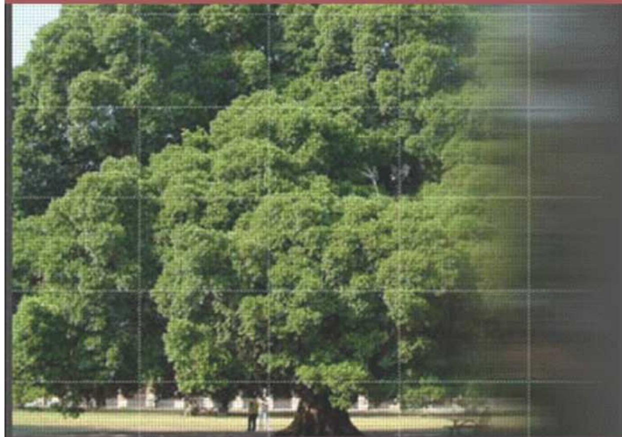


無失真資料壓縮
－統計模式

第5章

無失真壓縮

- 無失真資料壓縮一般採取的模式有二：統計模式與字典基礎模式(*statistical model and dictionary-based model*)。
- 統計模式的做法是根據每一個符號的出現機率來做編碼，每一個碼的長度因為出現機率的不同而不同。包括*Shanno-Fano*編碼、*Huffman*編碼、以及算術編碼三種，每種方法各有其高階模式與適應性模式。
- 字典基礎模式的做法則是以一個長度固定的碼取代一串符號，符號串的長度可長可短。包括*LZ77*與*LZ78*，以及由它們所衍生出來的*LZSS*與*LZW*。

無失真壓縮

- 近十幾年內的無失真資料壓縮研究重點幾乎都放在適應性(*adaptive*)模式。使用適應性模式時並不需要先看過輸入資料才能產生統計資料。它是一邊讀入資料、編碼，一邊更新統計資料。

- 讓整個系統運轉正確的關鍵是“更新模式”，不論是做編碼或解碼，在讀入輸入後，一定是在做完編碼或解碼之後才做更新(*update*)模式的動作。如此才能保證編碼端與解碼端都使用同一份最新的統計表。

Shannon-Fano編碼

- 演算法：*Shannon-Fano*編碼

第一步：對於所給定的符號源,計算每個符號的出現頻率;

第二步：將符號之出現頻率從大到小排序過;

第三步：將排序後之符號與頻率分成上下兩部分,其中上下兩部分的個別頻率和能接近;

第四步：上半部分給0、下半部分給1,這表示所有上半部分符號的碼都是以0為開頭,而下半部分符號的碼則都是以1為開頭;

第五步：對於上下兩部分繼續做第三步與第四步(分成兩部分並加一個位元)直到每個部分只剩下一個符號(變成樹葉)為止。

Shannon-Fano編碼

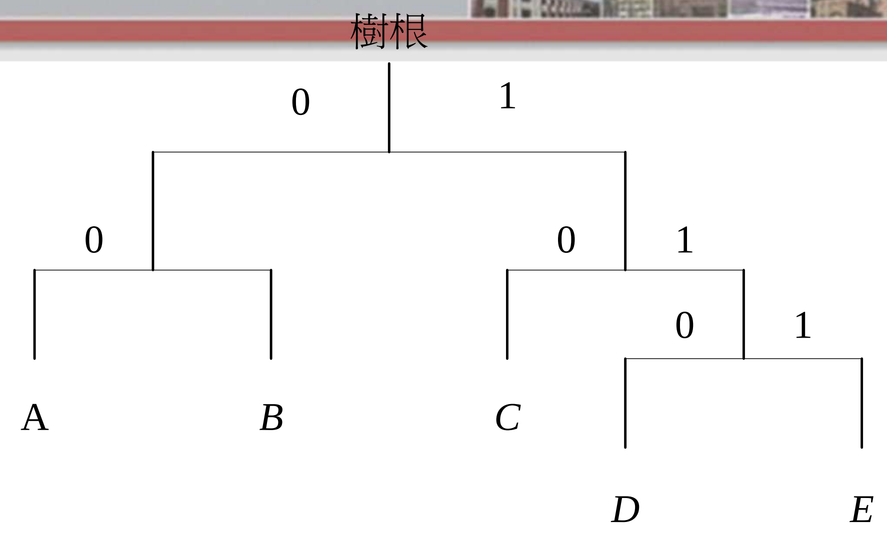
- 對於一個輸入位元串，解碼的過程是由樹根開始，決定於輸入位元串的第一個位元是1或0而選擇走入樹之右枝或左枝，然後再看下一個位元並且做同樣的事情，直到走到一樹葉為止，附於該樹葉的符號即為解碼出的符號。

- 如果輸入位元串還沒有被解碼完，我們便再回到樹根並且讀入下一個位元，繼續我們解碼的工作。

Shannon-Fano編碼

- 例：

  沿著樹根走到每一個樹葉，可得到下面的編碼表：



符號	頻率	
A	15	0 0
B	7	0 1
C	6	1 0
D	6	1 1 0
E	5	1 1 1

第2次切割

第1次切割

第3次切割

第4次切割

Huffman編碼

- *Shannon-Fano*解碼樹是從上往下建，先找出每個碼的*MSB*(*most significant bit*，最具意義位元)，然後往下做，直到樹葉為止。
- *Huffman*解碼樹則反過來，從樹葉開始工作起直到樹根為止。
- 建立*Huffman*樹首先先把每個符號攤開，成為*Huffman*樹的樹葉。這些樹葉將經由下列的演算法接成一棵樹。附在每一個樹葉的是每個符號的出現頻率或機率，並且當做該樹葉的加權值(*weight*)。

