Initial dictionary	
Index	Entry
1	a
2	b
3	ab
4	ba
5	aba
6	abab

4 → ba  
(buffer **abab**) **not** in the dictionary.  
add **abab** to dictionary (**index 6**)

This means that  
the **LZW decoder** has to contain an  
**exception handler**  
to handle the special case of decoding  
an index that does **not** have a  
corresponding **complete entry**  
in the decoding dictionary.



## 5.5 Application

- ✓ Since the publication of **Terry Welch's** article, there has been a steadily increasing number of applications that use some variant of the **LZ78 algorithm**. Among the LZ78 variants, by far the most popular is the LZW algorithm.
- ✓ Lately, there has been renewed interest in the **LZ77 approach** and its variants as well.
- ✓ However, for now, the LZ78 approach, or more specifically, the **LZW algorithm**, is one of the most widely used compression algorithms.
- ✓ **LZW: UNIX compress, GIF, and V.42 bis.**



## 5.5.1 File Compression – UNIX compress

- ✓ **UNIX** `compress` command is one of the earlier application of **LZW**.
- ✓ The **size** of the dictionary is **adaptive**.  
We start with a dictionary of size **512**.  
This means that the transmitted codewords are **9** bits long.
- ✓ Once the dictionary has filled up, the size of the dictionary is doubled to **1024** entries.  
The codeword transmitted at this point have 10 bits.
- ✓ The size of the dictionary is progressively doubled as it filled up.  
The **maximum size** of the codeword,  $b_{\max}$ , can be set by the user to between **9** and **16**, with **16** bits being the default.



## 5.5.1 File Compression – UNIX compress

- ✓ Once the dictionary contains  $2^{b_{\max}}$  entries, `compress` becomes a **static dictionary** coding technique.
- ✓ At this point the algorithm monitors the compression ratio.  
If the **compression ratio** falls below a **threshold**,  
the dictionary is flushed, and  
the dictionary building process is restarted.  
This way, the dictionary always **reflects the local characteristics** of the source.



## 5.5.2 Image Compression - GIF

- ✓ **Graphics Interchange Format (GIF)** was developed by **Compuserve Information Service** to encode graphical images.
- ✓ It is another implementation of the **LZW algorithm** and is very similar to the `compress` command.
- ✓ The compressed image is stored with the first byte (being the minimum number of bits **b** per pixel ) in the original image ( **b = 8** ).
- ✓ The binary number  $2^b$  is defined to be the **clear code**.  
This code is used to reset all compression and decompression parameters to a startup state.



## 5.5.2 Image Compression - GIF

### Dictionary

- ✓ The initial size of the dictionary is  $2^{b+1}$ . ( $2^9=512$ )  
When this fills up, the dictionary size is double,  
as was done in the `compress` algorithm,  
until the maximum dictionary size of **4096** is reached. ( $2^{12}$ )
- ✓ At this point the compression algorithm behaves like a **static dictionary algorithm**.



## 5.5.2 Image Compression - GIF

### Codeword

- ✓ The **codeword** from the **LZW algorithm** are stored in **blocks of characters**. The characters are **8 bits** long, and the maximum block size is **255**.
- ✓ Each block is preceded by a **header** that contains the **block size**.
- ✓ The block is terminated by a **block terminator** consisting of **8 0s**.
- ✓ The end of the compressed image is denoted by an **end-of-information code** with a value of  **$2^b+1$  . ( 257 )**  
This codeword should appear before the block terminator.



## 5.5.2 Image Compression - GIF

**Table 5.15 Comparison of GIF with arithmetic coding.**

Image	GIF	Arithmetic Coding of Pixel Values	Arithmetic Coding of Pixel Differences
Sene	51,085	53,431	31,847
Sensin	60,649	58,306	37,126
Earth	34,276	38,246	32,137
Omaha	61,580	56,061	51,393

