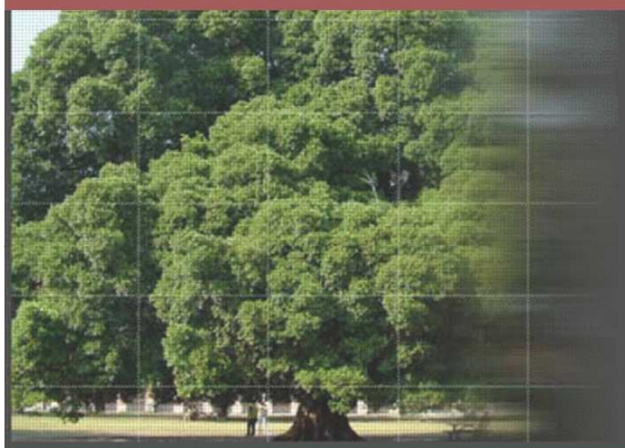


字典基礎模式

第6章

字典基礎模式 (*dictionary-based model*)

- 這類的演算法並不是使用不同長度的位元串來編碼每一個符號，也不需要預先估計出每個符號的出現機率，而是對於不同長度的符號串都用同一種記號(*token*) 來表示。

- 假設要編碼 "*Data Compression*" 這兩個英文字。查了字典後發現 "*Data*" 出現在牛津辭典的第271頁的第13個字，而 "*Compression*" 則出現在第213頁的第8個字，於是就可以用

  (271, 13) (213, 8)

  這兩個記號來表示 "*Data Compression*"。

字典基礎模式 (*dictionary-based model*)

- 解碼端也必須有同樣的一部字典。收到這兩個記號後，解碼端只需要查字典便可以將 "*Data Compression*" 解碼出來。

- 我們所使用的字典有1,354頁，每一頁裡最多不超過64個字。因此每一個記號可以用11個位元來表示頁數，6個位元來表示第幾個字，總共17個位元。

- 如果原來使用的是8個位元的*ASCII*碼，則 "*Data Compression*" 總共需要用8×16＝128個位元來表示。因此利用這個方法壓縮 "*Data Compression*" 所得到的資料壓縮比為128/34＝3.765倍。

靜態與適應性

- 在壓縮前先建好而在壓縮中不改變的字典，就如同靜態的統計模式無失真資料壓縮法之統計資料。
- 最大的優點是我們可以針對要壓縮的資料設計字典。
- 以汽車商標資料庫來說，我們可以使用較少的位元來表示 "*Ford*" 而使用較多的位元來表示 "*YUGO*"。

靜態與適應性

- 演算法：適應性字典基礎法模式編碼

  第一步：讀入下一個片語；

  第二步：從字典中找出片語之位置；

  第三步：如果從字典中找到該片語，則送出該片語於字典中的位置，否則以原始碼送出該片語並將該片語加入字典中；

  第四步：回到第一步，直到編碼結束為止

靜態與適應性

- 由上面的演算法可以看出一個適應性字典基礎模式無失真資料壓縮法含有以下幾個基本功能：
  1. 將輸入符號串分割成片語並與字典做比對，
  2. 比對片語與字典時，如果比對只達到部分的匹配 (*match*)，應該做適當的處理，
  3. 加入新片語至字典中，
  4. 編碼記號 (從字典中找到該片語) 與不經編碼之純文字 (字典中找不到匹配的片語) 必須可區分。

靜態與適應性

- 解碼端所需要做的工作只是：

(1) 將輸入的符號串分別解讀成字典位置 (指標) 或未經編碼之純文字，

(2) 將新片語加入字典中，

(3) 將字典位置轉換成片語，

(4) 將片語輸出。

Lempel Ziv 壓縮法

- *LZ77*
  - 屬於“滑動視窗”技術，意思是它的字典是由前文透過固定長度之視窗所看到之片語共同組成。
  - 視窗的大小通常介於2到16 *Kbytes*
  - 片語的最大允許長度則從16個*bytes*到64個*bytes*都有。

- *LZ78*則採取完全不同的方法來建立字典內容，不採用前文透過固定長度之視窗所看到的片語集，而是每次加一個符號以組成新片語，即當匹配到一個片語時，再加一個符號形成新的、更長的片語。

- 一般人都將它們統稱為“*Lempel Ziv*壓縮法” 或“*LZ*系列壓縮法”

滑動視窗壓縮法 (Sliding window compression)