Huffman編碼

- 演算法：*Huffman*編碼

  第一步：令自由節點(*freenode*)為所有的樹葉；

  第二步：從自由節點中找出加權值最小的兩個節點；

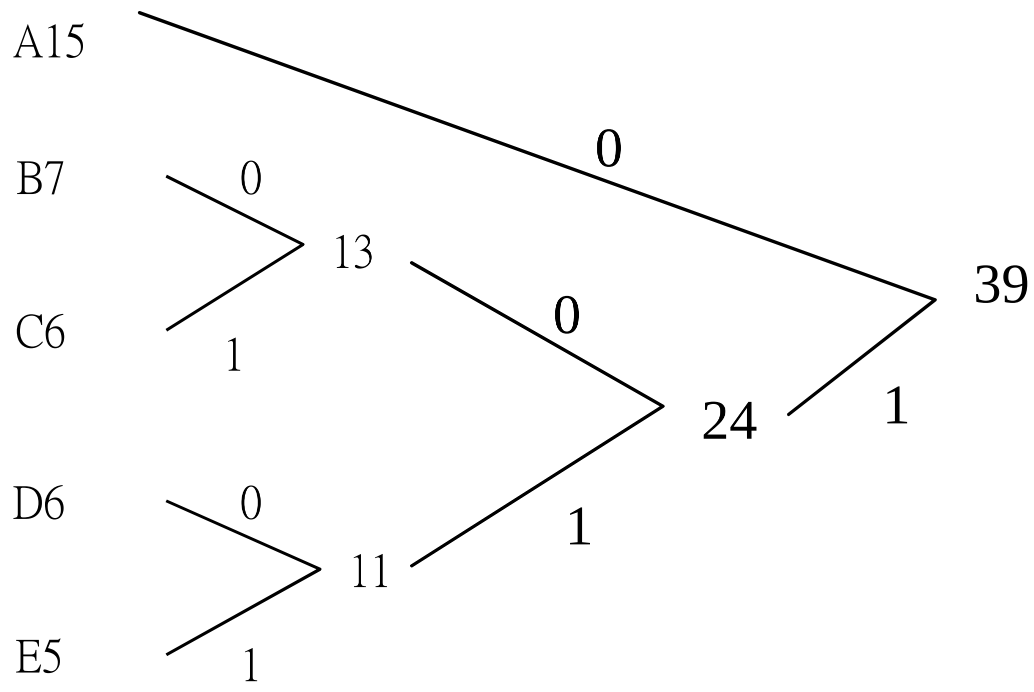
  第三步：為這兩個節點做一個父親節點，他的加權值為這兩個兒子節點的加權值和；

  第四步：將父親節點加入自由節點的行列而兩個兒子節點則從自由節點的行列中除名；

  第五步：由父親節點做到兩個兒子節點的兩枝樹枝，一枝給0另一枝給1；

  第六步：重覆執行第二步到第五步，直到只剩下一個自由節點，此為樹根。

Huffman編碼



*Huffman*碼

A	0
B	100
C	101
D	110
E	111

Huffman編碼

- *Huffman*與*Shannon-Fano*效能比較：

符號	頻率	*SF*碼 位元數	*SF*碼 總位元數	*Huff*碼 位元數	*Huff*碼 總位元數
A	15	2	30	1	15
B	7	2	14	3	21
C	6	2	12	3	18
D	6	3	18	3	18
E	5	3	15	3	15

因此，對此含有85.25個位元之資訊量的訊息，*Shannon-Fano*編碼需要89個位元，而*Huffman*編碼卻只需要87個位元。

適應性Huffman編碼

- 適應性編碼允許我們使用高階次的模式而不需要為增加的統計資料付出任何代價。
- 之所以可以做到這點是因為它只利用已經讀過的資料機動地調整*Huffman*樹，對未來的資料則絲毫不需要知道。
- 這種編碼方式的關鍵處其實就是同步。

適應性Huffman編碼

- 兄弟性質(*sibling property*)：
  如果一棵樹的節點可以按照加權值從小排到大而且每個節點又和自己的兄弟相鄰的話，我們便稱這棵樹具有兄弟性質。

