CMPT 365 Multimedia Systems

Lossless Compression

Spring 2017

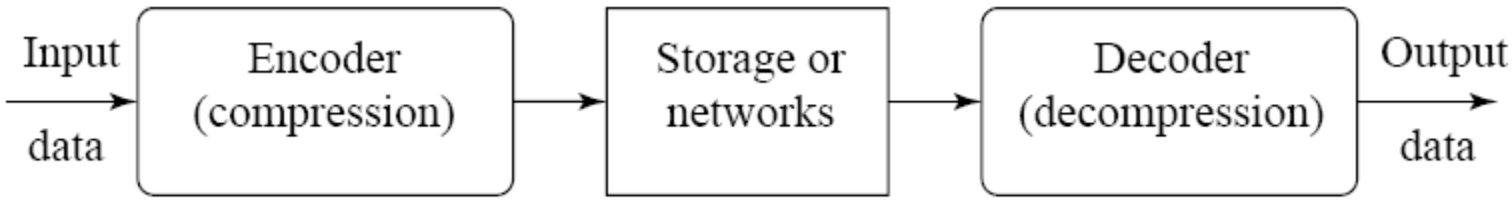
Edited from slides by Dr. Jiangchuan Liu

Outline

- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

Compression

- **Compression:** the process of coding that will effectively reduce the total number of bits needed to represent certain information.



Why Compression ?

- Multimedia data are too big
  - "A picture is worth a thousand words ! "

File Sizes for a One-minute QCIF Video Clip

Frame Rate	Frame Size	Bits / pixel	Bit-rate (bps)	File Size (Bytes)
30 frames/sec	176 x 144 pixels	12	9,123,840	**68,428,800**

Approximate file sizes for 1 sec audio

Channels	Resolution	Fs	File Size
Mono	8bit	8Khz	64Kb
Stereo	8bit	8Khz	128Kb
Mono	16bit	8Khz	128Kb
Stereo	16bit	16Khz	512Kb
Stereo	16bit	44.1Khz	1441Kb*
Stereo	24bit	44.1Khz	2116Kb

1CD 700M 70-80 mins

Lossless vs Lossy Compression

- If the compression and decompression processes induce no information loss, then the compression scheme is **lossless**; otherwise, it is **lossy**.
- **Compression ratio:**

$$compression\ ratio = \frac{B_0}{B_1}$$

B_0 – number of bits before compression

B_1 – number of bits after compression

Why is Compression possible ?

- Information Redundancy

- Question: How is “information” measured ?

Outline

- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

Self-Information

Information is related to probability
Information is a measure of uncertainty (or "surprise")

- Intuition 1:
  - I've heard this story many times vs This is the first time I hear about this story
  - Information of an event is a function of its probability:

    $i(A) = f(P(A))$. Can we find the expression of f()?

- Intuition 2:
  - Rare events have high information content
    - Water found on Mars!!!
  - Common events have low information content
    - It's raining in Vancouver.

  → Information should be a decreasing function of the probability:

  Still numerous choices of f().

- Intuition 3:
  - Information of two independent events = sum of individual information:

    If $P(AB)=P(A)P(B)$ ➔ $i(AB) = i(A) + i(B)$.

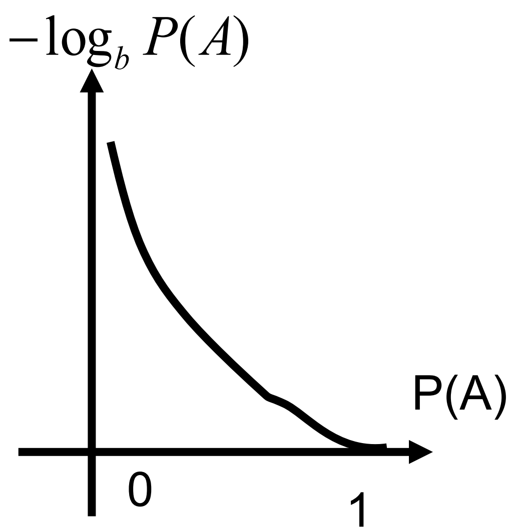
  → Only the logarithmic function satisfies these conditions.

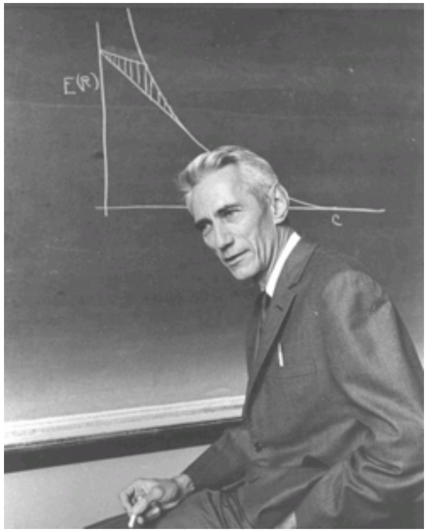
Self-information

- Shannon's Definition [1948]:
  - Self-information of an event:

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

If b = 2, unit of information is bits





Entropy

- Suppose:
  - a data source generates output sequence from a set $\{A_1, A_2, \ldots, A_N\}$
  - $P(A_i)$: Probability of A_i
- First-Order Entropy (or simply Entropy):
  - the average self-information of the data set

$$H = \sum_i -P(A_i)\log_2 P(A_i)$$

- The first-order entropy represents the minimal number of bits needed to losslessly represent one output of the source.

Example 1

- X is sampled from {a, b, c, d}
- Prob: {1/2, 1/4, 1/8, 1/8}
- Find entropy.

Example 1

- The entropy η represents the *average* amount of information contained per symbol in the source S

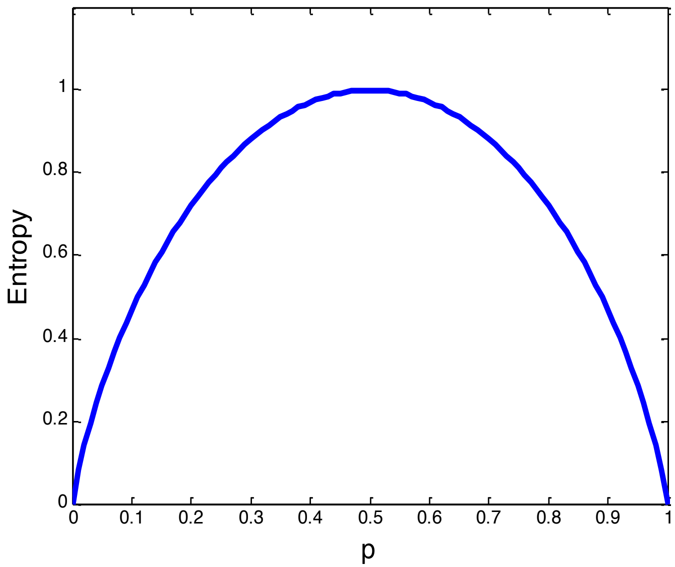
- η specifies the lower bound for the average number of bits to code each symbol in S, i.e.,

