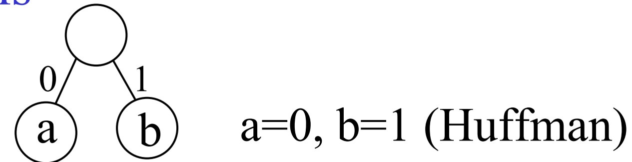
Chapter 5

Arithmetic Coding

Advantages of arithmetic coding:

1. Deal with source data with few symbols

X={a(0.1), b(0.9)}

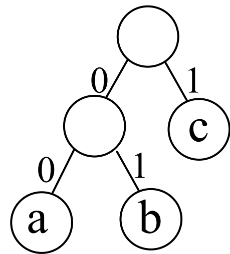


a=0, b=1 (Huffman)

2. Deal with source data with symbols having highly skewed probabilities

If X={a(0.05), b(0.05), c(0.9)}, then H=4.32*0.05*2+0.15*0.9=0.57 bits

Huffman Coding



symbol	Code
a	00
b	01
c	1

The difference between the average code length and the entropy is 0.53 bits, about 93% of the entropy.

H=2*0.05*2+1*0.9=1.1 bit

3. Generate a unique identifier or tag for the sequence to be encoded

4. Doesn't need to generate the codewords for each input symbol

Arithmetic Coding

- 算術編碼法與霍夫曼編碼法都是屬於無失真資料壓縮，亦即壓縮過的資料可以完整重建成原來的模樣。
- 這兩種方法所依據的理論基礎相當類似
  - 都是利用出現機率較高的符號，使用較少位元的字碼來表示，出現機率較低的符號，則使用較多位元的字碼來表示。

Encoding Procedure

假設資料源中，一共含有n個不同的輸入符號，輸入符號集A可表示成$A=\{s_1, s_2,..., s_n\}$，**累積密度函數**(cumulative density function) C定義如下：

$$C(k)=\sum_{i=1}^{k}P(s_i)=P(s_1)+P(s_2)+P(s_3)+...+P(s_k),$$

$P(s_k)$ 表示第 k 個來源符號的機率值，且 $1 \le k \le n$；

$C(k)$ 表示前面 k 個來源符號的機率值累積總和。

算術編碼演算法：(假設資料源中，含有n個不同的符號)

步驟一： 計算資料源中每個符號的出現機率，並求符號的累積機率值。

步驟二： 令回合計數i=0，初始編碼區間定為$[l_{(0)}, u_{(0)}) = [0,1)$ (大於等於0但小於1)，其中 $l_{(0)}$ 代表初始編碼區間的下限(lower limit)，$u_{(0)}$ 則代表初始編碼區間的上限(upper limit)。

Arithmetic Coding(Cont.)

- 霍夫曼編碼法是利用二元編碼樹的方式來編碼資料，它會將整個編碼空間以最佳整數個位元的方式來分配
- 算術編碼是用一個實數來表示要壓縮的資料
  - 換句話說，它將整個編碼空間以精確至小數的方式來分配，以達到更佳的編碼效果。
  - 算術編碼法最大的特點在於不需要每個符號都編碼並且送出結果(霍夫曼法每個輸入符號都要編碼成整數個位元)
  - 算術編碼用一個介於0到1之間的實數來代表編碼到目前的情形。當輸入符號持續進入編碼時，這個實數會跟著改變，由於存在0到1之間的實數共有無限多個，因此任何待壓縮檔案都能用一個獨一無二的實數來代表它。

Encoding Procedure

步驟三：i=i+1，若x代表目前要編碼的輸入符號，依照符號們的機率值來分割目前的編碼區間，並且找出輸入符號集A內和x相等的那個符號。假設是符號s_k，則將s_k對應的符號機率區間當成一個新的編碼區間 $[l_{(i)}, u_{(i)})$。

實際計算新編碼區間的過程，可用下式來實現：

$$[l_{(i)}, u_{(i)}) = [\, l_{(i\text{-}1)}+(u_{(i\text{-}1)}-l_{(i\text{-}1)})\times C(k-1)\,,\; l_{(i\text{-}1)}+(u_{(i\text{-}1)}-l_{(i\text{-}1)})\times C(k)\,)$$

$l_{(i\text{-}1)}$ 代表讀入此回合輸入符號x前的編碼區間下限，$u_{(i\text{-}1)}$ 則代表讀入此回合輸入符號 x 前 的編碼區間上限。$C(k-1)$ 代表 s_k前面 k-1 個來源符號的累積機率值，$C(k)$代表前面k個來源符號(含s_k)的累積機率值。

步驟四： 重複步驟三，每次讀入一個輸入符號，並根據該符號適當地調整編碼區間，一直到所有輸入符號都處理完畢為止。

步驟五： 從最後的編碼區間 $[l_{(i)}, u_{(i)})$內任選出一個實數值，稱為標籤(*label*)，即 $l_{(i)} \le label < u_{(i)}$，將標籤值當成編碼的結果輸出。

Example for Arithmetic Coding

Example 5.2: (page 110, textbook)