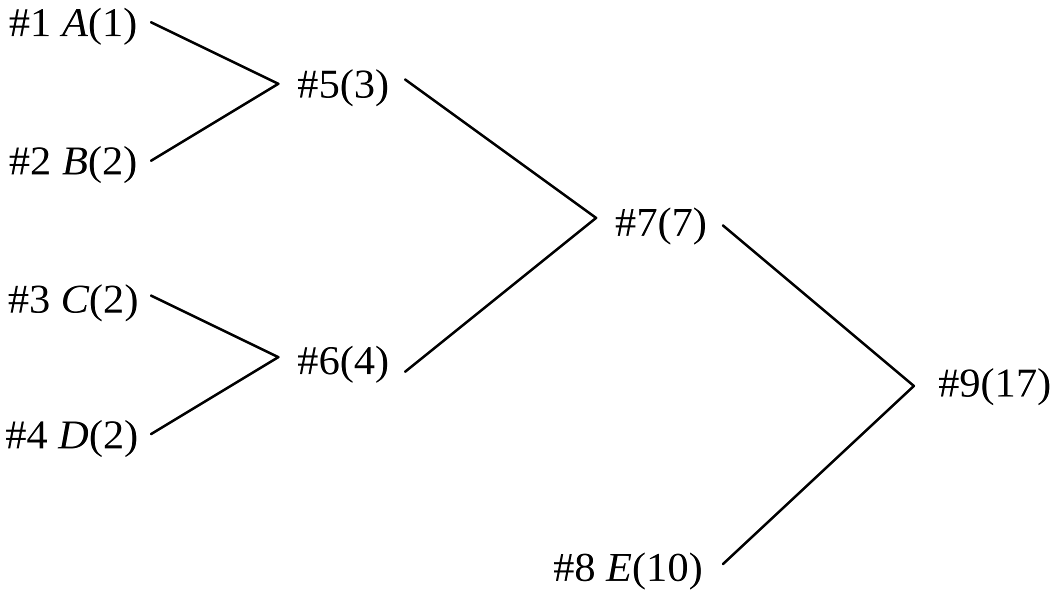


圖5.5　*Huffman*樹，其中括號內為加權值。

適應性Huffman編碼

- 定理：一棵二元樹是*Huffman*樹若且唯若它遵守兄弟性質。

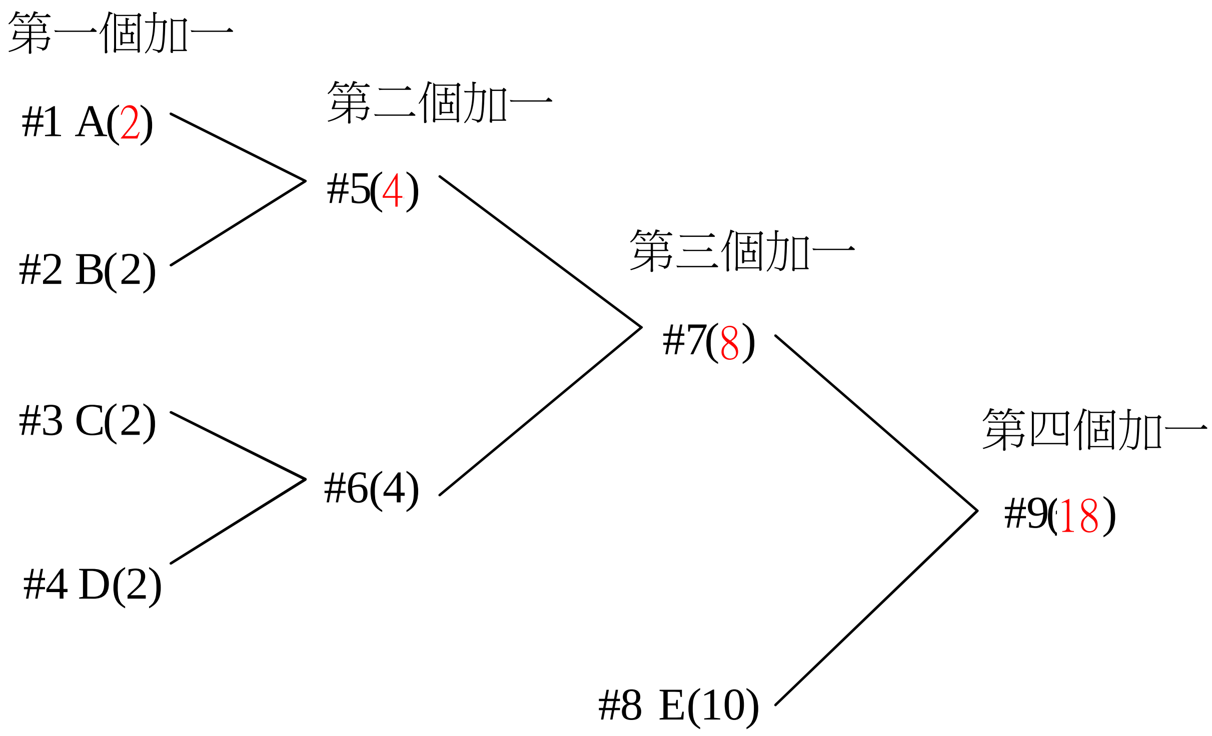
- 兄弟性質對適應性*Huffman*編碼非常重要。當某一個符號的頻率改變時，兄弟性質提供我們修改*Huffman*樹的方法與原則，只要我們保持住兄弟性質，我們就可以確定*Huffman*樹經過修改後仍然是*Huffman*樹。

適應性Huffman編碼

- 修改、或更新一棵*Huffman*樹第一個步驟是頻率的增加
- 從屬於該符號的樹葉開始，把它的頻率加1，然後往上找到它的父親節點。由於父親節點的加權值為兩個兒子節點的加權值和，因此父親節點的加權值也得加1。
- 這個程序再往上做，直到樹根的加權值也加1為止。

適應性Huffman編碼

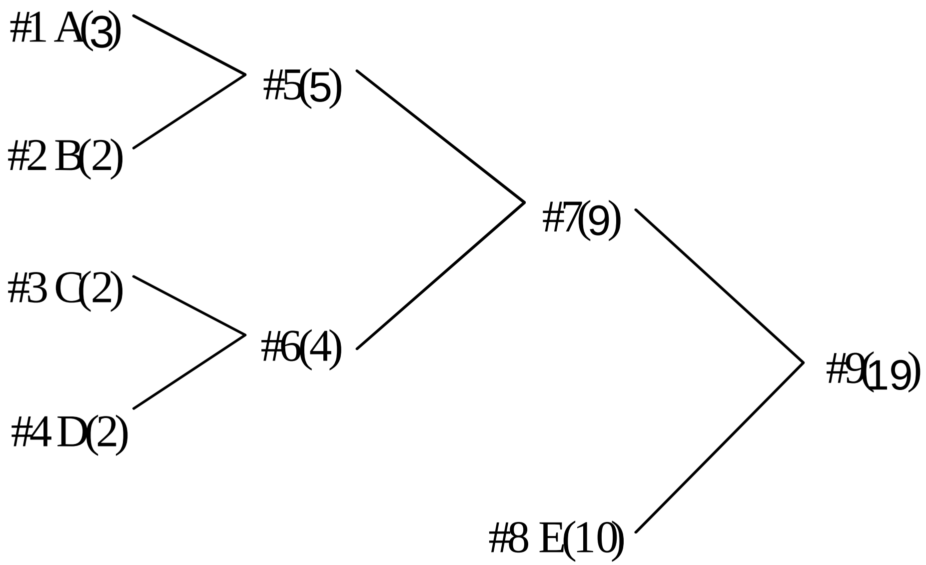
- 讀到A:



適應性Huffman編碼

- 第二個需要做的步驟是：如果增加加權值因而使得兄弟性質不再滿足時，我們得做調整的動作，否則它就不再是一棵*Huffman*樹了。

  假設我們又讀到一個*A*:

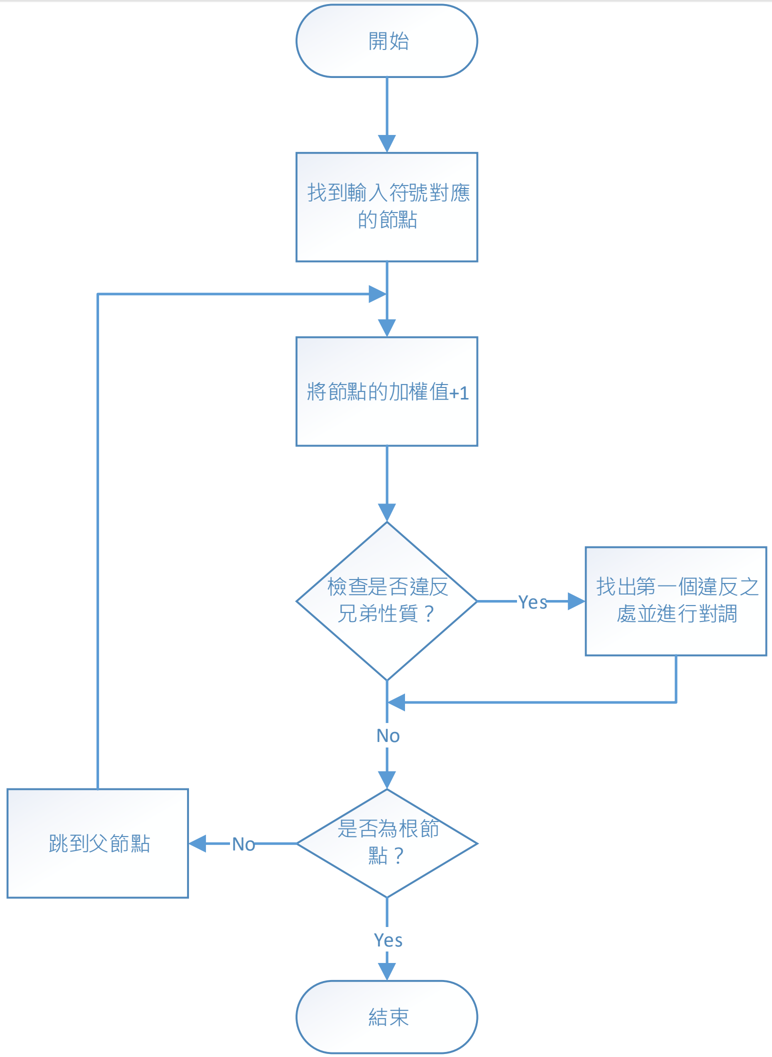


如圖所示，兄弟性質不再滿足，我們必須做調整的工作。

適應性Huffman編碼

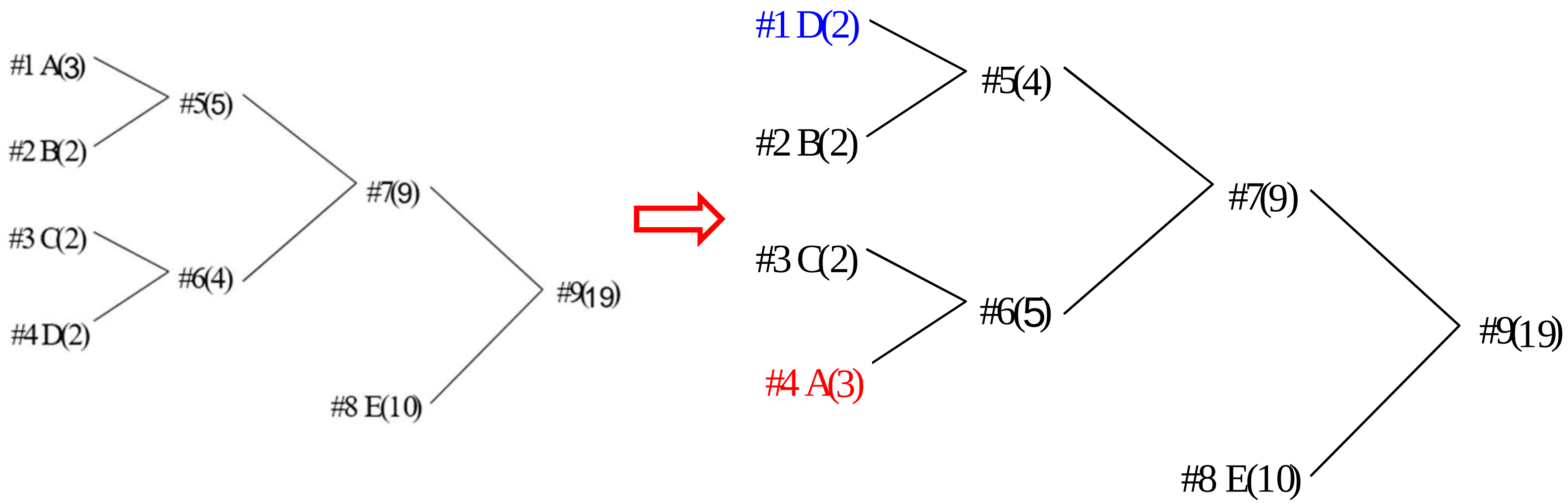
- 一般的情況是：令s_i為新讀進的符號，其編號為#*j*。當s_i的頻率增加1後，假設其加權值為*w*+1而且兄弟性質不再滿足，則編號為#(*j*+1)的節點其加權值必然為*w*。
- 令編號為#(*j*+1),#(*j*+2),...,#(*j*+*k*)的這*k*個節點其加權值也都為*w*，而編號#(*j*+*k*+1)的節點其加權值則大於等於*w*+1。
- 將編號為#*j*的節點(s_i)與編號為#(*j*+*k*)的節點內容互換(編號不動)。交換完畢後，繼續更新編號#*j*及#(*j*+*k*)節點的父親節點(以及父親的父親...)的加權值，直到樹根為止。