$$\eta \leq \bar{l}$$

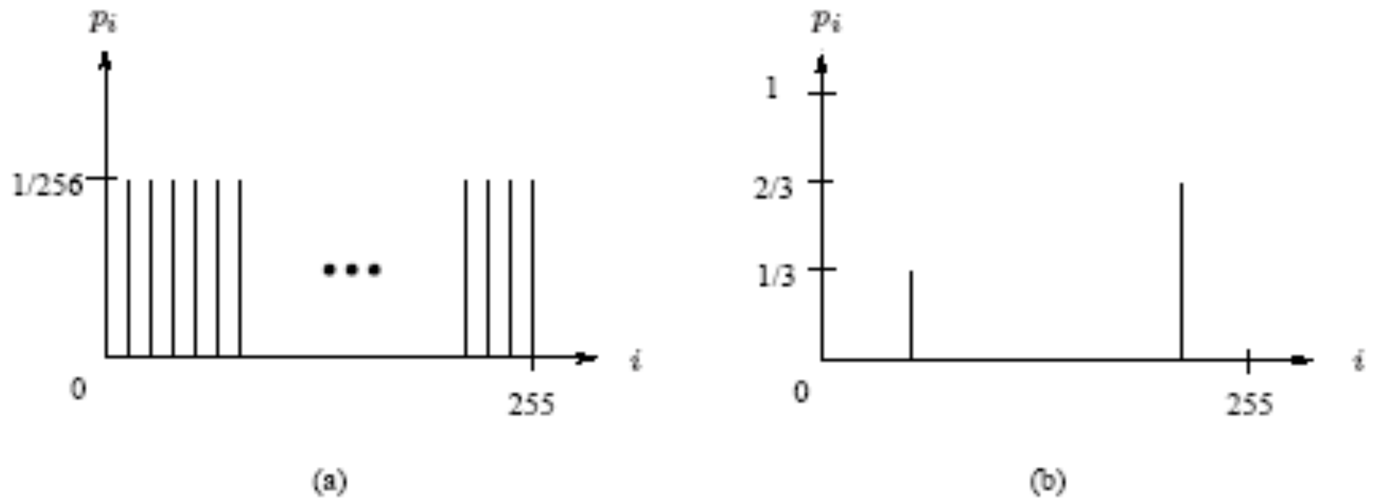
$\bar{l}$ - the average length (measured in bits) of the codewords produced by the encoder.

Example 2

- A binary source: only two possible outputs: 0, 1
  - Source output example: 000101000101110101……
  - $P(X=0) = p$, $P(X=1)= 1 – p$.
- First order entropy:
  - $H = p\,(-\log_2(p)) + (1-p)\,(-\log_2(1-p))$
  - $H = 0$ when $p = 0$ or $p = 1$
    - Fixed output, no information
  - H is largest when $p = 1/2$
    - Largest uncertainty
    - $H = 1$ bit in this case



Example 3



(a) histogram of an image with *uniform* distribution of gray-level intensities, i.e., p_i = 1/256. Entropy = $\log_2 256$=8

(b) histogram of an image with two possible values. Entropy=0.92.

Outline

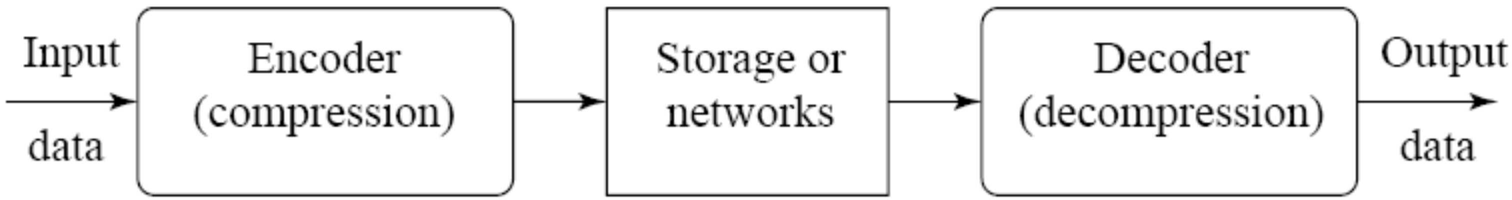
- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

Runlength Coding

- **Memoryless Source:**
  - an information source that is independently distributed.
  - i.e., the value of the current symbol does not depend on the values of the previously appeared symbols.
- Instead of assuming memoryless source, *Run-Length Coding (RLC)* exploits memory present in the information source.
- **Rationale for RLC:**
  - if the information source has the property that symbols tend to form continuous groups, then such symbol and the length of the group can be coded.

Entropy Coding

- Design the mapping from source symbols to codewords
- Goal: minimizing the average codeword length
  - Approach the entropy of the source



Example: Morse Code

- Represent English characters and numbers by different combinations of dot and dash (codewords)
- Examples:

E • I • • A • —

T — O — — — S • • • Z — — • •

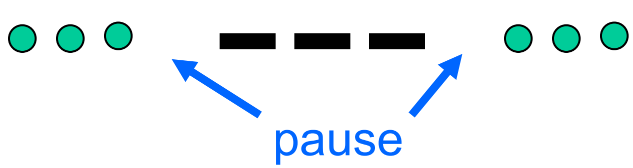
- Problem:

• • • — — — • • •   Not uniquely decodable!

