Chapter 5 VARIABLE-LENGTH CODING

---- Information Theory Results (II)

Some Fundamental Results

- **Coding an Information Source**
  - Consider an information source, represented by **a *source alphabet*** S.

    $$S = \{s_1, s_2, \cdots, s_m\} \quad , \qquad \textbf{(5.1)}$$

    where s_i 's are ***source symbols***.
  - Source symbol and information message.
    - Used interchangeably in the literature.
    - Let us consider an information message as a source symbol, or a combination of source symbols.

Coding an Information Source

- Denote ***code alphabet*** as *A*
  $A=\{a_1,a_2,\cdots,a_r\}$
  where a_j's are ***code symbols***.
- A ***message code*** is a sequence of code symbols that represents a given information message.
- Simplest case, a message consists of only one source symbol.
  Encoding is then a procedure to assign a ***codeword*** to the source symbol. i.e.,
  $s_i \rightarrow A_i=(a_{i1},a_{i2},\cdots,a_{ik})$,
  where codeword A_i is a string of *k* code symbols assigned to the source symbol s_i .

Coding an Information Source

- The term **message ensemble** is defined as the entire set of messages.
- A **code** (**ensemble code)** is defined as a mapping of all the possible sequences of symbols of *S* (message ensemble) into the sequences of symbols in *A*.
- In binary coding, the number of code symbols *r* is equal to 2,
  - There are only two code symbols available: the binary digits “0” and “1”.

Coding an Information Source

- **Example 5.1**
  - Consider an English article and ASCII code.
  - In this context, the source alphabet consists of all the English letters in both lower and upper cases and all the punctuation marks.
  - The code alphabet consists of the binary 1 and 0.
  - There are a total of 128 7-bit binary codewords.
  - From Table 5.1, we see that codeword assigned to capital letter A (a source symbol) is 1000001 (the codeword).

Coding an Information Source

Table 5.1 Seven-bit American standard code for information interchange (ASCII)

Bits				5	0	1	0	1	0	1	0	1
				6	0	0	1	1	0	0	1	1
1	2	3	4	7	0	0	0	0	1	1	1	1
0	0	0	0	NUL		DLE	SP	0	@	P	`	p
1	0	0	0	SOH		DC1	!	1	A	Q	a	q
0	1	0	0	STX		DC2	"	2	B	R	b	r
1	1	0	0	ETX		DC3	#	3	C	S	c	s
0	0	1	0	EOT		DC4	$	4	D	T	d	t
1	0	1	0	ENQ		NAK	%	5	E	U	e	u
0	1	1	0	ACK		SYN	&	6	F	V	f	v
1	1	1	0	BEL		ETB	'	7	G	W	g	w
0	0	0	1	BS		CAN	(	8	H	X	h	x
1	0	0	1	HT		EM	)	9	I	Y	i	y
0	1	0	1	LF		SUB	*	:	J	Z	j	z
1	1	0	1	VT		ESC	+	;	K	[	k	{
0	0	1	1	FF		FS	,	<	L	\	l	|
1	0	1	1	CR		GS	-	=	M	]	m	}
0	1	1	1	SO		RS	.	>	N	^	n	~
1	1	1	1	SI		US	/	?	O	—	o	DEL

NUL	Null, or all zeros	DC1	Device control 1
SOH	Start of heading	DC2	Device control 2
STX	Start of text	DC3	Device control 3
ETX	End of text	DC4	Device control 4
EOT	End of transmission	NAK	Negative acknowledgment
ENQ	Enquiry	SYN	Synchronous idle
ACK	Acknowledge	ETB	End of transmission block

Coding an Information Source

- **Example 5.2**
  - Table 5.2 lists what is known as the (5,2) code.
    It is a linear block code (for channel coding).
  - The source alphabet consists of four (2^2) source symbols: 00, 01, 10 and 11.
  - The code alphabet consists of the binary 1 and 0.
  - The code assigns a 5-bit codeword to each source symbol.

Table 5. 2 A (5,2) linear block code

Source Symbol	Codeword
S_1 (0 0)	0 0 0 0 0
S_2 (0 1)	1 0 1 0 0
S_3 (1 0)	0 1 1 1 1
S_4 (1 1)	1 1 0 1 1

Some Desired Characteristics

- **Block Code**
  - A code is said to be a block code if it maps each source symbol in *S* into a fixed codeword in A.
    - E.g., codes in the above two examples
- **Uniquely Decodable Code**
  - A code is uniquely decodable if it can be unambiguously decoded.

Table 5. 3 A not uniquely decodable code

Source Symbol	Codeword
s_1	0 0
s_2	1 0
s_3	0 0
s_4	1 1

Uniquely Decodable Code

- **Nonsingular Code**
  - **A block code is nonsingular if all the codewords are distinct.**

Table 5. 4 A nonsingular code

Source Symbol	Codeword
S_1	1
S_2	11
S_3	00
S_4	01

Uniquely Decodable Code

- **Example 5.4**
  - Table 5.4 shows a nonsingular code, since all four codewords are distinct.
  - If a code is not a nonsingular code, i.e., at least two codewords are identical, then the code is not uniquely decodable.
  - However, a nonsingular code does not guarantee unique decodability.
    - The code shown in Table 5.4 is such an example. It is nonsingular yet it is not uniquely decodable (because once the binary string "11" is received, we do not know if the source symbols transmitted are s_1 followed by s_1 or simply s_2 .)