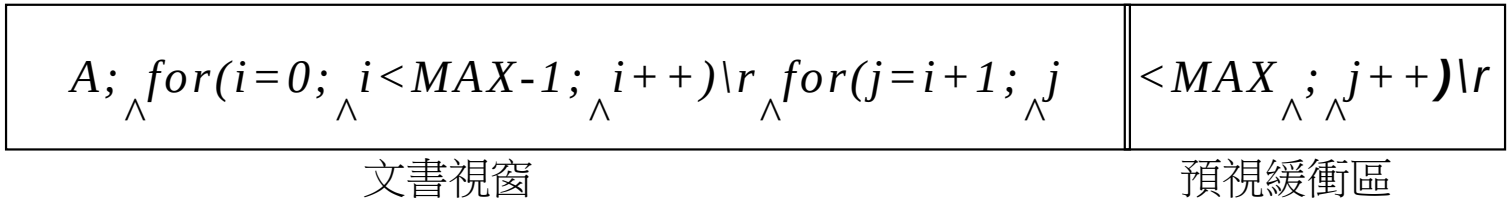
- LZ77用前面剛編碼過的輸入做字典，並用一個指標指向字典中片語的位置來取代輸入片語，以達到資料壓縮。
- 資料壓縮的效能則決定於匹配到的片語之長短、窺視前文之視窗的大小、以及對應於*LZ77*模式之符號源的熵。
- *LZ77*主要的資料結構是文書視窗與預視緩衝區，前者的內容是一大塊剛剛已經編碼完的輸入，做為*LZ77*的適應性字典；後者的預視緩衝區通常比前者小許多，存放的是現在才從輸入串讀進來而尚未編碼的一段文字。

滑動視窗壓縮法 (Sliding window compression)

- *LZ77*會產生一連串的記號，每一個記號利用三個欄位值以定義出文書視窗中一個片語的位置及長度，這三個欄位依序為：
  (1) 文書視窗內一個片語的起始位置
  (2) 片語之長度
  (3) 在預視緩衝器內緊接在該片語之後的符號。

滑動視窗壓縮法 (Sliding window compression)

- 例：

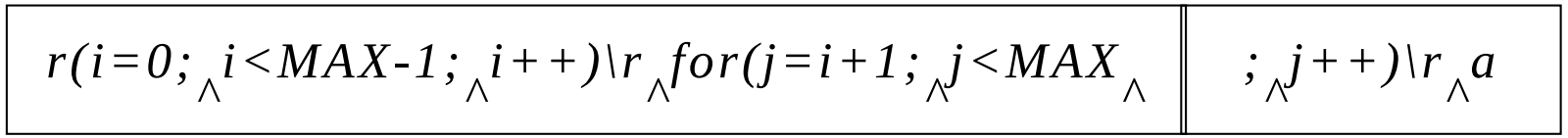


- 以上例子而言，預視緩衝區的內容為 “<MAX^;^j++”。從文書視窗內做搜尋，我們發現匹配到的片語為 “<MAX”，其長度為4。

- 預視緩衝區內緊跟在片語 “<MAX” 之後的是空白，因此LZ77使用記號 (14,4,`^’) 來編碼 “<MAX^”，共五個字元。

滑動視窗壓縮法 (Sliding window compression)

- *LZ77*首先輸出記號，然後文書視窗及預視緩衝區各往右移五個位置，如圖所示。



- 下一個匹配到的片語是 “;^j”，而預視緩衝區中緊跟其後的是 “＋” ，因此下一個送出的記號是 (33,3, ‘+’)。

- 這種表示法也可適用於找不到匹配的情況。我們可以設定前兩個欄位為0，表示匹配到的片語長度為0。這種方法當然效率不夠，我們等於使用三個欄位來編碼一個符號，但是至少它行得通。

滑動視窗壓縮法 (Sliding window compression)

- *LZ77*的解碼就更簡單了，因為它不需要做比對的工作，只是重覆地做著以下的工作：讀入記號，輸出記號所指定的片語及緊接在後面的符號，然後移位。

- *LZ77*有一個很有趣的現象：我們可以用預視緩衝區內的符號來編碼自己。

滑動視窗壓縮法 (Sliding window compression)

- 例：

?????.....???*ABC*	*ABCABCABCD*

其中 “?” 表示*A*、*B*、*C*以外的符號。

假設文書視窗的大小為48，則預視緩衝區內之 “*ABCABCABCABCD*” 可用(46,12,‘*D*’) 來表示。

解碼端收到記號時，先從文書視窗的第46、47、48位置找到片語的前三個符號為*ABC*。

此時文書視窗已到盡頭而片語還有九個符號未定，於是文書視窗恰似延伸到剛解碼出的符號*ABC*為文書視窗之第49、50、51個符號，又解碼出一個*ABC*。

滑動視窗壓縮法 (Sliding window compression)