Encode the "arithmetic"

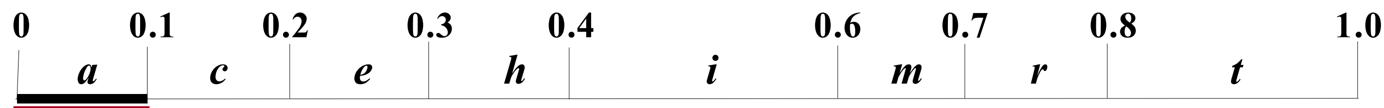
$$A=\{s_1, s_2,\ldots, s_8\}=\{a,c,e,h,i,m,r,t\}$$

Pre-scan and determine the probability and cumulative density of each symbol

$$P(a)=0.1; P(c)=0.1; P(e)=0.1; P(h)=0.1; P(i)=0.2; P(m)=0.1; P(r)=0.1; P(t)=0.2,$$

$$C(1)=0.1; C(2)=0.2; C(3)=0.3; C(4)=0.4; C(5)=0.6; C(6)=0.7; C(7)=0.8; C(8)=1$$

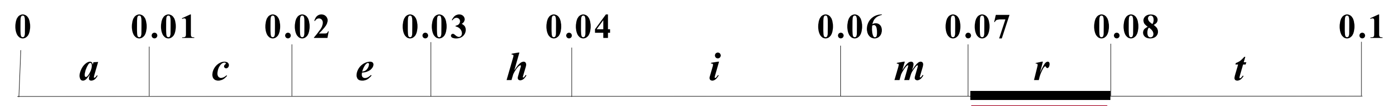
i=0, initial interval $[l_{(0)}, u_{(0)})=[0, 1)$, encode the first input symbol a



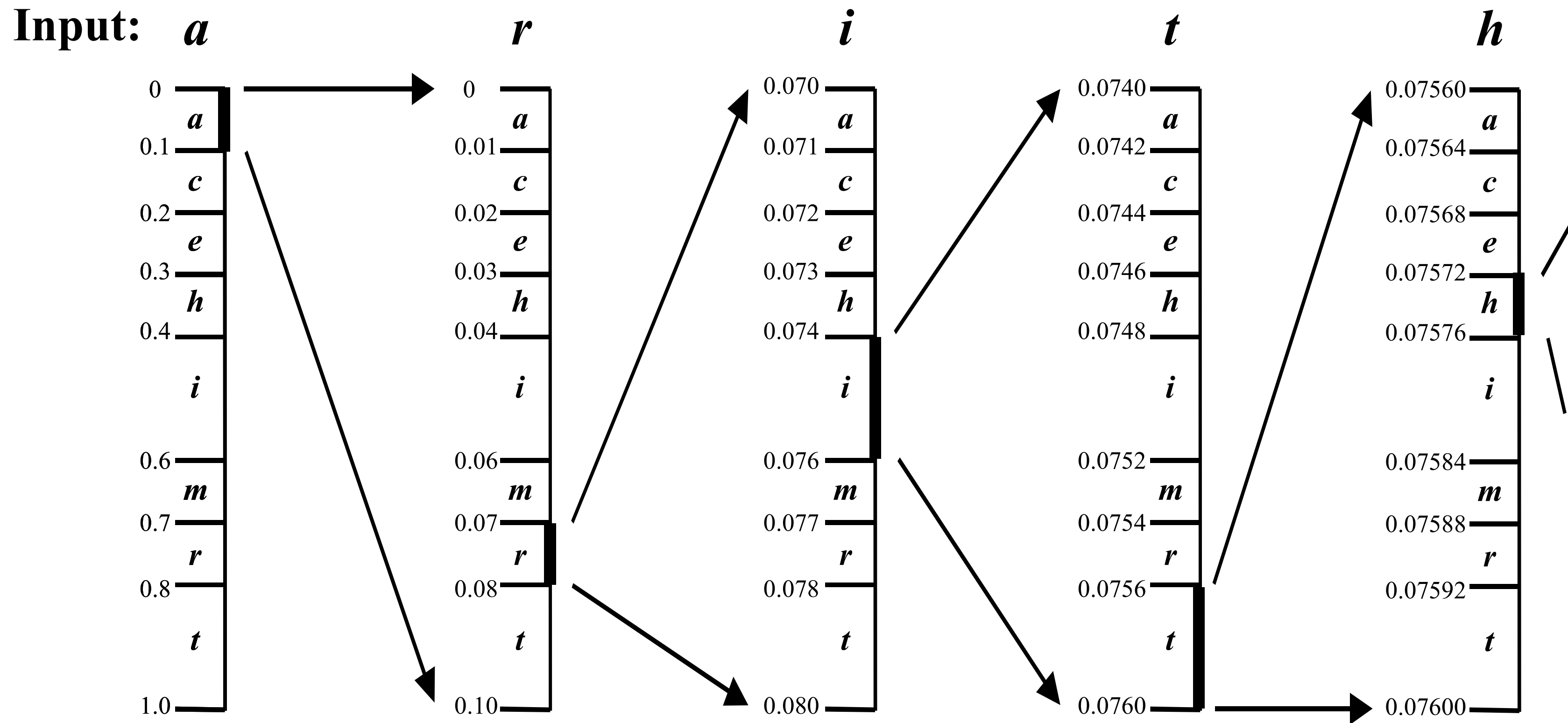
$$[l_{(1)}, u_{(1)}) = [l_{(0)} + (u_{(0)} - l_{(0)}) \times C(k-1), l_{(0)} + (u_{(0)} - l_{(0)}) \times C(k))$$

$$= [0+(1-0)\times C(1-1), 0+(1-0)\times C(1)) = [0+(1-0)\times 0, 0+(1-0)\times 0.1) = [0, 0.1)$$