  - Letters have to be separated by space,

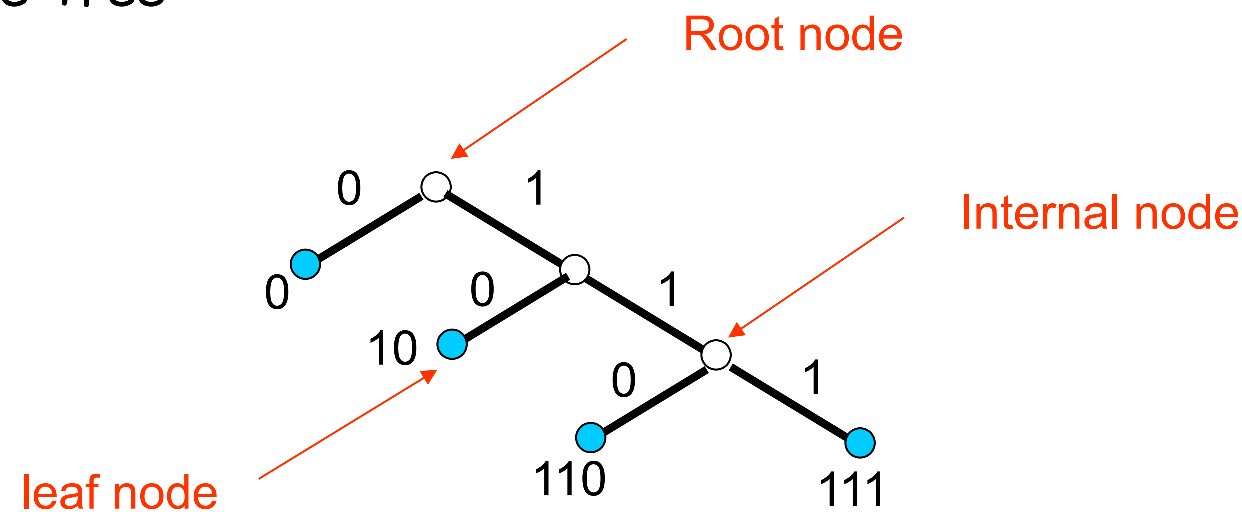
    Or paused when transmitting over radio.

    SOS:



Entropy Coding: Prefix-free Code

- No codeword is a prefix of another one.
- Can be uniquely decoded.
- Also called prefix code
- Example: 0, 10, 110, 111
- Binary Code Tree



- Prefix-free code contains leaves only.
- How to state it mathematically?

Outline

- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

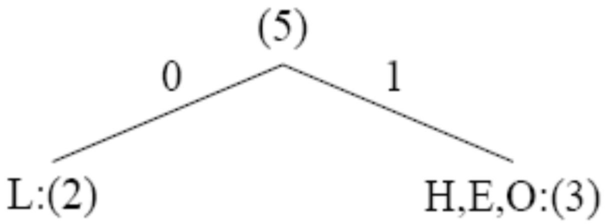
Shannon-Fano Coding

- **Shannon-Fano Algorithm** - a top-down approach
  - Sort the symbols according to the frequency count of their occurrences.
  - Recursively divide the symbols into two parts, each with approximately the same number of counts, until all parts contain only one symbol.
- **Example: coding of "HELLO"**

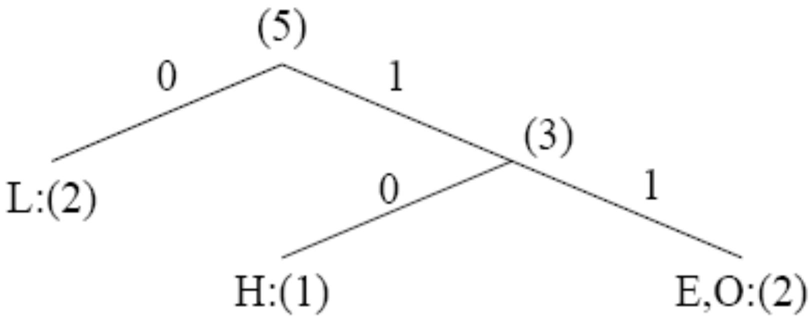
Symbol	H	E	L	O
Count	1	1	2	1

Frequency count of the symbols in "HELLO"

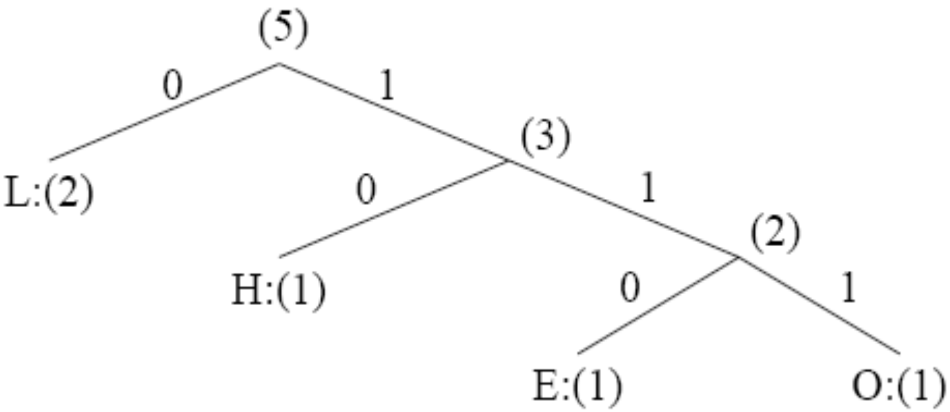
Coding Tree



(a)



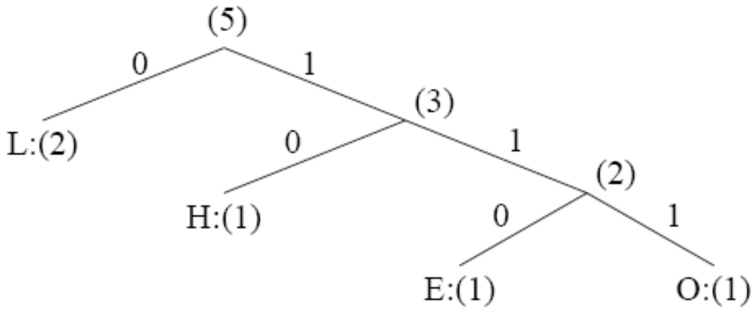
(b)



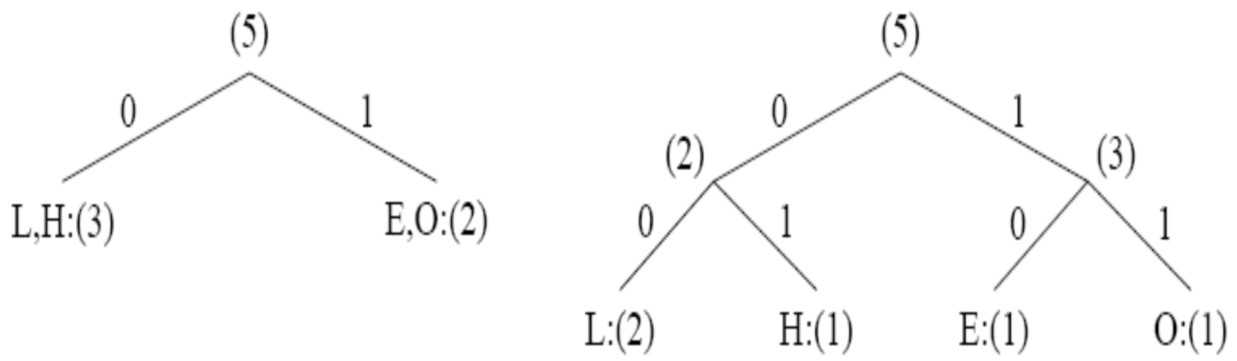
(c)