Uniquely Decodable Code

- **The *n*th Extension of a Block Code**
  - The *n*th extension of a block code, which maps the source symbol $\underline{s_i}$ into the codeword $\underline{A_i}$, is a block code that maps the sequences of source symbols $\underline{s_{i1}s_{i2}\cdots s_{in}}$ into the sequences of codewords $A_{i1}A_{i2}\cdots A_{in}$.
- **A Necessary and Sufficient Condition of Block Codes' Unique Decodability**
  - A block code is uniquely decodable if and only if the *n*th extension of the code is nonsingular for every finite *n*.

Uniquely Decodable Code

Table 5. 5 The second extension of the nonsingular block code shown in Example 5.4

Source Symbol	Codeword	Source Symbol	Codeword
S_1S_1	11	S_3S_1	001
S_1S_2	111	S_3S_2	0011
S_1S_3	100	S_3S_3	0000
S_1S_4	101	S_3S_4	0001
S_2S_1	111	S_4S_1	011
S_2S_2	1111	S_4S_2	0111
S_2S_3	1100	S_4S_3	0100
S_2S_4	1101	S_4S_4	0101

Instantaneous Codes

- **Definition**
  - A uniquely decodable code is said to be ***instantaneous*** if it is possible to decode each codeword in a code symbol sequence without knowing the succeeding codewords.

Table 5. 6 Three uniquely decodable codes

Source Symbol	Code 1	Code 2	Code 3
S_1	0 0	1	1
S_2	0 1	0 1	10
S_3	1 0	0 0 1	1 0 0
S_4	1 1	0 0 0 1	1 0 0 0

Instantaneous Codes

- **Example 5.6**
  - Table 5.6: three uniquely decodable codes.
  - The first one is in fact a two-bit NBC. When decoding, we can immediately tell which source symbols are transmitted since each codeword has the same length.
  - In the second code, code symbol "1" functions like a comma. Whenever we see a "1", we know it is the end of the codeword.
  - The third code is different from the previous two codes in that if we see a "10" string we are not sure if it corresponds to s_2 until we see a succeeding "1". Specifically, if the next code symbol is "0", we still cannot tell if it is s_3 since the next one may be "0" (hence s_4) or "1" (hence s_3). In this example, the next "1" belongs to the succeeding codeword. Therefore we see that code 3 is uniquely decodable. It is not instantaneous, however.

Instantaneous Codes

- **Definition of the *j*th Prefix**
  - Assume a codeword $A_i = a_{i1}a_{i2}\cdots a_{ik}$. Then the sequences of code symbols $a_{i1}a_{i2}\cdots a_{ij}$ with $1 \le j \le k$ is the *j*th order prefix of the codeword A_i .
- **Example 5.7**
  - If a codeword is 11001, it has the following five prefixes: 11001, 1100, 110, 11, 1.
  - The first order prefix is 1, while the fifth order prefix is 11001.

Instantaneous Codes

- **A Necessary and Sufficient Condition for Being Instantaneous Codes**
  - A code is instantaneous if and only if no codeword is a prefix of some other codeword.
  - This condition is called the ***prefix condition***. Hence, the instantaneous code is also called the prefix condition code or sometimes simply the prefix code.
  - In many applications, we need a block code that is nonsingular, uniquely decodable, and instantaneous.

Compact Code

- A uniquely decodable code is said to be ***compact*** if its average length is the minimum among all other uniquely decodable codes based on the same source alphabet *S* and code alphabet *A*.
- A compact code is also referred to as a ***minimum-redundancy*** code, or an ***optimum*** code.

Discrete Memoryless Sources

- The simplest model of an information source.
- In this model, the symbols generated by the source are **independent** of each other. That is, the source is memoryless or it has a zero-memory.

Discrete Memoryless Sources

- Consider the information source $S = \{s_1, s_2, \cdots, s_m\}$ as a discrete memoryless source.
  - The occurrence probabilities of the source symbols can be denoted by $p(s_1), p(s_2), \cdots, p(s_m)$
  - The lengths of the codewords are denoted by $l_1, l_2, \cdots, l_m$
  - The average length of the code is then equal to

$$L_{avg} = \sum_{i=1}^{m} l_i p(s_i) \qquad (5.4)$$

Extensions of a Discrete Memoryless Source

- Instead of coding each source symbol in a discrete source alphabet, it is often useful to code blocks of symbols.
- If *n* symbols are grouped into a block, then there are a total of m^n blocks. Each block is considered as a new source symbol.
- These m^n blocks thus form an information source alphabet, called the ***n*th extension of the source S**, denoted by S^n

Entropy

- Let each block be denoted by β_i and

$$\beta_i = (s_{i1}, s_{i2}, \cdots, s_{in}) \quad . \qquad (5.\ 6)$$

- Because of the memoryless assumption, we have

$$p(\beta_i) = \prod_{j=1}^{n} p(s_{ij}) \quad . \qquad (5.\ 7)$$

- It can be proved that (exercise)

$$H(S^n) = n \cdot H(S) \quad . \qquad (5.\ 8)$$

Noiseless Source Coding Theorem

- For a discrete zero-memory information source *S*, Shannon's noiseless coding theorem can be expressed as

$$H(S) \le L_{avg} < H(S)+1 \quad ; \qquad (5.9)$$

i.e., there exists a variable-length code whose average length is bounded below by the entropy of the source and bounded above by the entropy plus 1.