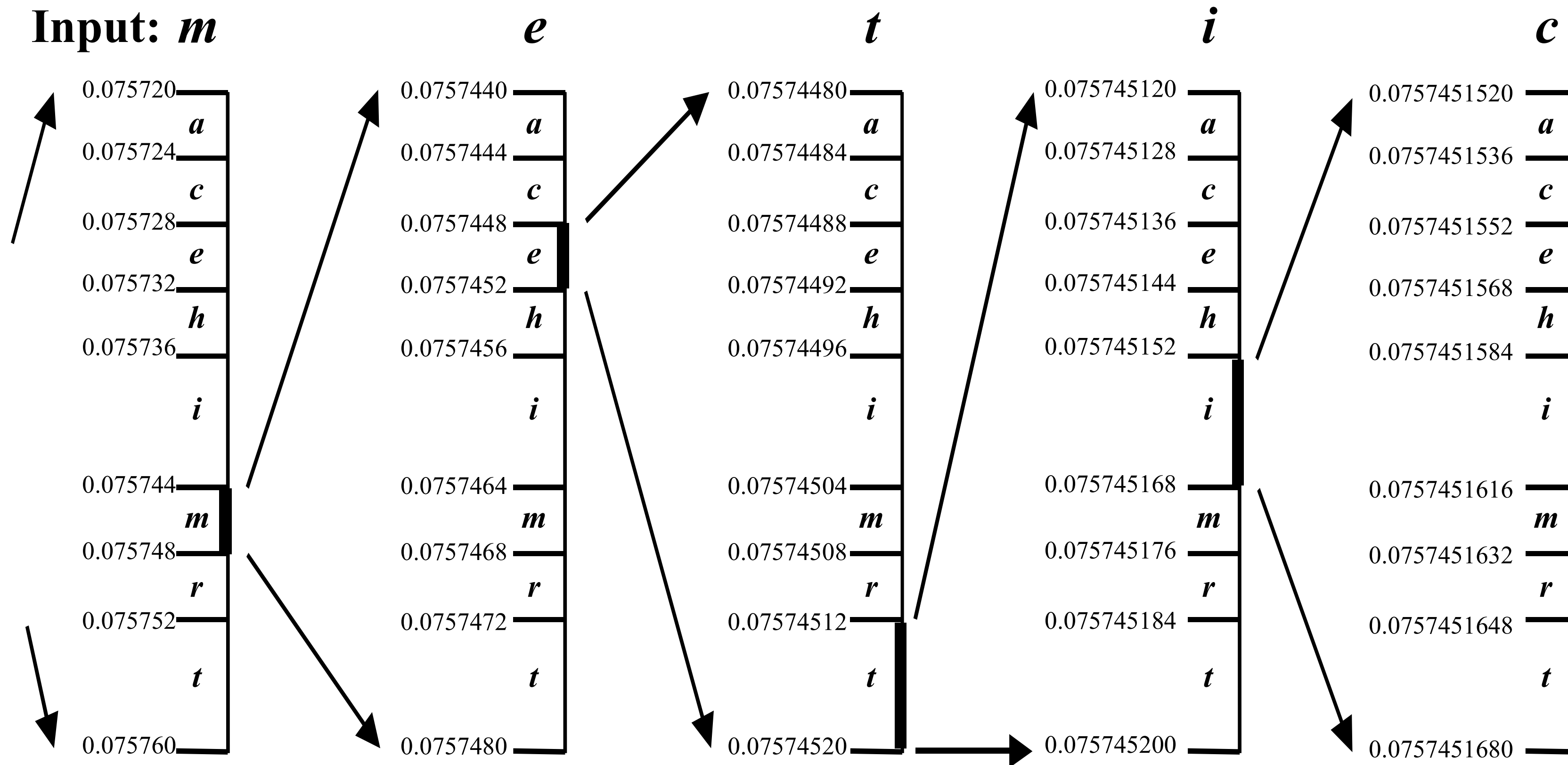
i=1, the interval $[l_{(1)}, u_{(1)})=[0, 0.1)$, encode the second input symbol r



Example for Arithmetic Coding



Example for Arithmetic Coding(Cont.)