- 這個例子也突顯出*LZ77*的高度適應性。如果圖中的“?”是空白，而文書視窗內只含*ABC*三個符號，恰似*LZ77*剛開始做壓縮最先碰到的三個符號。結果它只讀入三個符號卻能編碼12個符號。

- *LZ77*很顯然地會有個瓶頸：在做編碼時，它得為預視緩衝區內的字串與文書視窗內的每一個位置做比對的工作。如果它想藉由加大文書視窗以改進壓縮效率，這個瓶頸只會更糟糕。

滑動視窗壓縮法 (Sliding window compression)

- 除此之外，*LZ77*還有另外一個效率上的問題，即文書視窗處理的問題。
  - 比較好的方法是採用環狀佇列，否則文書視窗在滑行時便得額外做許多次移位的動作。

- *LZ77*還有一個主要的效率上的問題，如果在字典中找不到匹配，*LZ77*還是得用三個欄位的記號 (0,0,*c*)來編碼單一一個符號*c*。

滑動視窗壓縮法 (Sliding window compression)

- *LZSS*試圖避免*LZ77*所發生的一些瓶頸及效能問題，其主要改變有二：
  - 文書視窗的處理方式：

    在*LZ77*，文書視窗裡的片語是一個連續的文字區段，除此之外沒有任何其他組織存在。

    *LZSS*仍然將文書 (片語) 存放於連續視窗裡，不同的是它額外產生一個可以改善片語組織的資料結構。

滑動視窗壓縮法 (Sliding window compression)

- 三欄位記號的使用。

  *LZ77*使用的記號包含片語於文書視窗的位置、片語長度、及緊跟在片語後面的符號共三個欄位。如果在字典中找不到匹配時，*LZ77*得為一個符號額外多送兩個指標。

  *LZSS*允許另一種形式的記號。程式一開始執行時，可能接連好幾個符號都在字典中找不到匹配，*LZ77*在這種情況下送出(0,0,*c*)，仍然是三個欄位的記號。*LZSS*採取兩種記號，每種記號的前面各有一個位元，根據這個位元是0或1，分別表示後面跟著的是一個位置/長度的記號或者單一的符號。

滑動視窗壓縮法 (Sliding window compression)

- *LZSS*使用了兩個重要的資料結構：
  - 第一個是文書視窗：

    文書視窗一般就是一個一維矩陣

    unsigned char window [*WINDOW_SIZE*];