Result of Shannon-Fano Coding

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10



Another Coding Tree



(a) (b)

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
TOTAL number of bits:				10

Symbol	Count	$\log_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
TOTAL number of bits:				10

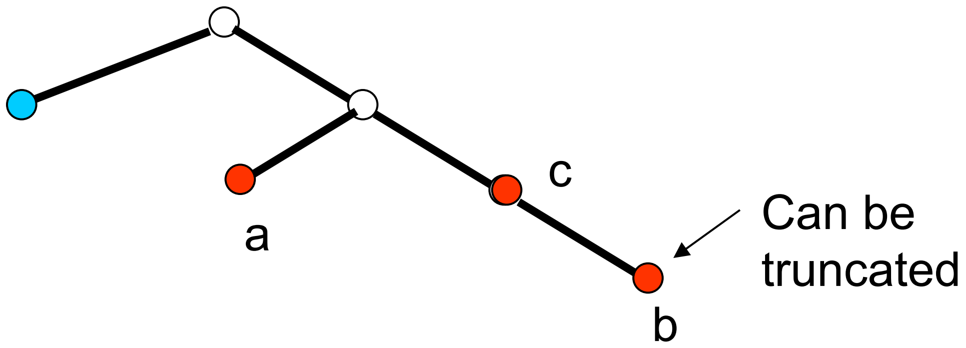
Outline

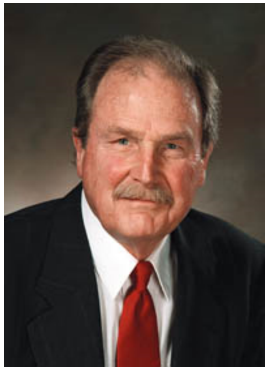
- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

Huffman Coding

- A procedure to construct optimal prefix-free code
- Result of David Huffman's term paper in 1952 when he was a PhD student at MIT

  Shannon → Fano → Huffman
- Observations:
  - Frequent symbols have short codes.
  - In an optimum prefix-free code, the two codewords that occur least frequently will have the same length.





Huffman Coding

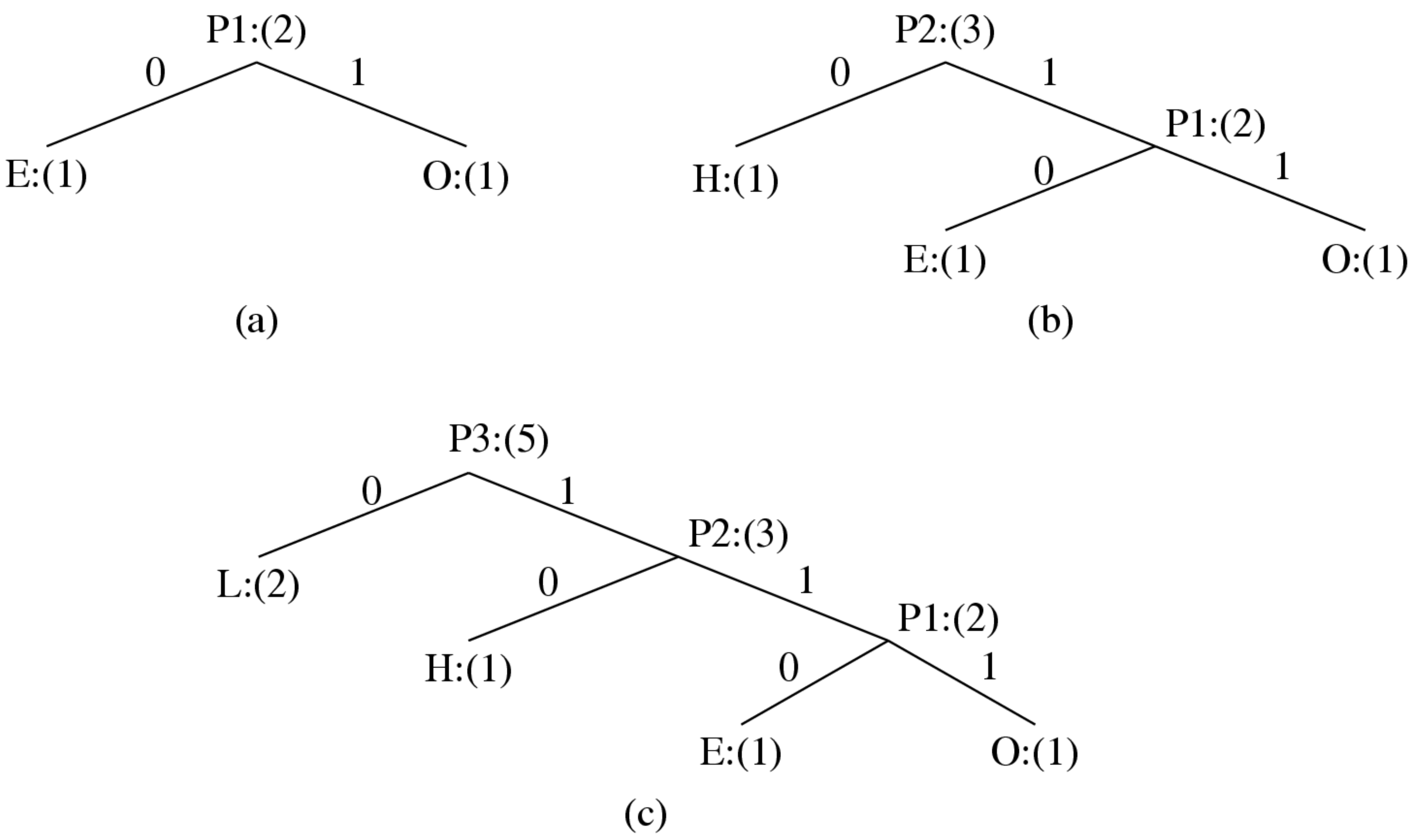
- **Human Coding** - a bottom-up approach
  - Initialization: Put all symbols on a list sorted according to their frequency counts.
    - This might not be available !
  - Repeat until the list has only one symbol left:

    (1) From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node.

    (2) Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.