Advantages of Arithmetic Coding

- 算術編碼法是根據目前的輸入符號為何，來持續縮小編碼區間
- 若輸入符號的出現機率較高的話，編碼區間縮小的比例會較小(縮小速度較慢)
  - 當輸入符號的出現機率較低，編碼區間縮小的比例會較大(縮小速度較快)。因此，要編碼的資料如果出現機率一直都很高的話，編碼區間縮小速度較慢，編碼結束時最後的編碼區間會較大，標籤所需要的小數精確度較少，故資料量較少，壓縮效果較好。
  - 要編碼的資料如果出現機率一直都很低的話，編碼區間縮小速度較快，編碼結束時最後的編碼區間會較小，標籤所需要的小數精確度非常高，故資料量較多，壓縮效果較差。
- 和霍夫曼編碼的主要原理--出現機率愈高的符號，使用愈少位元的字碼來表示，有異曲同工之妙。-----如果希望編碼達到較好的壓縮效

Decoding Procedure

算術解碼演算法：

步驟一： 令回合計數i=0，初始編碼區間定為 $[l_{(0)}, u_{(0)}) = [0, 1)$。

步驟二： i=i+1，依照符號們的機率值來分割目前的編碼區間，並根據輸入的$label$值找出$label$位在那一個符號區間內。假設是符號s_k，則將s_k當成解碼符號送出，並將s_k對應的符號機率區間當成一個新的編碼區間 $[l_{(i)}, u_{(i)})$。

實際決定第s_k個符號為何的計算過程，可用下式來實現：

從目前的輸入符號集A中，找出能滿足下式的第s_k個符號：

$$l_{(i)} \le label < u_{(i)} \qquad (5.1)$$

其中 $l_{(i)} = l_{(i-1)} + (u_{(i-1)} - l_{(i-1)}) \times C(k-1)$ 而 $u_i = l_{(i-1)} + (u_{(i-1)} - l_{(i-1)}) \times C(k))$

也就是將s_k的符號區間視為新編碼區間 $[l_{(i)}, u_{(i)})$。

事實上，(5.1)式可改寫為：

$$l_{(i-1)} + (u_{(i-1)} - l_{(i-1)}) \times C(k-1) \le label < l_{(i-1)} + (u_{(i-1)} - l_{(i-1)}) \times C(k)$$

故也可寫為 $$C(k-1) \le \frac{label - l_{(i-1)}}{u_{(i-1)} - l_{(i-1)}} < C(k) \qquad (5.2)$$

Example for Arithmetic Decoding

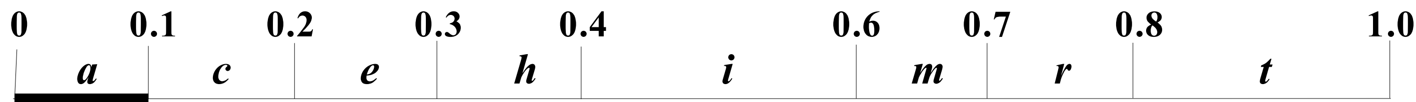
Example 5.3: (page 114, textbook)

$P(a)=0.1; P(c)=0.1; P(e)=0.1; P(h)=0.1; P(i)=0.2; P(m)=0.1; P(r)=0.1; P(t)=0.2$

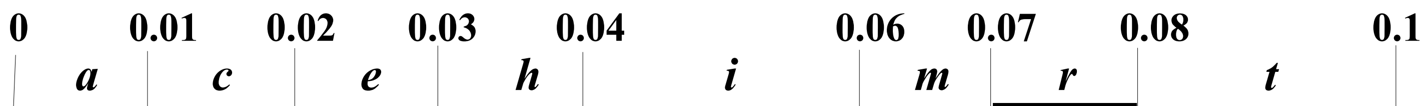
Decode the "0.075745154"

$C(1)=0.1; C(2)=0.2; C(3)=0.3; C(4)=0.4; C(5)=0.6; C(6)=0.7; C(7)=0.8; C(8)=1$

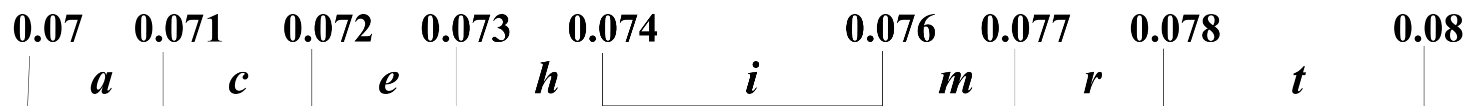
i=0, initial interval $[l_{(0)}, u_{(0)})=[0,1)$, find the location for 0.075745154



$0 \le 0.075745154 < 0.1$, decode a, the new interval becomes $[l_{(1)}, u_{(1)})=[0, 0.1)$



$0.07 \le 0.075745154 < 0.08$, decode r, the new interval becomes $[l_{(2)}, u_{(2)})=[0.07, 0.08)$



$0.074 \le 0.075745154 < 0.076$, decode i, the new interval becomes $[l_{(3)}, u_{(3)})=[0.074, 0.076)$