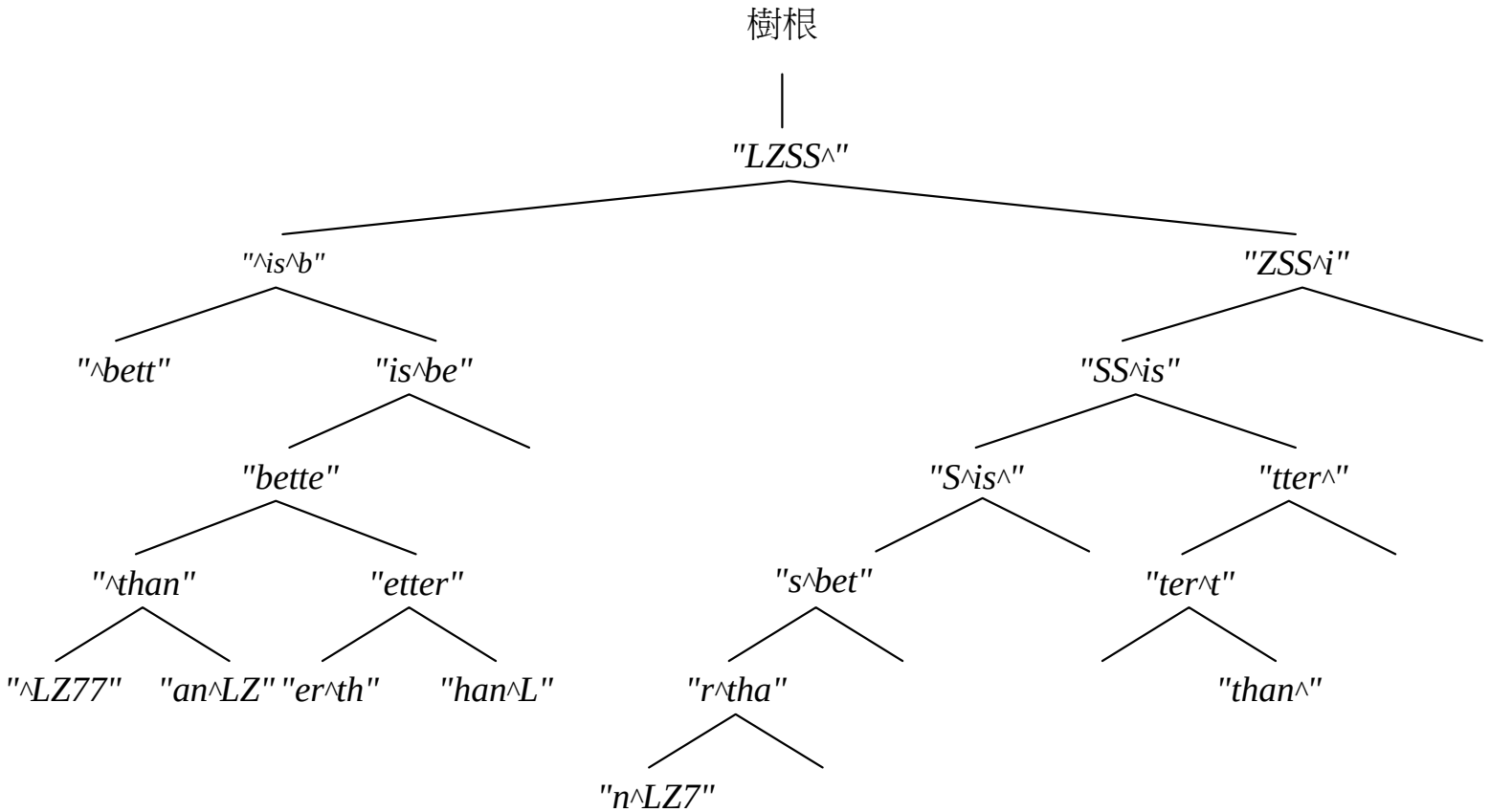
    如果*WINDOW_SIZE*的值是2的整數次幕，則 (*i*＋1) *mod* *WINDOW_SIZE*的計算會變得有效率許多。

    另外一個好處是可以達到最有效率的視窗位置編碼，即2^n個*n*位元組合都完全用到。

滑動視窗壓縮法 (Sliding window compression)

- 第二個資料結構即二元搜尋樹。

  例：“*LZSS is better than LZ77*” 之二元搜尋樹



滑動視窗壓縮法 (Sliding window compression)

- *LZ77*與*LZSS*兩者都屬於 “貪婪” 演算法 (*greedy algorithm*)，它們並不對全程做一個最佳化的匹配安排。
- 例：

字典中含有

Go^T
o^S
tat
^Stat

要編碼的

GO^TO^Statement

貪婪的編碼器	
“*Go^T*”	：25個位元
“*o^S*”	：25個位元
“*tat*”	：25個位元
	：75個位元

最佳化做法	
“*Go^T*”	：25個位元
“*o*”	： 9個位元
“*^Stat*”	：25個位元
	：59個位元

滑動視窗壓縮法 (Sliding window compression)

- 在資料壓縮這個領域，一些好的經驗法則 (*heuristics*) 常常比一個可以被證明為最佳化的演算法來得受尊敬。

- 貪婪演算法就是一個經驗法則，它不保證可以得到最佳解，但是它所求得的解通常都不錯，甚至於偶爾它還可以碰上最佳解。

LZ78與LZW

- 滑行視窗的理念使得*LZ77*(*LZSS*) 偏重於最近出現的文字。大部分的資料看起來和最近剛出現的資料較接近，而與出現已久的資料相差較多。

- 另外一個使*LZ77*效能不理想的原因是它所能匹配到的片語長度受限於預視緩衝區的大小。

LZ78與LZW

- *LZ78*捨棄了文書視窗的觀念。同樣是由片語所構成的字典，*LZ77*是藉由文書視窗所看到的前面剛讀過的文章來定義，而*LZ78*則可以無限制地由前面所讀過的所有片語來組成。

- 和*LZ77*不同的另一點是*LZ78*並不使用一個充滿文章的文書視窗做為字典。每當它輸出一個記號，它就新產生一個片語並且將此片語放入字典中。在新片語加入字典之後，它就可以永遠為編碼端所使用。