適應性Huffman編碼-更新流程



適應性Huffman編碼

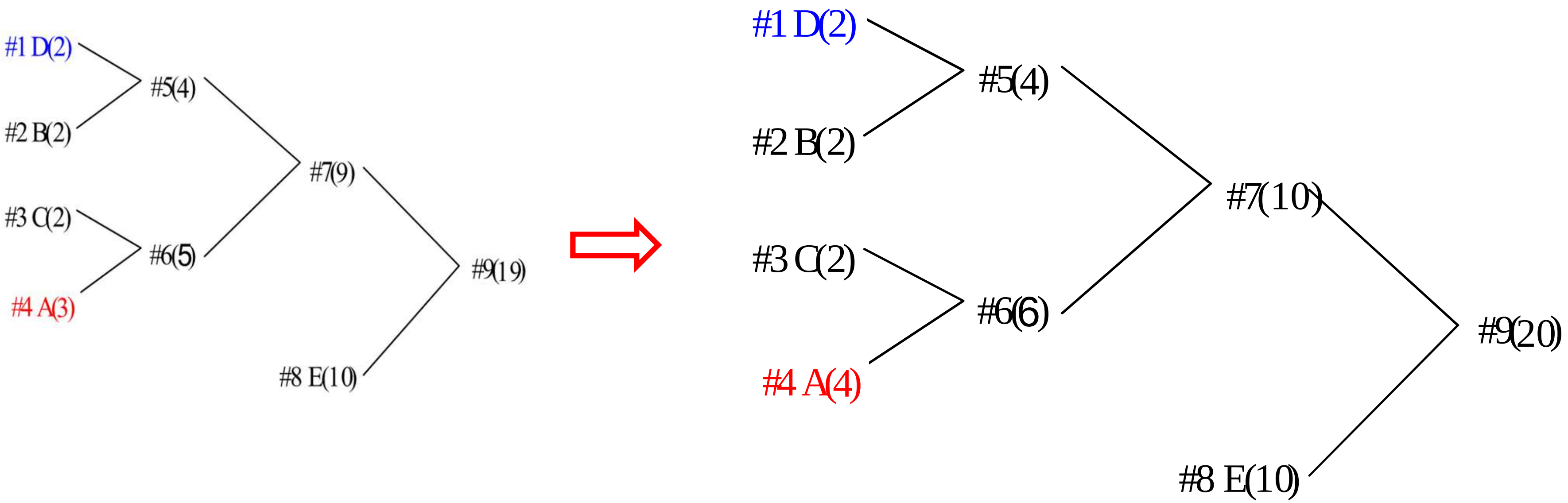
- 造成兄弟性質不再滿足的節點在這個例子中是*A*，調整這棵樹的方法便是交換*A*(造成兄弟性質不再滿足的節點)與*D*(加權值比*A*少1而且編號最高的節點)的位置。



- 在交換位置之後，*A*的父親與*D*的父親，這兩個節點的加權值也要跟著更新，這個動作一直做到樹根為止。

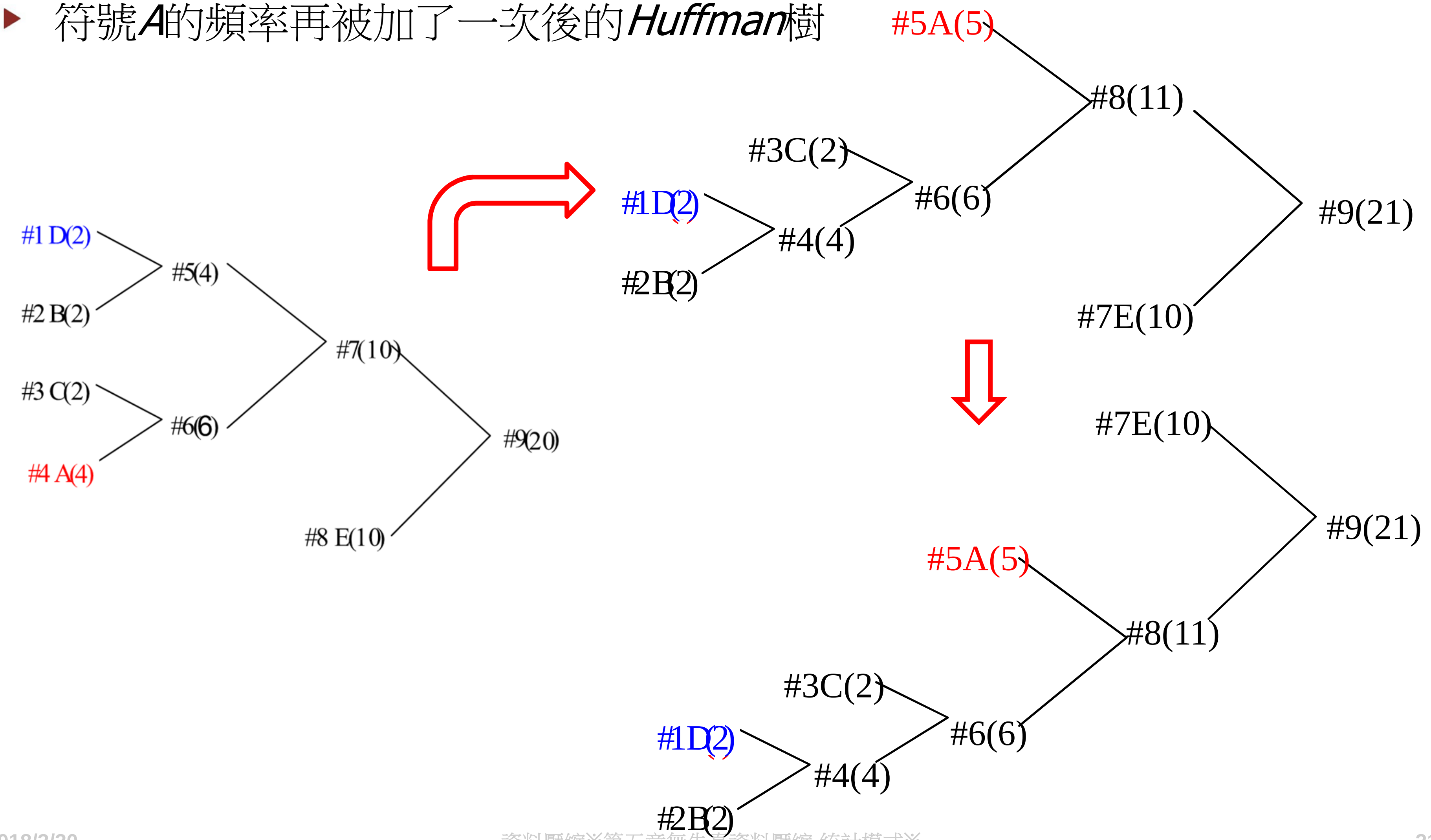
適應性Huffman編碼

- 符號*A*的頻率又被加了一次後的*Huffman*樹



適應性Huffman編碼

- 符號*A*的頻率再被加了一次後的*Huffman*樹



適應性Huffman編碼

- 一開始*A*的加權值只有1，它的碼長度為三個位元；現在它的加權值增加到5，它的碼長度變為兩個位元。
- 符號*C*的碼長度仍然是三個位元，但是*B*與*D*的碼長度則已經變為四個位元。

適應性Huffman編碼

- 要使編碼更有效能的辦法之一，便是確定我們的編碼端不會把空間浪費在訊息中根本不存在的符號上。
- 比較好的方法是從一個空的統計表開始，只有當符號被讀到才將其加入統計表內。
- 真正實行的方法是一開始將*Huffman*樹設定成只含有兩個特殊符號：*EOF*(*end of file*，代表檔案結束)及*ESC*。兩個特殊符號的加權值都設為1，而且永遠不改變。