- Since the *n*th extension of the source alphabet, S^n, is itself a discrete memoryless source, we can apply the above result to it.

$$H(S^n) \le L_{avg}^n < H(S^n)+1, \qquad (5.10)$$

where L_{avg}^n is the average codeword length of a variable-length code for the S^n.

Noiseless Source Coding Theorem

- Since $H(S^n) = nH(S)$ and $L_{avg}^n = nL_{avg}$, we have

$$H(S) \le L_{avg} < H(S) + \frac{1}{n} . \qquad (5.\ 11)$$

- Therefore, when coding blocks of *n* source symbols, the noiseless source coding theory states that for an arbitrary positive number ε, there is a variable-length code that satisfies

$$H(S) \le L_{avg} < H(S) + \varepsilon \qquad (5.\ 12)$$

- To make ε arbitrarily small, we have to make the block size *n* large enough
  - Implies high complexity, large memory, and long delay.

Huffman Codes

- In many cases, we need a direct encoding method that is optimum and instantaneous (hence uniquely decodable) for an information source with finite source symbols in source alphabet *S*.
- Huffman code is the first such optimum code [Huffman'52].
  - Most frequently used today.
  - It can be used for r-ary encoding as *r>2*.
  - For notational brevity, however, we discuss only the binary case here.

Rules for Optimum Instantaneous Codes

- Consider an information source:

$$S = (s_1, s_2, \cdots, s_m) \quad . \qquad (5.\ 13)$$

- WLOG, assume the occurrence probabilities of the source symbols are as follows:

$$p(s_1) \geq p(s_2) \geq \cdots \geq p(s_{m-1}) \geq p(s_m) \qquad (5.\ 14)$$

- Since we are seeking the optimum code for *S*, the lengths of codewords assigned to the source symbols should be

$$l_1 \leq l_2 \leq \cdots \leq l_{m-1} \leq l_m \quad . \qquad (5.\ 15)$$

Rules for Optimum Instantaneous Codes

- Based on the requirements of the optimum and instantaneous code, Huffman derived the following rules (restrictions):
  - $l_1 \le l_2 \le \cdots \le l_{m-1} = l_m$. (5. 16)
  - The codewords of the two least probable source symbols should be the same except for their last bits.
  - Each possible sequence of length $l_m - 1$ bits must be used either as a codeword or must have one of its prefixes used as a codeword.

Huffman Coding Algorithm

- Based on these three rules, the two least probable source symbols have equal-length codewords.
- These two codewords are identical except for the last bits, with binary 0 and 1, respectively.
- Therefore, these two source symbols can be combined to form a single new symbol.
  - Its occurrence probability is the sum of the two source symbols, i.e., $p(s_{m-1}) + p(s_m)$
  - Its codeword is the common prefix of order $l_m - 1$ of the two codewords assigned to s_m and s_{m-1}, respectively.

Huffman Coding Algorithm

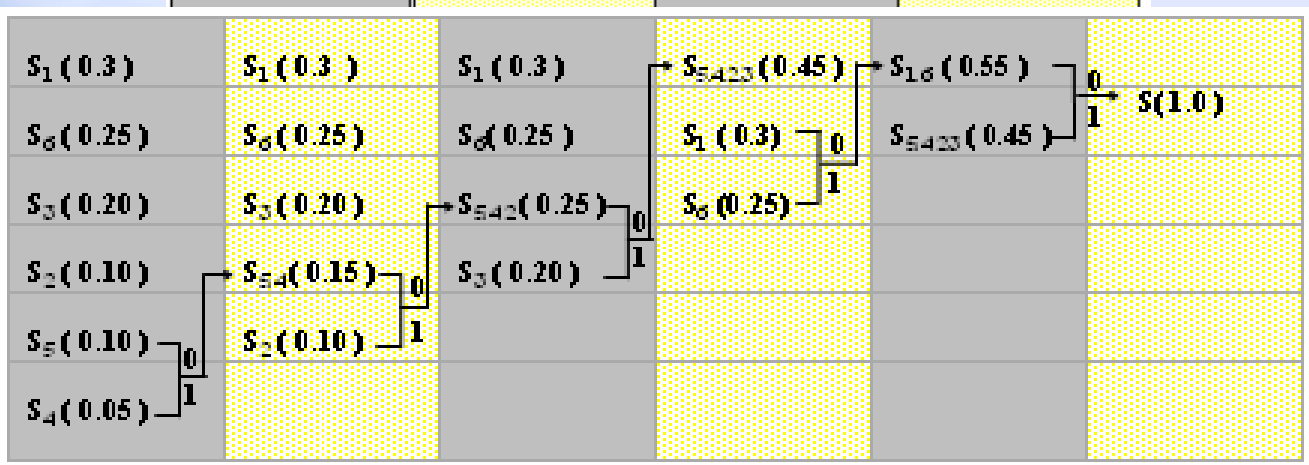
- The new set of source symbols thus generated is referred to as the first auxiliary source alphabet, which is one source symbol less than the original source alphabet.
- In the first auxiliary source alphabet, we can rearrange the source symbols according to a nonincreasing order of their occurrence probabilities.
- The same procedure can be applied to this newly created source alphabet.
- The second auxiliary source alphabet will again have one source symbol less than the first auxiliary source alphabet.
- This is repeated until we form a single source symbol with a probability of 1.
- Start from the source symbol in the last auxiliary source alphabet and trace back to each source symbol in original source alphabet to find the codewords.