LZ78與LZW

- *LZ78*與*LZ77*在某些方面相類似。
  - *LZ77*的輸出記號含三個欄位：片語位置、片語長度、及緊跟在片語之後的符號。
  - *LZ78*也輸出具有相同意義的記號，每一個*LZ78*的記號包括代表一個片語的碼及緊跟在該片語之後的符號。
  - 和*LZ77*不同的是，我們不需要送片語的長度給解碼端，因為解碼端知道。

LZ78與LZW

- 演算法：　*LZ78*

  第一步：讀入第一個符號為起始片語；設定$n \leftarrow 1$；

  第二步：從字典中找匹配之片語，若找到則跳到第三步；
  否則送出前$n-1$個符號所構成之片語的編號及
  第n個符號；
  將n個符號所構成之新片語加入字典中；
  回到第一步；

  第三步：讀入下一個符號；
  $n \leftarrow n+1$；
  回到第二步；

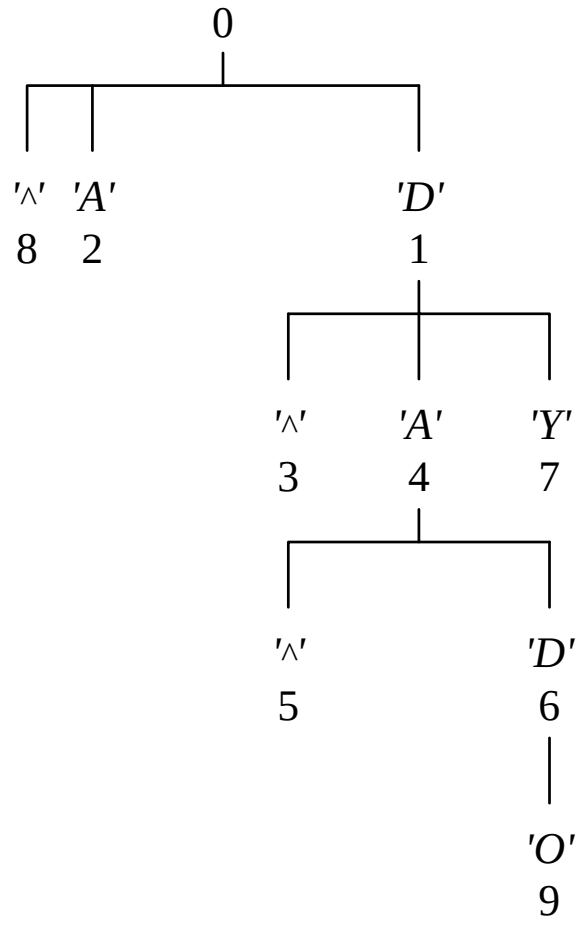
LZ78與LZW

- 例：輸入 “*DAD^DADA^DADDY^DADO …*”

輸出片語編號	輸出字母	新加入字典	新片語之編號
0	‘ D ‘	“D”	(1)
0	‘ A ‘	“A”	(2)
1	‘ ‘	“D^”	(3)
1	‘ A ‘	“DA”	(4)
4	‘ ‘	“DA^”	(5)
4	‘ D ‘	“DAD”	(6)
1	‘ Y ‘	“DY“	(7)
0	‘ ‘	“^“	(8)
6	‘ O ‘	“DADO”	(9)

LZ78與LZW

讀了19個字母後字典樹的樣子：



LZ78與LZW

- 一般這些片語是以多重路徑搜尋樹 (*multiway search tree*) 來表示。樹的樹根編號為0，代表空字串。每一個可以加在空字串之後的符號形成樹的一枝新分枝。而每個節點有一個編號，代表一個片語。

- 我們只需要沿著樹走，根據片語的每一個字母選擇該走那一個分枝，直到片語最後停在某一個特定的節點，我們就找到了該片語的匹配，該節點上的編號即此片語在字典中的編號。

LZ78與LZW

- *LZ78*有一個負面的效果是*LZ77*所沒有的，即解碼端也得維持同樣的字典樹。

- 無論我們給字典多大的空間，遲早它總會填滿，當它被填滿時，我們該怎麼處置？
  - *UNIX*所使用的壓縮程式是*LZ78*的一個變型，它監視檔案之壓縮比，以決定如何處理字典填滿的問題。如果壓縮比開始萎縮，整個字典便會被除掉，然後從起跑線重新開始。
  - 否則的話，儘管不再有新片語加入字典，它還是會繼續地被使用

LZ78與LZW

- *LZW*最大的改變是一開始它就先把符號源內所有的符號放到字典中，成為單一符號之片語。如此一來，即使一個符號是第一次出現，它也可以在字典中找到匹配的片語。

- *LZW*的輸出只是片語的編號，而不輸出緊跟在該片語之後的單一符號。和*LZ78*一樣，在輸出片語編號之後，*LZW*將該片語加上緊跟在後面的單一符號一起視為一個新片語，並且將它加入字典中。

LZ78與LZW

- 例：輸入字串 “^WED^WE^WEE^WEB^WET...”