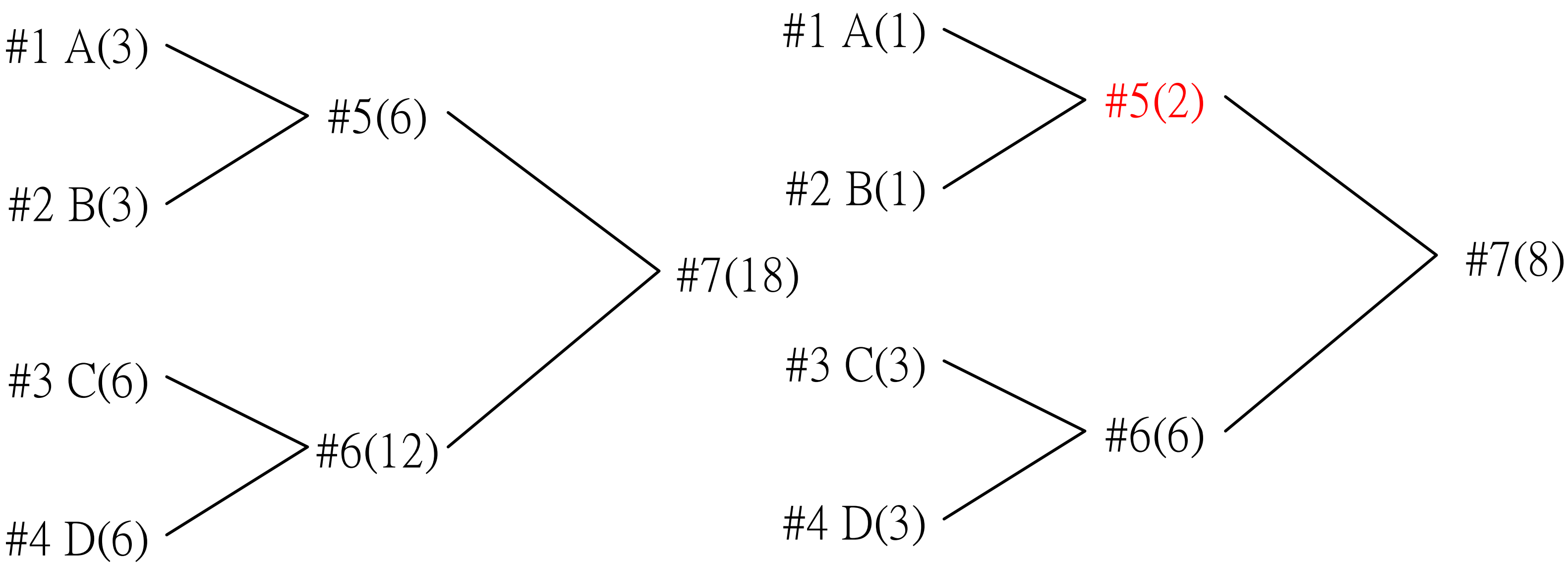
適應性Huffman編碼

- *ESC*的作用是：
  - 如果編碼端讀到的符號在*Huffman*樹裡已經有了，那麼按照正常程序將這個符號的*Huffman*碼送出，然後把這個符號的加權值加1，如果需要的話，做必要的調整。
  - 如果編碼端讀到的符號沒有出現過，這時編碼端會先送出*ESC*的*Huffman*碼，然後再送出這個符號的*ASCII*碼(或任何未編碼過的其他種類的碼)，最後再將這個符號加入*Huffman*樹裡並且設定其加權值為1，必要的話也為*Huffman*樹做調整的工作。

適應性Huffman編碼

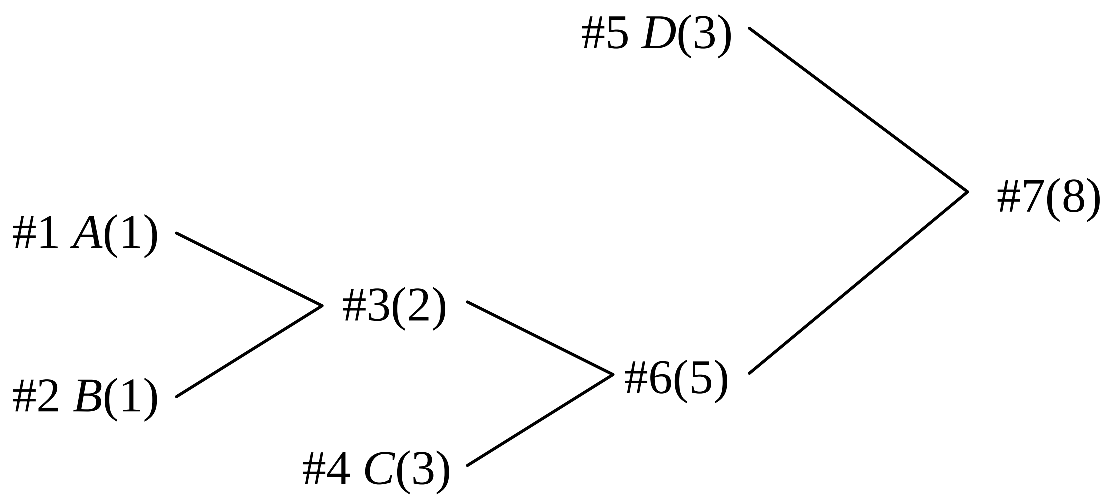
- 溢位問題(*the overflow problem*)
  - 解決的辦法通常是將所有節點的加權值都除以2。
  - 加權值除以2產生的問題是：*Huffman*樹的形狀可能必須改變。因為我們所處理的是整數除法，小數點後面的數完全忽略，因此可能導致樹不再是*Huffman*樹。

適應性Huffman編碼



將加權值除以2所導致的問題。

適應性Huffman編碼



重建後之*Huffman*樹

算術編碼(Arithmetic coding)