Huffman Coding Algorithm

- **Example 5.9**

Table 5. 7 The source alphabet, and Huffman codes in Example 5.9

Source symbol	Occurrence probability	Codeword assigned	Length of Codeword
S_1	0.3	00	2
S_2	0.1	101	3
S_3	0.2	11	2
S_4	0.05	1001	4
S_5	0.1	1000	4
S_6	0.25	01	2



Applications

- Recall that it has been used in differential coding and transform coding.
- In TC, the magnitude of the quantized nonzero transform coefficients and the run-length of zeros in the zigzag scan are encoded using the Huffman code.
- Comment: there will be multiple Huffman codes, given a source alphabet and code alphabet.
- HW #3: Ex. 5-6

Limitations of Huffman Coding

- Huffman coding is optimum for block-encoding a source alphabet, with each source symbol having an occurrence probability.
- The average codeword length achieved by Huffman coding satisfies the following inequality [Gallagher'78].

$$H(S) \le L_{avg} < H(S) + p_{\max} + 0.086 \qquad (5.22)$$

$p_{\max}$: the maximum occurrence probability in the set of the source symbols.

- In the case where the probability distribution among source symbols is **skewed**, the upper bound may be large, implying that the coding redundancy may not be small.

Limitations of Huffman Coding

- An extreme situation. There are only two source symbols. One has a very small probability, while the other has a very large probability (very close to 1).
  - Entropy of source alphabet is close to 0 since the uncertainty is very small.
  - Using Huffman coding, however, we need two bits: one for each.
    - average codeword length is 1
    - redundancy $\eta \approx 1$
    - This agrees with Equation 5.22.
    - This inefficiency is due to the fact that Huffman coding always encodes a source symbol with an integer number of bits.

Limitations of Huffman Coding

- The fundamental idea behind Huffman coding is **block coding**.
  - That is, some codeword having an integral number of bits is assigned to a source symbol.
  - A message may be encoded by cascading the relevant codewords. It is the ***block-based*** approach that is responsible for the limitations of Huffman codes.
- Another limitation is that when encoding a message that consists of a sequence of source symbols **the *n*th extension Huffman coding** needs to enumerate all possible sequences of source symbols having the same length, as discussed in coding the *n*th extended source alphabet. This is not computationally efficient.

Arithmetic Codes

- Quite different from Huffman coding, arithmetic coding is ***stream-based***. It overcomes the drawbacks of Huffman coding.
- A string of source symbols is encoded as a string of code symbols.
  - Free of the integral-bits-per-source-symbol restriction and more efficient.
- Arithmetic coding may reach the theoretical bound of coding efficiency specified in the noiseless source coding theorem for any information source.
- Gaining increasing popularity

Principle of Arithmetic Coding

- **Example 5.12**
  - Same source alphabet as that used in Example 5.9.
  - However, a string of source symbols $s_1 s_2 s_3 s_4 s_5 s_6$ is encoded.

Table 5.8 Source alphabet and cumulative probabilities in Example 5.12

Source symbol	Occurrence probability	Associated subintervals	CP
S_1	0.3	[0, 0.3)	0
S_2	0.1	[0.3, 0.4)	0.3
S_3	0.2	[0.4, 0.6)	0.4
S_4	0.05	[0.6, 0.65)	0.6
S_5	0.1	[0.65, 0.75)	0.65
S_6	0.25	[0.75, 1.0)	0.75

Dividing Interval [0,1) into Subintervals

- ***Cumulative probability*** (CP)
  Slightly different from that of cumulative distribution function (CDF) in probability theory.

$$CDF(s_i) = \sum_{j=1}^{i} p(s_j) \quad . \qquad (5.\ 23)$$

$$CP(s_i) = \sum_{j=1}^{i-1} p(s_j) \quad , \qquad (5.\ 24)$$

where $CP(s_1) = 0$.