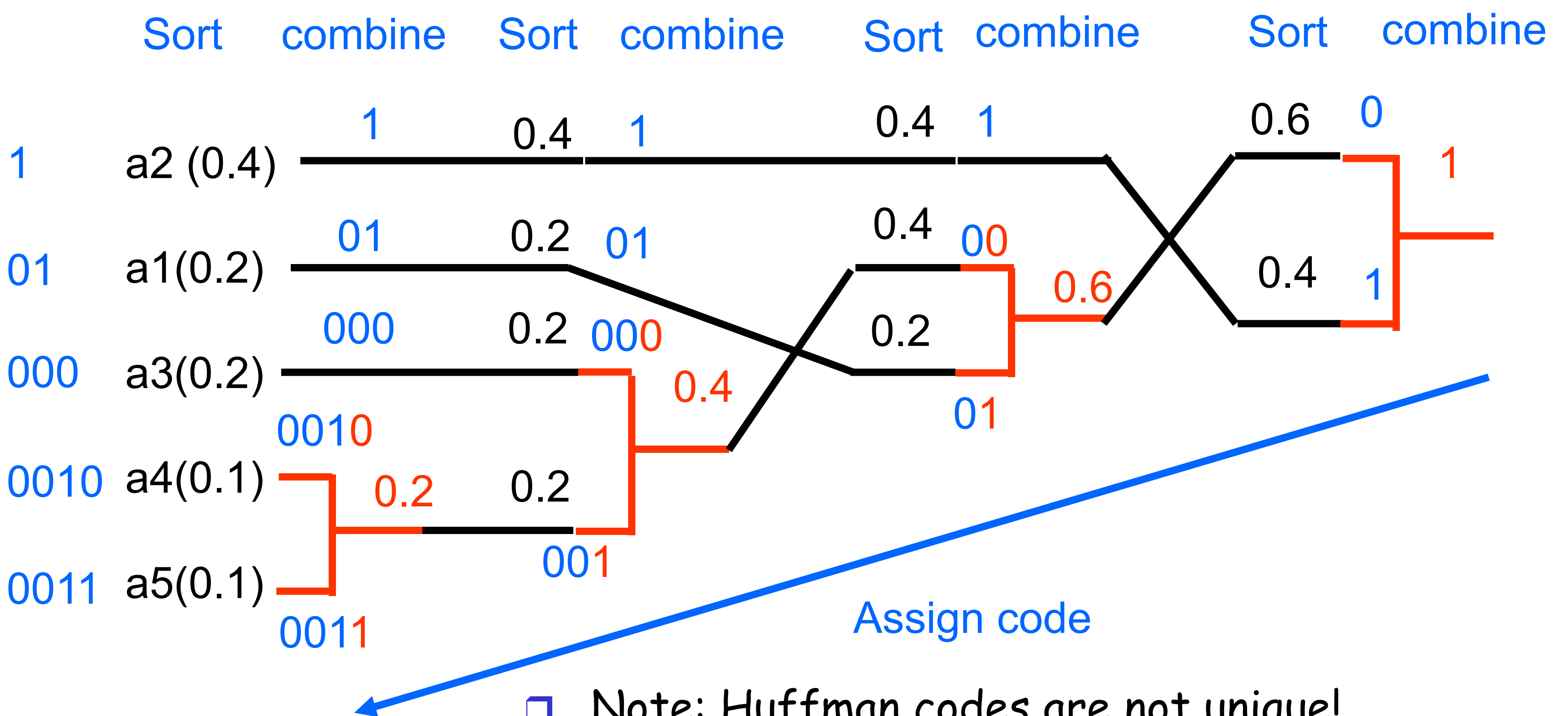
    (3) Delete the children from the list.
  - Assign a codeword for each leaf based on the path from the root.

Coding for “HELLO“



More Example

- Source alphabet A = {a1, a2, a3, a4, a5}
- Probability distribution: {0.2, 0.4, 0.2, 0.1, 0.1}



- Note: Huffman codes are not unique!
  - Labels of two branches can be arbitrary.
  - Multiple sorting orders for tied probabilities

Properties of Huffman Coding

- **Unique Prefix Property**:
  - No Human code is a prefix of any other Human code - precludes any ambiguity in decoding.
- **Optimality**:
  - *minimum redundancy code* - proved *optimal* for a given data model (i.e., a given, accurate, probability distribution) under certain conditions.
  - The two least frequent symbols will have the same length for their Human codes, differing only at the last bit.
  - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.
- Average Huffman code length for an information source *S* is strictly less than *entropy* + 1

$$\bar{l} < \eta + 1$$

Example

- Source alphabet A = {a, b, c, d, e}
- Probability distribution: {0.2, 0.4, 0.2, 0.1, 0.1}
- Code: {01, 1, 000, 0010, 0011}

- Entropy:

  H(S) = - (0.2*$\log_2$(0.2)*2 + 0.4*$\log_2$(0.4)+0.1*$\log_2$(0.1)*2)

  = 2.122 bits / symbol

- Average Huffman codeword length:

  L = 0.2*2+0.4*1+0.2*3+0.1*4+0.1*4 = 2.2 bits / symbol

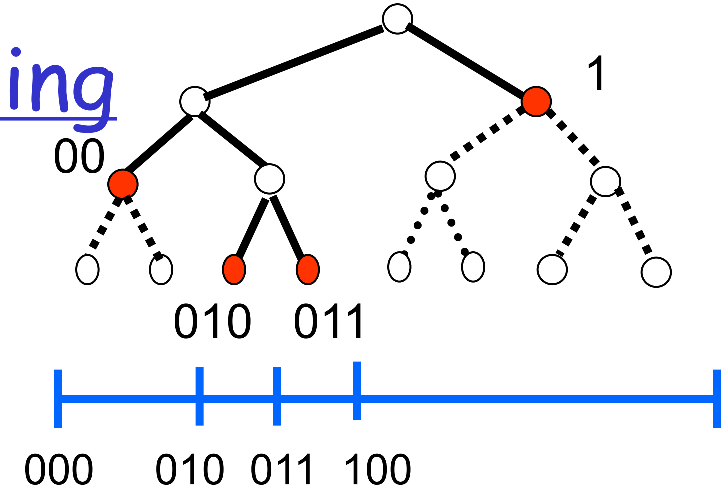
- In general: $H(S) \le L < H(S) + 1$

Huffman Decoding

- Direct Approach:
  - Read one bit, compare with all codewords…
  - Slow

- Binary tree approach:
  - Embed the Huffman table into a binary tree data structure
  - Read one bit:
    - if it's 0, go to left child.
    - If it's 1, go to right child.
    - Decode a symbol when a leaf is reached.
  - Still a bit-by-bit approach

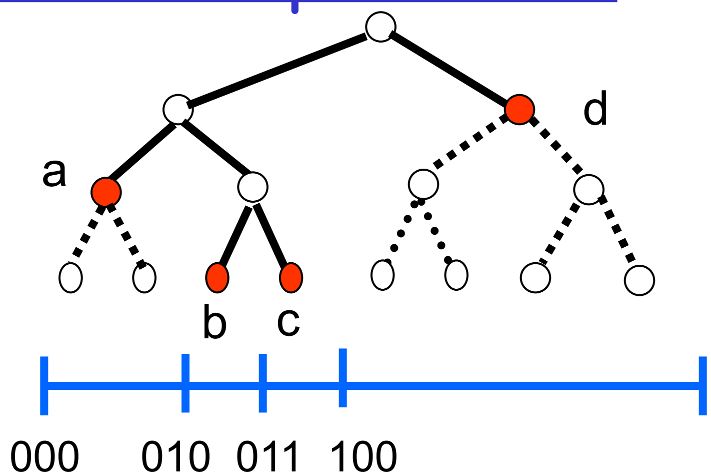
Huffman Decoding



- Table Look-up Method
  - N: # of codewords
  - L: max codeword length
  - Expand to a full tree:
    - Each Level-L node belongs to the subtree of a codeword.
    - Equivalent to dividing the range [0, 2^L] into N intervals, each corresponding to one codeword.