Rescaling

- 算術編碼整個資料檔後，所得到的標籤值可能會小到無法用電腦現有的精確度來表示
- 為了配合電腦使用，標籤值應以二進制方式表示
- 一種rescaling 做法：
  - 編碼進行時，一旦發現編碼區變得介於0 跟 0.5之間時， 送出字碼“0”來通知解碼端，並將編碼區間乘以2 來放大。
  - 若發現編碼區變得介於0.5 跟 1之間時， 送出字碼“1”來通知解碼端，並將編碼區間減去0.5再乘以2 來放大。
  - 若發現編碼區跨越0.5時， 不作任何放大動作。

Tag Generation With Rescaling

Example 5.4: (page 118, textbook)

$P(x_1)=0.6, P(x_2)=0.2, P(x_3)=0.2 \Rightarrow C(0)=0, C(x_1)=0.6, C(x_2)=0.8, C(x_3)=1,$ input: $x_2\, x_1\, x_2\, x_2$

(1) Input x_2, $l_{(1)}$=0.6; $u_{(1)}$=0.8; (2) Input x_1, $l_{(2)}$=0.6+0.2*0=0.6; $u_{(2)}$=0.6+0.2*0.6=0.72;

(3) Input x_2, $l_{(3)}$=0.6+0.12*0.6=0.672; $u_{(3)}$=0.6+0.12*0.8=0.696;

(4) Input x_2, $l_{(4)}$=0.672+0.024*0.6=0.686; $u_{(4)}$=0.672+0.024*0.8=0.691;

$\text{TAG} = 0.6875\ \ (0.686 \le \text{TAG} \le 0.691)$, output $(.1011)_2 = 0.5 + 0.125 + 0.0625$

Rescaling function $\boldsymbol{R_1 : [0, 0.5) \to [0, 1]}$**; output 0,** $\boldsymbol{R_1(x) = 2x}$**;**

$\boldsymbol{R_2 : [0.5, 1) \to [0, 1]}$**; output 1,** $\boldsymbol{R_2(x) = 2(x - 0.5)}$**;**

(1) Input x_2, $l_{(1)}$=0.6; $u_{(1)}$=0.8; contained in [0.5, 1) (upper half of unit interval)

Out=1 and rescaling; $l_{(1)}$=2*(0.6-0.5)=0.2; $u_{(1)}$=2*(0.8-0.5)=0.6 。

(2) Input x_1, $l_{(2)}$=0.2+(0.6-0.2)*0=0.2; $u_{(2)}$=0.2+(0.6-0.2)*0.6=0.44; [0, 0.5) (lower half of 1)

Out=0 and rescaling; $l_{(2)}$=2*0.2=0.4; $u_{(2)}$=2*0.44=0.88 。

(3) Int=x_2, $l_{(3)}$=0.4+(0.88-0.4)*0.6=0.688; $u_{(3)}$=0.4+0.48*0.8=0.784;

Out=1 and rescaling; $l_{(3)}$=2*(0.688-0.5)=0.376; $u_{(3)}$=2*(0.784-0.5)=0.568 。

(4) Int=x_2, $l_{(4)}$=0.376+(0.568-0.376)*0.6=0.491; $u_{(4)}$=0.529;

The interval [0.491, 0.529) is not confined to either the upper or lower half of the unit interval, so output nothing. Now, ***Output=101,*** TAG= 0.5(0.491<=0.5<=0.529)

So the binary code is .1; Final ***Output=1011*** 。

適應性算術編碼

- 適應性算術編碼(adaptive arithmetic coding)法的原理與精神，大體上都和算術編碼法差不多
- 其主要的特色是--適應性算術編碼法會隨著待壓縮資料的輸入，動態地(dynamically)更新符號的出現機率及其相對應的編碼區間，並藉由新決定之符號編碼區間來編碼接下來的輸入資料。
- 由於變動機率(適應性)比固定機率(非適應性)的方式更能抓住輸入資料的特性(**也就是說，機率的估計更準確**)，因此適應性算術編碼應能比固定機率的算術編碼達到更佳的壓縮效果。

Adaptive Arithmetic Coding

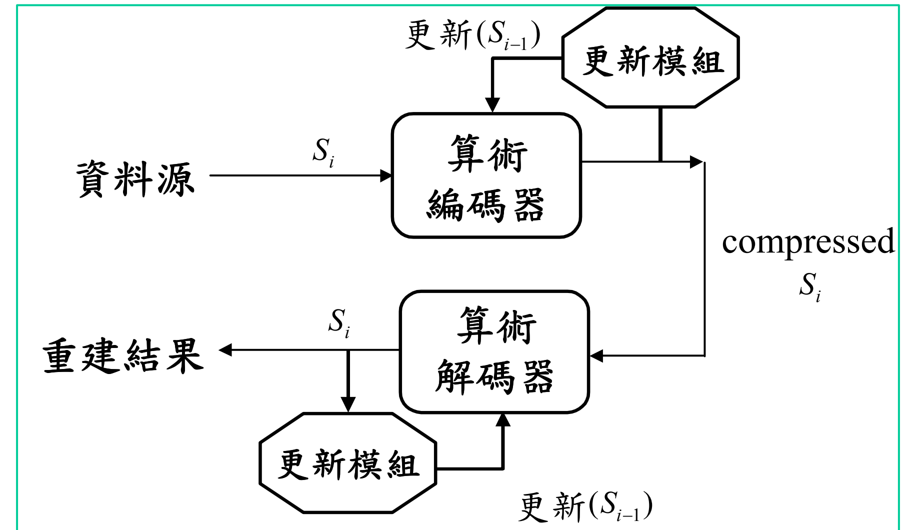
Example 5.5: (page 124, textbook)