Dividing Interval [0,1) into Subintervals

- Each subinterval
  - Its lower end point located at $CP(s_i)$.
  - Width of each subinterval equal to probability of corresponding source symbol.
  - A subinterval can be completely defined by its lower end point and its width.
  - Alternatively, it is determined by its two end points: the lower and upper end points (sometimes also called the left and right end points).

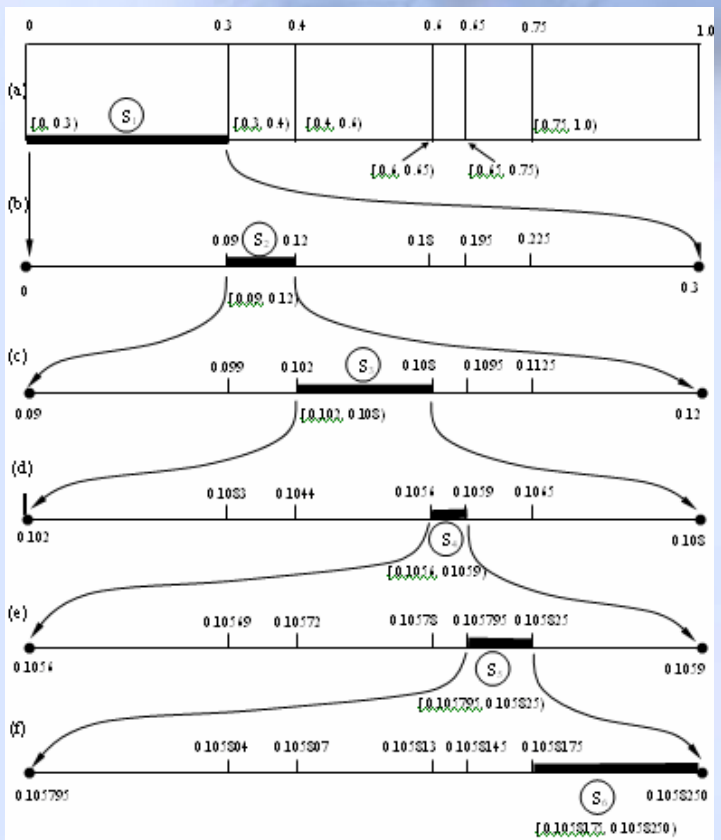


Figure 5. 2 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

Dividing Interval [0,1) into Subintervals

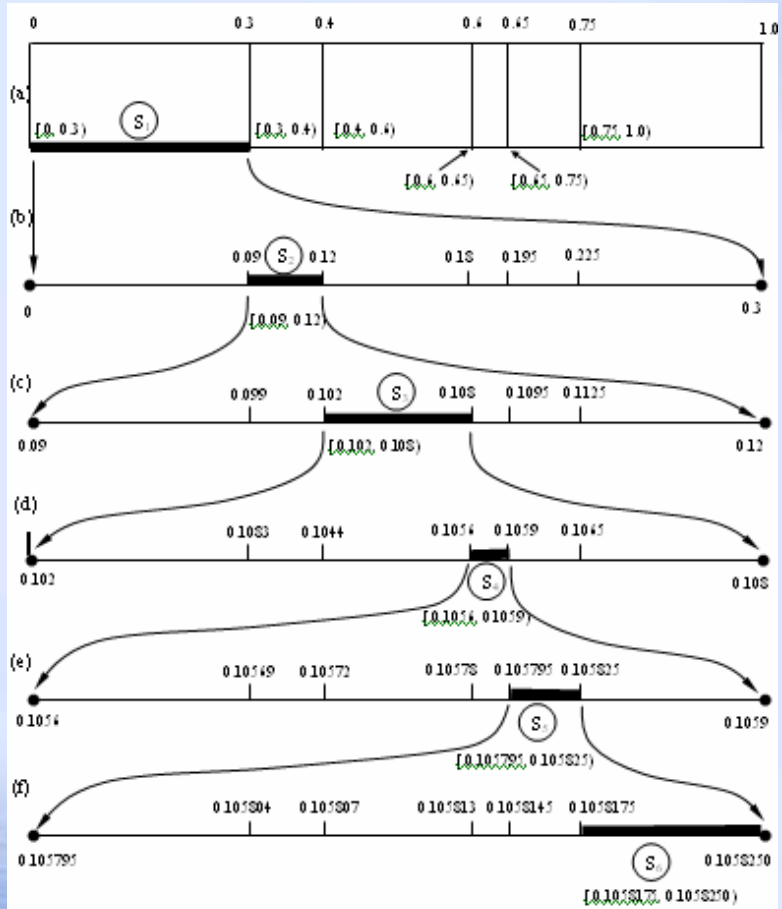


Figure 5. 2 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

Encoding

- Encoding the First Source Symbol
  - Refer to Part (a) of Figure 5.3. Since the first symbol is s_1 , we pick up its subinterval [0, 0.3). This means that any real number in the subinterval can be a pointer to the subinterval, thus representing the source symbol s_1. This can be justified by considering that all the six subintervals are disjoined.