- bar[5]: {000, 010, 011, 100, 1000}
- Read L bits, and find which internal it belongs to.
- How to do it fast?

Table Look-up Method



a: 00
b: 010
c: 011
d: 1

```
char HuffDec[8][2] = {
    {'a', 2},
    {'a', 2},
    {'b', 3},
    {'c', 3},
    {'d', 1},
    {'d', 1},
    {'d', 1},
    {'d', 1}
};
```

```
x = ReadBits(3);
k = 0;    //# of symbols decoded
While (not EOF) {
    symbol[k++] = HuffDec[x][0];
    length = HuffDec[x][1];
    x = x << length;
    newbits = ReadBits(length);
    x = x | newbits;
  x = x & 111B;
}
```

Limitations of Huffman Code

- Need a probability distribution
  - Usually estimated from a training set
  - But the practical data could be quite different
- Hard to adapt to changing statistics
  - Must design new codes on the fly
  - Context-adaptive method still need predefined table
- Minimum codeword length is 1 bit
  - Serious penalty for high-probability symbols
  - Example: Binary source, P(0)=0.9
    - Entropy: -0.9*log2(0.9)-0.1*log2(0.1) = 0.469 bit
    - Huffman code: 0, 1 ➔ Avg. code length: 1 bit
    - More than 100% redundancy !!!
    - Joint coding is not practical for large alphabet.

Extended Huffman Code

- Code multiple symbols jointly
  - Composite symbol: $(X_1, X_2, \ldots, X_k)$
  - Alphabet increased exponentioally: N^k
- Code symbols of different meanings jointly
  - JPEG: Run-level coding
  - H.264 CAVLC: context-adaptive variable length coding
    - # of non-zero coefficients and # of trailing ones
  - Studied later

Example

- $P(X_i = 0) = P(X_i = 1) = 1/2$
  - Entropy $H(X_i) = 1$ bit / symbol
- Joint probability: $P(X_{i-1}, X_i)$
  - $P(0, 0) = 3/8$, $P(0, 1) = 1/8$
  - $P(1, 0) = 1/8$, $P(1, 1) = 3/8$
- Second order entropy:

Joint Prob P(Xi-1, Xi)

Xi-1 \ Xi	0	1
0	3/8	1/8
1	1/8	3/8

$$H(X_{i-1}, X_i) = 1.8113 \text{ bits / 2 symbols, or } 0.9056 \text{ bits / symbol}$$

- Huffman code for Xi — 0, 1
- Average code length — 1 bit / symbol
- Huffman code for (Xi-1, Xi) — 1, 00, 010, 011
- Average code length — 0.9375 bit /symbol

Consider 10 00 01 00 00 11 11 11 -- every two; non-overlapped

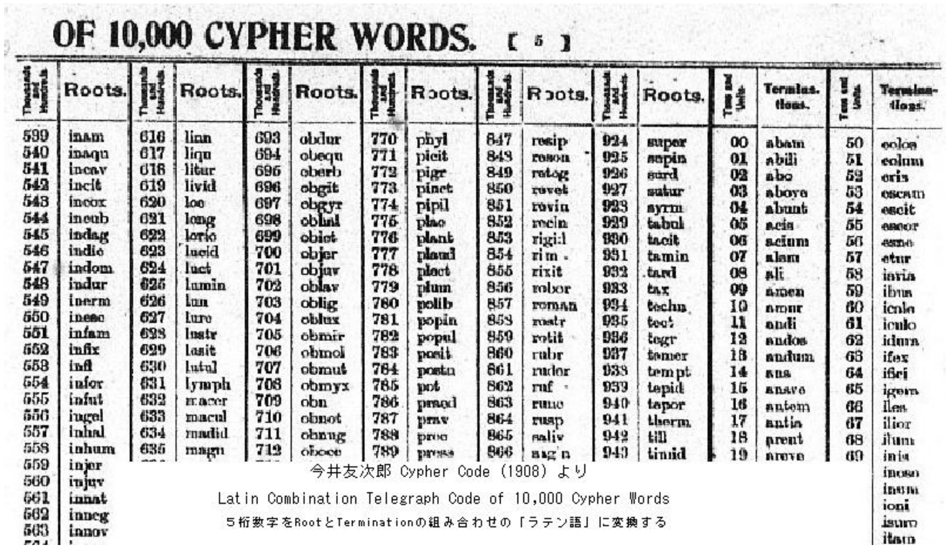
Outline

- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

LZW: Dictionary-based Coding

- LZW: Lempel-Ziv-Welch (LZ 1977, +W 1984)
  - Patent owned by Unisys http://www.unisys.com/about__unisys/lzw/
    - Expired on June 20, 2003 (Canada: July 7, 2004)
  - ARJ, PKZIP, WinZip, WinRar, Gif,
- Uses fixed-length codewords to represent variable-length strings of symbols/characters that commonly occur together
  - e.g., words in English text.
  - Encoder and decoder build up the same dictionary dynamically while receiving the data.
  - Places longer and longer repeated entries into a dictionary, and then emits the *code* for an element, rather than the string itself, if the element has already been placed in the dictionary.