- *Huffman*編碼已經被證明是即時碼中最好的，甚至於只要編碼方法是給每個符號一個特定的碼，那麼它的效能就無法超過*Huffman*編碼。
- *Huffman*編碼的問題在於碼的長度必須是整數。當某個符號的出現機率相當高時，*Huffman*編碼的效能跟理論推測會相差更遠。
- 假設我們所採用的模式計算出某個符號的出現機率為0.9，則其最佳編碼長度應該是0.15個位元。*Huffman*編碼即使只使用一個位元來編碼它，也比實際所需高六倍。

算術編碼(Arithmetic coding)

- 尤其是在壓縮二元影像的時候(例如傳真)，由於每一個像素只有0或1兩種可能值，不管0或1的各別出現機率為多少，*Huffman*編碼總之還是得各用一個位元來編碼它們，根本沒有辦法做什麼壓縮。
- 傳統解決辦法是將幾個位元合起來構成一個區段，然後再使用*Huffman*編碼。但是這就限制了*Huffman*編碼的一般性。

算術編碼(Arithmetic coding)

- 算術編碼避開了一個符號一個碼的想法而採取用一個實數來表示一串符號的新點子。
- 算術編碼的輸出是一個介於0與1之間的實數，利用這個實數，解碼端可以唯一地解碼回原來的訊息。要編碼之前，訊息內每個符號的出現機率必須先求出。
- 一旦每個符號的出現機率知道了，我們需要對每個符號設定一個從0到1之間的範圍，出現機率大則範圍大、出現機率小則範圍小。至於每個符號所擁有的範圍是從哪裡開始則無所謂，只要編碼端與解碼端一致便可。

算術編碼(Arithmetic coding)

- 例：編碼 "*BILL^GATES*"

符號	機率	範圍
SPACE(^)	1/10	0.0≤*r*<0.1
A	1/10	0.1≤*r*<0.2
B	1/10	0.2≤*r*<0.3
E	1/10	0.3≤*r*<0.4
G	1/10	0.4≤*r*<0.5
I	1/10	0.5≤*r*<0.6
L	2/10	0.6≤*r*<0.8
S	1/10	0.8≤*r*<0.9
T	1/10	0.9≤*r*<1.0

算術編碼(Arithmetic coding)

- 令*low*及*high*分別表示編碼後所得之實數其可能範圍的下界及上界。在還沒讀入任何輸入符號前，*low*設定為0.0而*high*則設定為1.0。
- 讀入第一個符號為*B*，由範圍表知道其範圍為0.2到0.3，因此我們將*low*重新設定為0.2，*high*重新設定為0.3。
- 再讀入下一個符號*I*，由範圍表知道其範圍為$0.5 \leq r < 0.6$，因此在編碼後，輸出實數的可能範圍得縮小到0.2～0.3這個範圍的第50%到第60%的部分，即*low*變為0.25而*high*變成0.26。

算術編碼(Arithmetic coding)

- 演算法：算術編碼－編碼

  第一步：$low \leftarrow 0.0$；$high \leftarrow 1.0$；

  第二步：讀入下一個符號，c；

  第三步：$range \leftarrow high\text{-}low$；

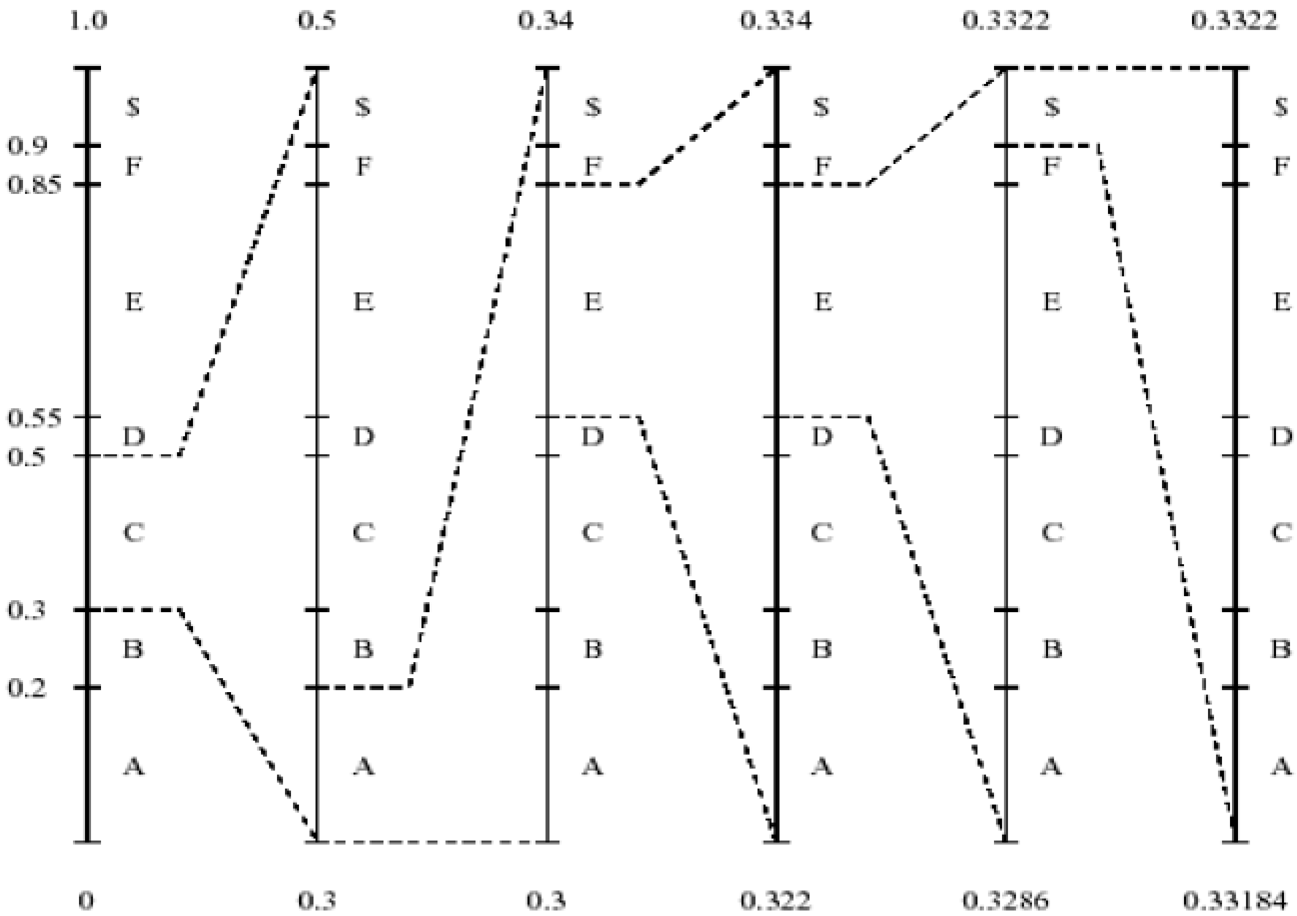
  第四步：查範圍表，令c的範圍為$l \le r < h$；

  設定$high \leftarrow low + range \times h$；$low \leftarrow low + range \times l$；

  第五步：如果還有輸入符號還沒編碼，則回到第二步，否則執行第六步；

  第六步：輸出low；

算術編碼(Arithmetic coding)