$\boldsymbol{P(x_1)} = 0.6, \boldsymbol{P(x_2)} = 0.2, \boldsymbol{P(x_3)} = 0.2$

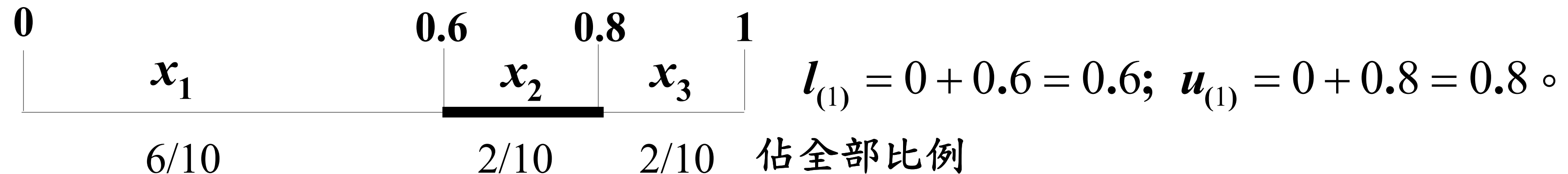
$\boldsymbol{C(x_1)} = 0.6, \boldsymbol{C(x_2)} = 0.8, \boldsymbol{C(x_3)} = 1.0$

Encode the sequence

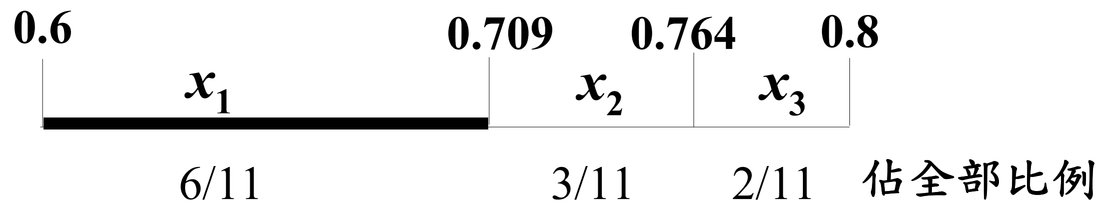
x_2, x_1, x_2, x_2



i=0, initial interval $[l_{(0)}, u_{(0)}) = [0, 1)$, encode the first input symbol x_2



i=1, the interval $[l_{(1)}, u_{(1)}) = [0.6, 0.8)$, encode the second input symbol x_1



佔全部比例

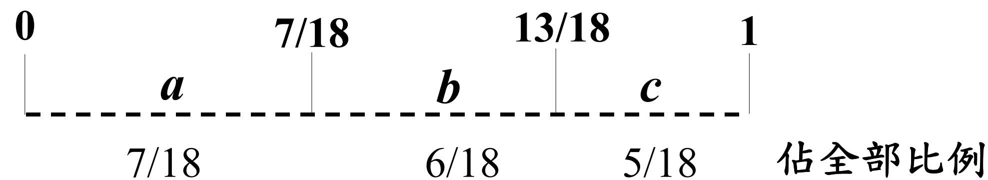
$l_{(2)} = 0.6 + 0.2 \times 0 = 0.6;\ u_{(2)} = 0.6 + 0.2 \times 0.545$ (即$6/11$) $= 0.709$ 。

n-order Markov-Arithmetic Coding

input sequence：a b a b c a b b c b a c c a a a b c (prescanning)

我們主要將探討--如何將馬可夫模式與算術編碼法相結合，藉由馬可夫模式來增進機率估計的準確性進而提昇算術編碼的壓縮效率。

zero-order Markov-Arithmetic Code



first-order Markov-Arithmetic Code

***P*(a|a)**：前一個字元為a且下一字元為a的機率(序列*S*中字母a出現7次；而字母a出現後出現a的次數則為2，因此*P*(a|a)=2/7)。

***P*(b|a)**：前一個字元為a且下一字元為b的機率(*S*中字母a出現7次；而字母a出現後出現b的次數為4次，因此*P*(b|a)=4/7)。

***P*(c|a)**：前一個字元為a且下一字元為c的機率(*P*(c|a)=1/7)。

• • •

n-order Markov-Arithmetic Coding

first-order Markov-Arithmetic Code (three symbols in the alphabet)