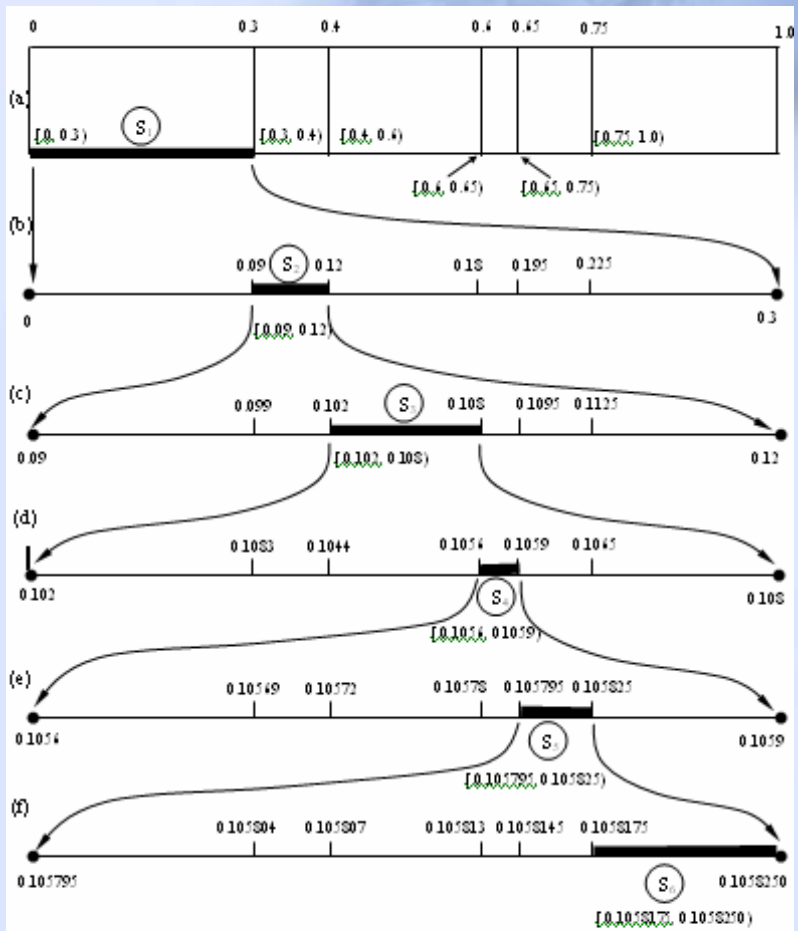


Figure 5. 2 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

Encoding

- Encoding the Second Source Symbol
  - Refer to Part (b) of Figure 5.3. We use the same procedure to divide the interval [0, 0.3) into six subintervals. Since the second symbol to be encoded is s_2 , we pick up its subinterval [0.09, 0.12).

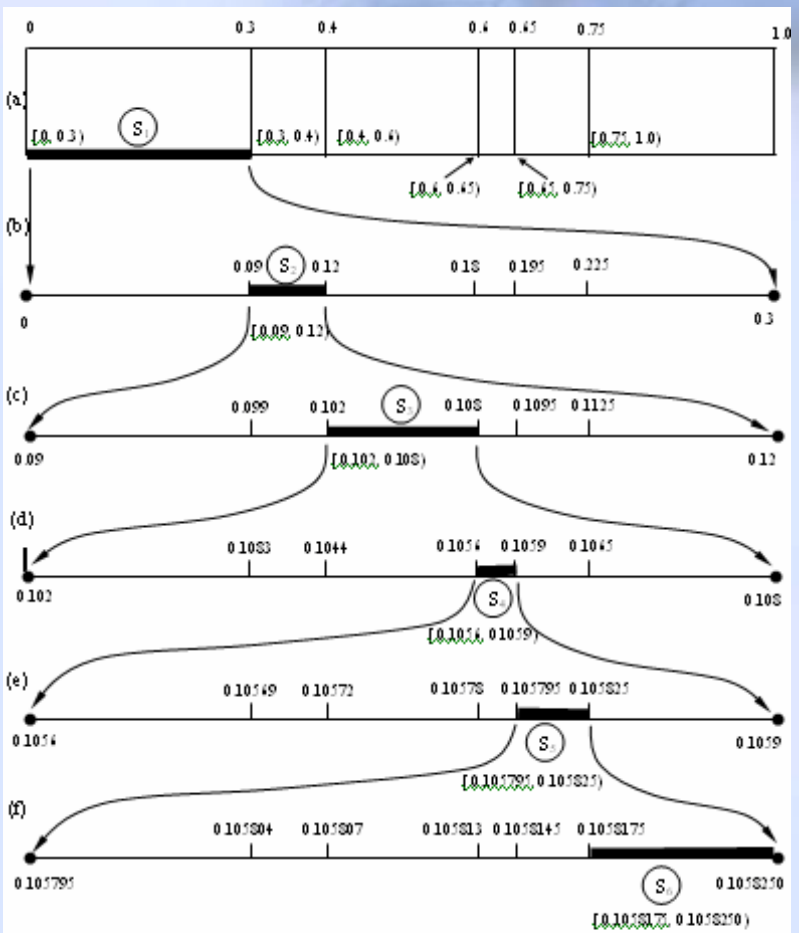


Figure 5. 2 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

Encoding

- Notice that the subintervals are recursively generated. An interval may be completely specified by its lower end point and width. Hence, the subinterval recursion is equivalent to the following two recursions: **end point recursion** and **width recursion.**
  - The lower end point recursion:

$$L_{new} = L_{current} + W_{current} \cdot CP_{new}$$

L : the lower end points W : the width

new: the new recursion *current:* the current recursion

  - The width recursion is

$$W_{new} = W_{current} \cdot p(s_i)$$

- Encoding the Third, Fourth, Fifth and Sixth Source Symbols
  - The resulting subinterval [0.1058175, 0.1058250) can represent the source symbol string $s_1 s_2 s_3 s_4 s_5 s_6$.

Decoding

- Theoretically, any real numbers in the interval can be the code string for the input symbol string since all subintervals are disjoined.
  - Often, however, the lower end of the final subinterval is used as the code string.
- Now let us examine how the decoding process is carried out with the lower end of the final subinterval.
  - The decoder knows the encoding procedure and therefore has the information contained in Part (a) of Figure 5.3.
    Since $0 < 0.1058175 < 0.3$.
    the symbol s_1 is first decoded.

Decoding

- Once the first symbol is decoded, the decoder knows the partition of subintervals shown in Part (b) of Figure 5.3. It is then determined that
  0.09<0.1058175<0.12.
  i.e., the lower end is contained in the subinterval corresponding to the symbol s_2 . As a result, s_2 is the second decoded symbol.
- The procedure is repeated until all six symbols are decoded.
- Note that a terminal symbol is necessary to inform the decoder to stop decoding.

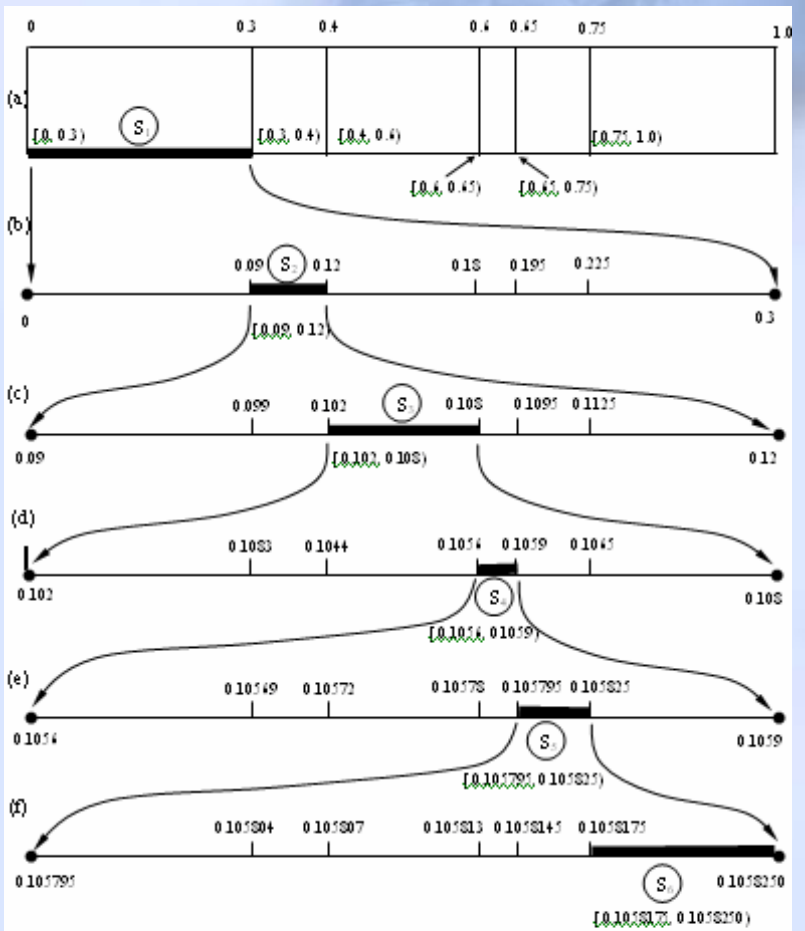


Figure 5. 2 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

Decoding

- The decoding process, however, does not need to construct Parts (b), (c), (d), (e) and (f) of Figure 5.3.
- Instead, the decoder only needs the information contained in Part (a) of Figure 5.3.
- Decoding can be split into the following three steps: ***comparison***, ***readjustment*** (subtraction), and ***scaling*** [langdon 1984].
- In summary, considering the way in which Parts (b), (c), (d), (e) and (f) of Figure 5.3 are constructed, we see that the three steps discussed above exactly "undo" what the encoding procedure has done.