算術編碼(Arithmetic coding)

新讀入符號	*low*值	*high*值
	0.0	**1.0**
B	**0.2**	**0.3**
I	**0.25**	**0.26**
L	**0.256**	**0.258**
L	**0.2572**	**0.2576**
^(space)	**0.25720**	**0.25724**
G	**0.257216**	**0.257220**
A	**0.2572164**	**0.2572168**
T	**0.25721676**	**0.2572168**
E	**0.257216772**	**0.257216776**
S	**0.2572167752**	**0.2572167756**

最後的*low*值0.2572167752將用來編碼 "*BILL^GATES*"這個訊息。

算術編碼(Arithmetic coding)

- 當收到0.2572167752這個實數值，解碼端先找出這個實數是落在那一個符號的範圍內。
- 由範圍表我們知道0.2572167752落在0.2到0.3的範圍內(符號*B*之範圍)，因此第一個符號必然是*B*。
- 先減去*B*的*l*值得到0.0572167752，然後除以*B*的範圍，$h–l$=0.1，得到0.572167752。然後再從範圍表裡找出這個數字落在哪個符號的範圍內，也就是我們下一個解碼出的符號，*I*。

算術編碼(Arithmetic coding)

- 演算法：算術編碼－解碼

第一步：讀入編碼值，*number*；

第二步：從範圍表裡找出*number*落在哪一個符號的範圍內，假設是*c*而且*c*的範圍從*l*到*h*；(當然，$l \le number < h$)

第三步：輸出*c*；

第四步：$number \leftarrow number - l$；

第五步：$number \leftarrow number / (h - l)$；

第六步：回到第二步直到*number*為0；

算術編碼(Arithmetic coding)

r	c	Low	High	range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	^	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

算術編碼(Arithmetic coding)

- 算術編碼的編碼過程是根據輸入符號將實數之可能範圍逐漸縮小。新範圍的大小跟該符號的出現機率大小成正比。
- 解碼則反過來，範圍依解碼出的符號之出現機率大小而或快或慢地擴張。
- 算術編碼雖然比*Huffman*編碼有更好的壓縮效能，卻仍然沒能完全取代掉*Huffman*編碼。這是因為它的編碼及解碼需要乘除法的運算，因此速度比較慢。

高階次模式與適應性模式

- 當我們使用統計模式的無失真資料壓縮法時，我們需要一個資料的模式。這個模式得做兩件事：

  (1)它必須能準確地預測輸入資料中每個符號的出現頻率或機率

  (2)由這個模式所產生的機率必須愈不平均愈好。

  做到了這兩件事才能達到資料壓縮的目的。

高階次模式與適應性模式

- 正確地預測出輸入資料中每個符號的出現機率：
  - 這類的編碼方式都是採取高出現機率則短編碼長度的基本原則。
  - 如果*E*的出現機率是1/4，那麼應該使用兩個位元來編碼*E*；如果*Z*的出現機率只有1/1000，可能會使用10個位元來編碼*Z*。
  - 如果模式沒有辦法正確地預測出各項出現機率，可能會使用10個位元來編碼*E*，使用兩個位元來編碼*Z*，結果做的是資料擴充而不是資料壓縮。

高階次模式與適應性模式

- 預測的機率分布不是一致分布：
  - 我們可以建一個模式，對於所有256個可能符號都給它們出現機率為1/256。採用這個模式沒辦法做什麼資料壓縮，每個符號和原來一樣就是需要八個位元來表示。
  - 只有在我們能正確地預測出不同於一致分布的機率分布，才能達到資料壓縮的目的。
  - 決定於我們所採取的模式，我們會得到不同的機率分布。

高階次模式與適應性模式

- 譬如，壓縮一個C語言程式，在整個C程式裡，“換行”這個符號的出現機率可能只有1/40。如果採用的模式是第一階馬可夫過程，那麼在“}”這個符號之後是“換行”這個符號的機率便提高到1/2。

- 使用較高階次的模式是提高壓縮比的好方法之一，能以高機率預測一個符號的出現，則該符號所需之編碼位元數便能降低(∵資訊量降低)，而愈高階次的模式一般也都能讓我們以更高的機率預測一個符號的出現。

高階次模式與適應性模式

- 不幸的是，當模式的階次線性增加，所需之統計表所佔的空間會呈現指數成長。因為第n階馬可夫過程需要q^n個機率表，其中q為符號源內之符號數。

- 適應性壓縮可以解決這個問題。在適應性壓縮方法裡，壓縮端與解壓縮端一致地從同一個統計表開始運作，它們同步地編碼、解碼、以及更新統計表，因此對於同一個符號的編碼與解碼，使用的統計表始終是同一個。

- 整個過程中，編碼端不需要送統計表或編碼表給解碼端。適應性模式也苦於更新模式時所需要花的代價。舉例來說，當使用算術編碼時，更新某一個符號的出現頻率(機率)可能使其他符號的範圍設定也必須跟著調整。解決的辦法一般是近似地調整符號的範圍

高階次模式與適應性模式

- 資料壓縮比的測試顯示，統計模式至少可以表現得和字典基礎模式一樣好。但是由於這些程式必須採用高階次的適應性模式，因此執行速度比較慢。幸運的是，硬體進步的速度非常快。
- 使用*Huffman*編碼的標準包括：*JPEG*、*ITU-T Group* 3與*Group* 4等。
- 使用算術編碼的標準包括：*JBIG*、*JBIG2*、*JPEG*、*JPEG2000*、*H.263*、*MPEG-4*、*H.264*等。