LZW: Dictionary-based Coding



OF 10,000 CYPHER WORDS. [5]

Thousands and Hundreds	Roots.	Thousands and Hundreds	Roots.	Thousands and Hundreds	Roots.	Thousands and Hundreds	Roots.	Thousands and Hundreds	Roots.	Thousands and Hundreds	Roots.	Tens and Units	Terminations.	Tens and Units	Terminations.
539	inam	616	linn	693	obdur	770	phyl	847	resip	924	super	00	abam	50	eolos
540	inaqu	617	liqu	694	obequ	771	picit	848	reson	925	supin	01	abili	51	eolum
541	incav	618	litur	695	oberb	772	pigr	849	retog	926	surd	02	abo	52	eris
542	incit	619	livid	696	obgit	773	pinct	850	revet	927	satur	03	aboye	53	escam
543	incox	620	loc	697	obgyr	774	pipil	851	revin	928	syrm	04	abunt	54	escit
544	incub	621	long	698	oblat	775	plac	852	recin	929	tabul	05	acis	55	esnor
545	indag	622	loric	699	obict	776	plant	853	rigid	930	tacit	06	acium	56	esme
546	indic	623	lucid	700	objer	777	plaud	854	rim	931	tamin	07	alam	57	etur
547	indom	624	luct	701	objuv	778	plect	855	rixit	932	tard	08	ali	58	iaris
548	indur	625	lumin	702	oblav	779	plum	856	robor	933	tax	09	amen	59	ibus
549	inerm	626	lun	703	oblig	780	polib	857	roman	934	techn	10	amur	60	icula
550	inesc	627	lurc	704	oblux	781	popin	858	rostr	935	tect	11	andi	61	iculo
551	infam	628	lustr	705	obmir	782	popul	859	rotit	936	tegr	12	andos	62	idum
552	infix	629	lusit	706	obmol	783	posit	860	rubr	937	temer	13	andum	63	ifex
553	infl	630	lutul	707	obmut	784	postu	861	rudor	938	tempt	14	ans	64	ifici
554	infor	631	lymph	708	obmyx	785	pot	862	ruf	939	tepid	15	ansve	65	igem
555	infut	632	macer	709	obn	786	praed	863	runc	940	tepor	16	antem	66	iles
556	ingel	633	macul	710	obnot	787	prav	864	rusp	941	therm	17	antis	67	ilior
557	inhal	634	madid	711	obnug	788	proc	865	saliv	942	till	18	arent	68	ilum
558	inhum	635	magn	712	oboce	789	press	866	sag'n	943	timid	19	arove	69	inis
559	injer														inosa
560	injuv														insum
561	innat														ioni
562	inneg														isum
563	innov														itam

今井友次郎 Cypher Code (1908) より

Latin Combination Telegraph Code of 10,000 Cypher Words

５桁数字をRootとTerminationの組み合わせの「ラテン語」に変換する

69	Structure	Mode
70	Structure	Access Permissions
71	Structure	Alarm –Float
72	Structure	Alarm-Discrete
73	Structure	Event-Update
74	Structure	Alarm-Summary
75	Structure	Alert-Analog
76	Structure	Alert-Discrete
77	Structure	Alert-Update
78	Structure	Trend-Float
79	Structure	Trend-Discrete
80	Structure	Trend-BitString
81	Structure	FB Link
82	Structure	Simulate-Float
83	Structure	Simulate-Discrete
84	Structure	Simulate-BitString
85	Structure	Test
86	Structure	Action-Instantiate/Delete

LZW Algorithm

```
BEGIN
   s = next input character;
   while not EOF
   {
     c = next input character;

     if s + c exists in the dictionary
        s = s + c;
     else
        {
        output the code for s;
        add string s + c to the dictionary with a new code;
        s = c;
        }
   }
   output the code for s;
END
```

Example

- **LZW compression for string "ABABBABCABABBA"**
- Start with a very simple dictionary (also referred to as a "string table"), initially containing only 3 characters, with codes as follows:

code	string
1	A
2	B
3	C

- Input string is "ABABBABCABABBA"

```
BEGIN
   s = next input character;
   while not EOF
   {
     c = next input character;

     if s + c exists in the dictionary
        s = s + c;
     else
        {
        output the code for s;
        add string s + c to the
   dictionary with a new code;
        s = c;
        }
   }
   output the code for s;
END
```

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	A	4	10	ABA
A	B			
AB	B			
ABB	A	6	11	ABBA
A	EOF	1		

- Input ABABBABCABABBA
- Output codes: 1 2 4 5 2 3 4 6 1. Instead of sending 14 characters, only 9 codes need to be sent (compression ratio = 14/9 = 1.56).

LZW Decompression (simple version)

```
BEGIN
   s = NIL;
   while not EOF
    {
     k = next input code;
     entry = dictionary entry for k;
     output entry;
     if (s != NIL)
        {add string s + entry[0] to dictionary with a new code;}

        s = entry;
    }
END
```

- **Example 7.3:** LZW decompression for string "ABABBABCABABBA".
- Input codes to the decoder are 1 2 4 5 2 3 4 6 1.
- The initial string table is identical to what is used by the encoder.

- The LZW decompression algorithm then works as follows:
- **Input: 1 2 4 5 2 3 4 6 1**

```
BEGIN
   s = NIL;
   while not EOF
    {
     k = next input code;
     entry = dictionary
    entry for k;
     output entry;
     if (s != NIL)
        add string s +
    entry[0] to dictionary
    with a new code;
        s = entry;
    }
END
```

S	K	Entry/output	Code	String
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	4	AB	9	CA
AB	6	ABB	10	ABA
ABB	1	A	11	ABBA
A	EOF			

❒ Apparently, the output string is "ABABBABCABABBA", a truly lossless result!

Exceptions

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	B			
ABB	A	6	10	ABBA
A	B			
AB	B			
ABB	A			
ABBA	X	10	11	ABBAX

- Input ABABBABCABBABBAX....
- Output codes: 1 2 4 5 2 3 6 10

- Input ABABBABCABBABBAX....
- Output codes: 1 2 4 5 2 3 6 10

s	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	6	ABB	9	CA
ABB	10	???		

- Code 10 was most recently created at the encoder side, formed by a concatenation of Character, String, Character.
- Whenever the sequence of symbols to be coded is Character, String, Character, String, Character, and so on
- the encoder will create a new code to represent Character + String + Character and use it right away, before the decoder has had a chance to create it!

s	c	output	code	string
			1	A
			2	B
			3	C
A	B	1	4	AB
B	A	2	5	BA
A	B			
AB	B	4	6	ABB
B	A			
BA	B	5	7	BAB
B	C	2	8	BC
C	A	3	9	CA
A	B			
AB	B			
ABB	A	6	10	ABBA
A	B			
AB	B			
ABB	A			
ABBA	X	10	11	ABBAX

s	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	6	ABB	9	CA
ABB	10	???		

- Code 10 was most recently created at the encoder side, formed by a concatenation of Character, String, Character.
- Whenever the sequence of symbols to be coded is Character, String, Character, String, Character, and so on
- the encoder will create a new code to represent Character + String + Character and use it right away, before the decoder has had a chance to create it!