狀態a的符號編碼區間

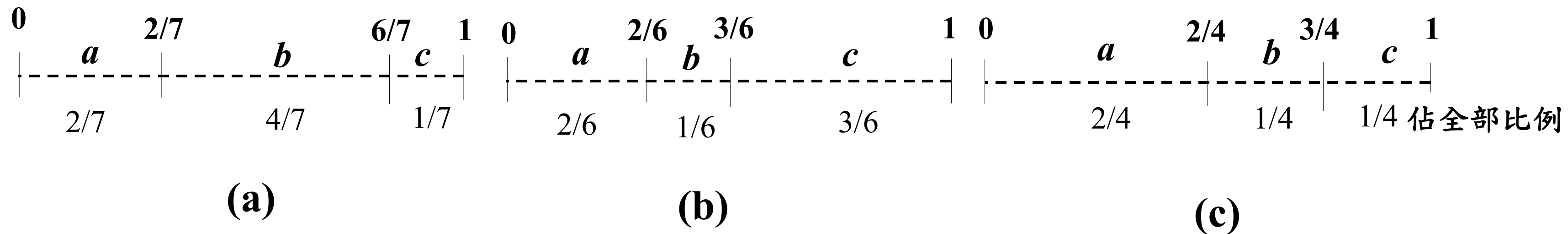
$P(a|a) = 2/7$, $P(b|a) = 4/7$,
$P(c|a) = 1/7$

狀態b的符號編碼區間

$P(a|b) = 2/6$, $P(b|b) = 1/6$,
$P(c|b) = 3/6$

狀態c的符號編碼區間

$P(a|c) = 2/4$, $P(b|c) = 1/4$,
$P(c|c) = 1/4$



(a) (b) (c)

序列: | ababca

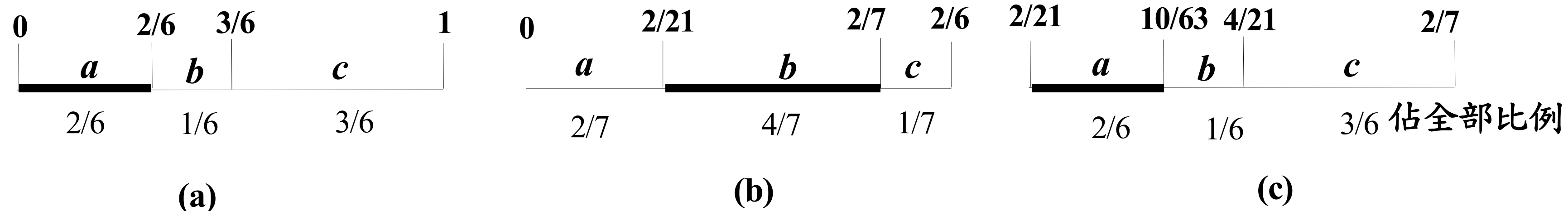
使用狀態b 的符號編碼區間

序列: a | babca

使用狀態a符號編碼區間

序列: ab | abca

使用狀態b符號編碼區間



(a) (b) (c)

n-order Markov-Adaptive-Arithmetic Coding

適應性算術編碼與非適應算術編碼最大的不同在於<u>適應性算術編碼法每編碼一個符號，就必須更新符號機率區間一次</u> 。

first-order Markov-Adaptive-Arithmetic Code

狀態a的符號編碼區間

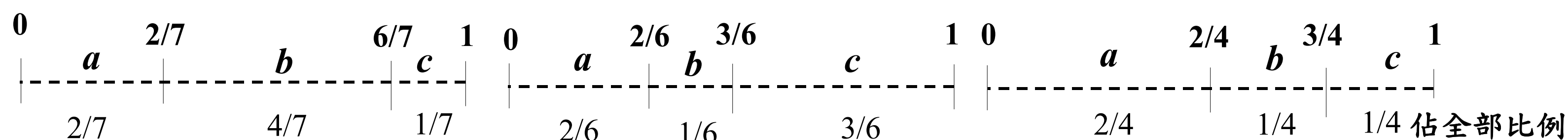
P (a|a) = 2/7, P (b|a) = 4/7, P (c|a) = 1/7

狀態b的符號編碼區間

P(a|b) = 2/6, P(b|b) = 1/6, P(c|b) = 3/6

狀態c的符號編碼區間

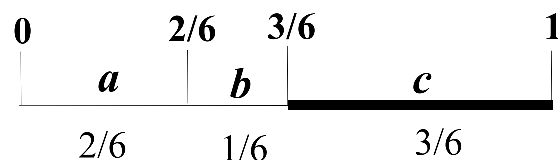
P(a|c) = 2/4, P(b|c) = 1/4, P(c|c) = 1/4



(a) (b) (c)

序列: |cbabcb

使用狀態 b 的符號編碼區間

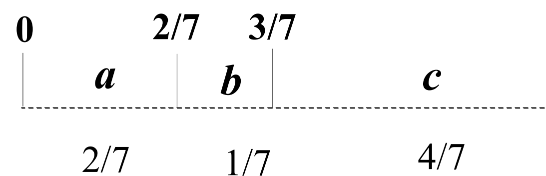


$[l_1, u_1) = [3/6, 1)$

(a)