Observations

- Both encoding and decoding involve only arithmetic operations (addition and multiplication in encoding, subtraction and division in decoding).
- We see that an input source symbol string $s_1 s_2 s_3 s_4 s_5 s_6$, via encoding, corresponds to a subinterval [0.1058175, 0.1058250). Any number in this interval can be used to represent the string of the source symbols.
- Arithmetic coding can be carried out in an ***incremental*** manner. That is, source symbols are fed into the encoder one by one and the final subinterval is refined continually, i.e., the code string is generated continually.
- Furthermore, it is done in a manner called ***first in first out*** (FIFO). i.e., source symbol encoded first is decoded first.

Observations

- **Precision problem**
  - The width of the final subinterval becomes smaller and smaller when the length of the source symbol string becomes larger and larger.
  - This problem prohibited arithmetic coding from practical usage for quite a long period of time.
  - Solved in the late 1970s, making arithmetic coding an increasingly important coding technique.
- It is necessary to have a termination symbol at the end of an input source symbol string.
  - Tell the decoder when to terminate decoding.

Observations

- Arithmetic coding is more efficient than the Huffman coding
  - To encode the same source symbol string, Huffman coding can be implemented in two different ways.
  - One is shown in Example 5.9, where a fixed codeword is constructed for each source symbol. Since Huffman coding is instantaneous, we can cascade the corresponding codewords to form the output, a 17-bit code string 00.101.11.1001.1000.01,
    - where the five periods are used to indicate different codewords for easy reading.

Observations

- For the same source symbol string, the final subinterval obtained by arithmetic coding is [0.1058175, 0.1058250). It is noted that the 15-bit binary decimal, 0.000110111111111, is equal to the decimal 0.1058211962, which falls into the final subinterval representing the string $s_1 s_2 s_3 s_4 s_5 s_6$.

Observations

- Another way is to use 6th extension of Huffman block code:
  - Treat each group of six source symbols as a new source symbol; then apply the Huffman coding algorithm to the 6th extension of the discrete memoryless source.
  - (the 6th extension of) Huffman coding encodes all of the $6^6 = 46656$ codewords in the 6th extension of the source alphabet.
  - This implies a high complexity in implementation and a large codebook, hence not efficient.
- Similar to the case of Huffman coding, arithmetic coding is also applicable to r-ary encoding with $r>2$.

Implementation Issues

- The **growing precision problem**.
  - This problem has been resolved and **finite precision** arithmetic is now used in arithmetic coding.
  - This advance is due to the **incremental implementation** of arithmetic coding.

Incremental Implementation

- We observe that after the third symbol, s_3, is encoded, the resultant subinterval is [0.102, 0.108).
  - That is, the two most significant decimal digits are the same and they remain the same in the encoding process.
  - We can transmit these two digits without affecting the final code string.
- After the fourth symbol s_4 is encoded, the resultant subinterval is [0.1056, 0.1059).
  - One more digit, 5, can be transmitted.
- After the sixth symbol is encoded, the final subinterval is [0.1058175, 0.1058250).
  - The cumulative output is 0.1058. Refer to Table 5.11.

Incremental Implementation

- This important observation reveals that we are able to incrementally transmit output (the code symbols) and receive input (the source symbols that need to be encoded).

- Table 5. 9 Final subintervals and cumulative output in Example 5.12

Source symbol	Final subinterval		Cumulative output
	Lower end	Upper end	
S_1	0	0.3	—
S_2	0.09	0.12	—
S_3	0.102	0.108	0.10
S_4	0.1056	0.1059	0.105
S_5	0.105795	0.105825	0.105
S_6	0.1058175	0.1058250	0.1058

Other Issues

- **Eliminating Multiplication**
- **Carry-Over Problem**

History

- The idea of encoding by using cumulative probability in some ordering, and decoding by comparison of magnitude of binary fraction was introduced in **Shannon**'s celebrated paper [Shannon'48].
- The recursive implementation of arithmetic coding was devised by **Elias** (another member in Fano's first information theory class at MIT).
  - This unpublished result was first introduced by Abramson as a note in his book on information theory and coding [Abramson'63].
- The result was further developed by Jelinek in his book on information theory [Jelinek'68].
- The growing precision problem prevented arithmetic coding from practical usage, however. The proposal of using finite precision arithmetic was made independently by **Pasco** [Pasco'76] and **Rissanen** [Rissanen'76].

History

- Practical arithmetic coding was developed by **several independent groups** [Rissanen'79, Rubin'79, Guazzo'80].
- A well-known tutorial paper on arithmetic coding appeared in [**Langdon**'84].
- The tremendous efforts made in **IBM** lead to a new form of adaptive binary arithmetic coding known as the Q-coder [Pennebaker'88].
- Based on the Q-coder, the activities of **JPEG and JBIG** combined the best features of the various existing arithmetic coders and developed the binary arithmetic coding procedure known as the QM-coder [Pennebaker'92].

Applications

- Arithmetic coding is becoming popular.
- Note that **in text and bilevel image applications** there are only two source symbols (black and while), and the occurrence probability is skewed. Therefore binary arithmetic coding achieves high coding efficiency.
- It has been successfully applied to bilevel image coding [Langdon'81] and adopted by the international standards for bilevel image compression JBIG.
- It has also been adopted by the JPEG.
- HW #4: Ex. 5-8, 5-9