輸入之符號	輸出碼	新片語及其編號
“W”	‘^‘	256=“^W”
“E “	‘W’	257=“WE”
“D“	‘E’	258=“ED”
“^“	‘D’	259=“D^”
“WE”	256	260=“^WE”
“^“	‘E’	261=“E^“
“WEE”	260	262=“^WEE”
“^W”	261	263=“E^W”
“EB”	257	264=“WEB”
“^“	‘B‘	265=“B^“
“WET”	260	266=“^WET”
“^”	‘T‘	267=“T^”

LZ78與LZW

- *LZW*之所以有效能，其原因之一是編碼端無需將字典送給解碼端，字典可以在解碼的過程中完全正確地建立。

- 為了方便起見，在編碼例子中輸出為，例如‘*W*’之編號的，我們直接用*W*代表。解碼之結果如圖所示。

LZ78與LZW

輸入碼：

“^*WED*<256>*E*<260><261><257>*B*<260>*T*”

輸入新編號	舊編號	輸出片語	符號	新加入字典中之片語
‘W’	‘ ‘	“W”	‘W’	256=“^W”
‘E’	‘W’	“E”	‘E’	257=“WE”
‘D’	‘E’	“D”	‘D’	258=“ED”
256	‘D’	“^W”	‘ ‘	259=“D^”
‘E’	256	“E”	‘E’	260=”^WE”
260	‘E’	“^WE”	‘ ‘	261=“E^“
261	260	“E^“	‘E’	262=“^WEE”
257	261	“WE”	‘W’	263=“E^W”
‘B’	257	“B”	‘B’	264=“WEB”
260	‘B’	“^WE”	‘ ‘	265=“B^“
‘T’	260	“T”	‘T’	266=”^WET”

為了方便起見，在編碼例子中輸出為，例如 ‘*W* 之編號的，我們直接用*W*代表。

LZ78與LZW

- 解碼端最後的字典和編碼端完全一致，最後之輸出字串也與編碼前之字串完全相同。

- 將編碼端與解碼端同時看，我們可以發現它們的字典雖然一樣，但解碼端總是慢了一步，於是問題就因此而產生。

- 在第n個回合編碼端的字典編號已經到了 $(255+n)$，而解碼端卻只到 $(255+n-1)$。如果編碼端在第 $(n+1)$ 個回合所送出的編號便是 $(255+n)$，解碼端收到這個編號時將不知所云。

LZ78與LZW

- *LZ78*有送出緊跟在片語之後的符號，所以編碼端與解碼端的字典完全同步，沒這個問題。*LZW*並不送出該單一符號，所以才有此問題。

LZ78與LZW

- 例：輸入 “... *IWOMBAT* ... *IWOMBATIWOMBATIXXX* ...”

輸入符號	新片語及其編號	輸出碼
……I		
WOMBAT	300=IWOBAT	288(IWOBAT)
.	.	.
.	.	.
…I	.	.
WOMBATI	400= IWOMBATI	300(IWOMBAT)
WOMBATIX	401= IWOMBATIX	400(IWOMBATI)

- 當解碼端收到編號串，先解碼出300為*IWOMBAT*，同時在這一回合定義前一個片語加 “*I*” 為其字典中之第399個片語。然後讀到下一個編號400，並且發現在字典中還沒編號至此。

LZ78與LZW

- 由於這是唯一會碰上未定義編號的情況，我們可以在解碼端的演算法中加一個例外處理副程式(*exception handler*)。以上例來說，當讀到編號400且發現編號400仍未定義時，便跳到例外處理副程式。

- 例外處理副程式所做的事也很簡單，它只是回到前一個片語，300=*IWOMBAT*，將這個片語的第一個字母拷貝到它自己的最後面得到 "*IWOMBATI*"，即為編號400的片語，然後回到主程式繼續解碼的工作。