狀態 b 的符號編碼區間更新為

P(a|b) = 2/7, P(b|b) = 1/7, P(c|b) = 4/7

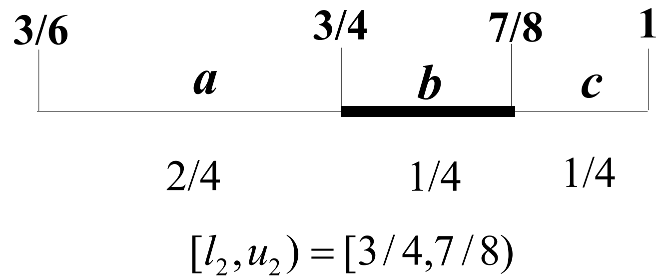


(b)

n-order Markov-Adaptive-Arithmetic Coding

序列: c | babcb

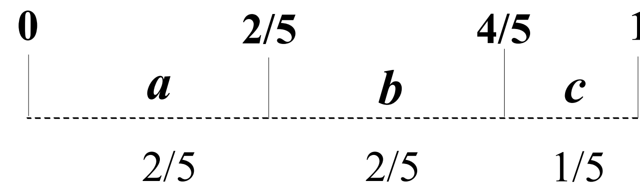
使用狀態 c 符號編碼區間



(c)

狀態 c 的符號編碼區間更新為

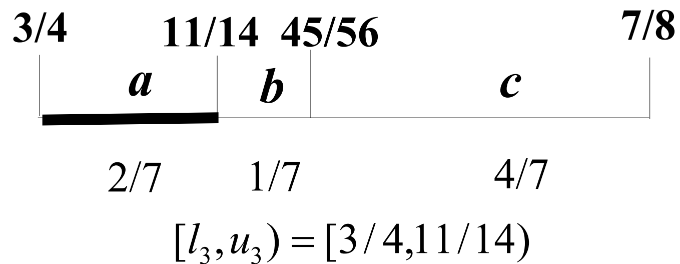
P(a|c) = 2/5, P(b|c) = 2/5,
P(c|c) = 1/5



(d)

序列: cb | abcb

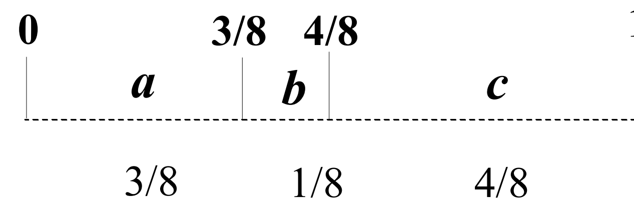
使用狀態 b 符號編碼區間



(e)

狀態 b 的符號編碼區間更新為

P(a|b) = 3/8, P(b|b) = 1/8,
P(c|b) = 4/8



(f)

Binary Adaptive Arithmetic Coding

二進制適應性算術編碼演算法：(資料源只含0與1兩個符號)

步驟一： 決定資料源中符號0的初始出現機率，令其為 $P(0)$。

步驟二： 令回合計數i=0，初始編碼區間定為 $[l_{(0)}, u_{(0)}) = [0,1)$(大於等於0但小於1)，其中 $l_{(0)}$ 代表初始編碼區間的下限，$u_{(0)}$ 則代表初始編碼區間的上限。

步驟三： i=i+1，讀入要編碼的二進制輸入符號x。

$$\boldsymbol{temp = l_{(i-1)} + (u_{(i-1)} - l_{(i-1)}) \times P(0)}$$

假如x==0

$$\boldsymbol{[l_{(i)}, u_{(i)}) = [l_{(i-1)}, temp);}$$

否則　　　　// (表示x==1)

$$\boldsymbol{[l_{(i)}, u_{(i)}) = [temp, u_{(i-1)});}$$

步驟四： 更新模組(update module)使用剛剛編碼過的符號x來更新$P(0)$的機率值。

步驟五： 重複步驟三到四，每次讀入一個二進制輸入符號，編碼後就會更新$P(0)$的機率值，一直到所有輸入符號都處理完畢為止。

Performance Comparisons

BMP			
原始檔案大小	HUFF	ARITH	A_ARITH
1162446	1037830	1032837	959522
1104102	810979	805956	757335
1154966	914227	910040	748252
1001430	905381	902319	850099
636246	553344	551290	516611
519694	451969	449955	418991

TXT			
原始檔案大小	HUFF	ARITH	A_ARITH
34837	21980	21761	20921
9909	7841	7798	6944
51540	37266	37082	35914
215525	153307	152404	149761
87489	67343	67083	65224
309902	257843	256864	251812

Performance Comparisons

C PROGRAM			
原始檔案大小	HUFF	ARITH	A_ARITH
40195	27435	27238	26426
4644	3885	3865	2988
6712	5402	5361	4494
4230	3720	3700	2816
18978	13509	13406	12578
3219	3080	3067	2178

OBJ			
原始檔案大小	HUFF	ARITH	A_ARITH
88308	55624	55270	51663
13343	5750	5438	4557
27944	6970	5108	4128
32726	10237	8861	7680
142620	117130	116753	110769
29205	18967	18756	17704