LZW Decompression (modified)

```
BEGIN
  s = NIL;
  while not EOF
  {
    k = next input code;
    entry = dictionary entry for k;

    /* exception handler */
    if (entry == NULL)
              entry = s + s[0];

    output entry;
    if (s != NIL)
       add string s+entry[0] to dictionary with a new
    code;
       s = entry;
    }
END
```

s	k	entry/output	code	string
			1	A
			2	B
			3	C
NIL	1	A		
A	2	B	4	AB
B	4	AB	5	BA
AB	5	BA	6	ABB
BA	2	B	7	BAB
B	3	C	8	BC
C	6	ABB	9	CA
ABB	10	???		

LZW Coding (Cont'd)

- In real applications, the code length l is kept in the range of $[l_0, l_{max}]$. The dictionary initially has a size of 2^{l_0}. When it is filled up, the code length will be increased by 1; this is allowed to repeat until $l = l_{max}$.
- When l_{max} is reached and the dictionary is filled up, it needs to be flushed (as in Unix *compress*, or to have the LRU (least recently used) entries removed.

Outline

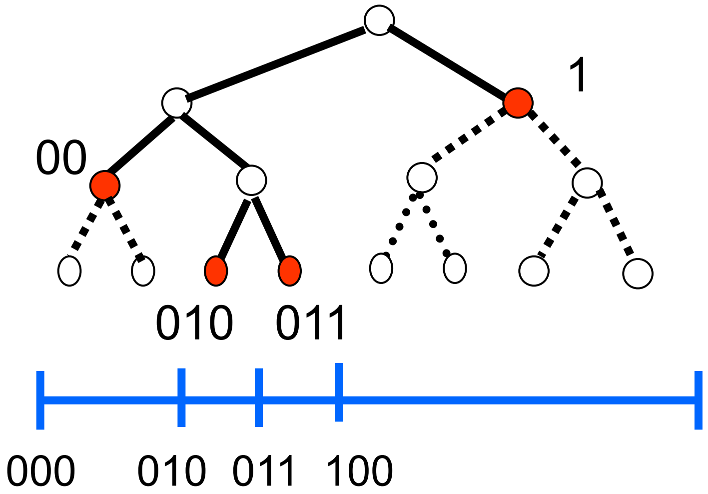
- Why compression ?
- Entropy
- Variable Length Coding
  - Shannon-Fano Coding
  - Huffman Coding
  - LZW Coding
  - Arithmetic Coding

Recall: Limitations of Huffman Code

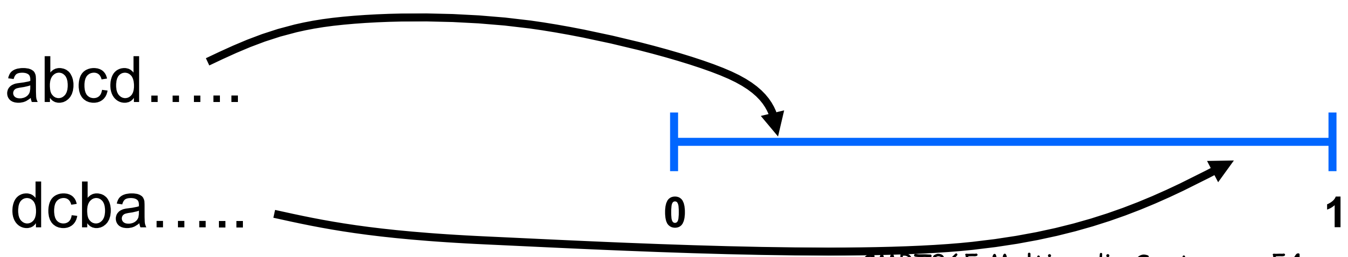
- Need a probability distribution
- Hard to adapt to changing statistics
- Minimum codeword length is 1 bit
  - Serious penalty for high-probability symbols
  - Example: Binary source, P(0)=0.9
    - Entropy: -0.9*log2(0.9)-0.1*log2(0.1) = 0.469 bit
    - Huffman code: 0, 1 ➔ Avg. code length: 1 bit
    - Joint coding is not practical for large alphabet.
- Arithmetic coding:
  - Can resolve all of these problems.
  - Code a sequence of symbols without having to generate codes for all sequences of that length.

Basic Idea

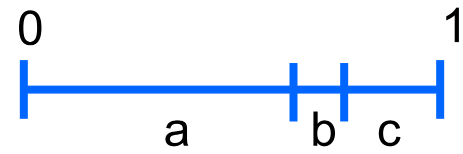
- Recall table look-up decoding of Huffman code
  - N: alphabet size
  - L: Max codeword length
  - Divide [0, 2^L] into N intervals
  - One interval for one symbol
  - Interval size is roughly proportional to symbol prob.



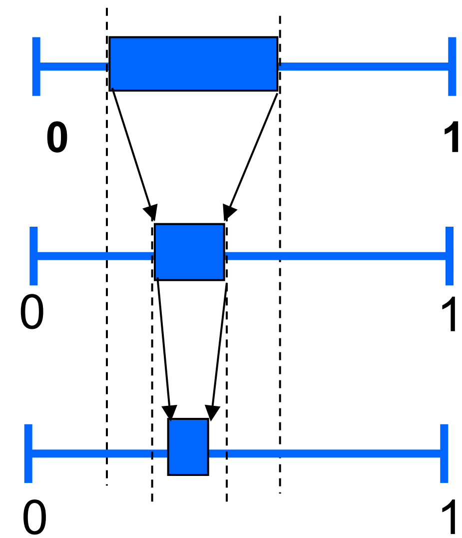
- Arithmetic coding applies this idea recursively
  - Normalizes the range [0, 2^L] to [0, 1].
  - Map a sequence to a unique tag in [0, 1).



Arithmetic Coding



- Disjoint and complete partition of the range [0, 1)
  [0, 0.8), [0.8, 0.82), [0.82, 1)
- Each interval corresponds to one symbol
- Interval size is proportional to symbol probability



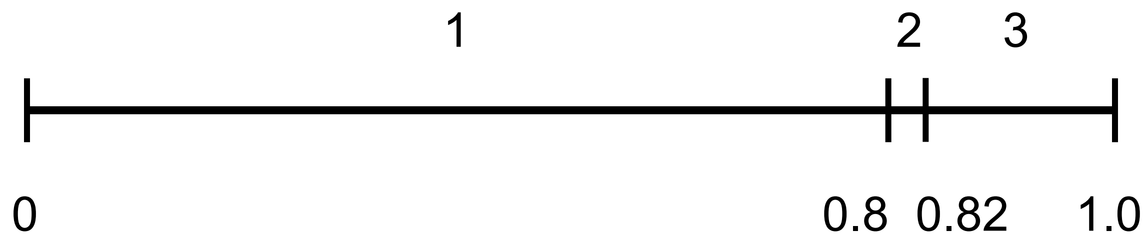
- The first symbol restricts the tag position to be in one of the intervals
- The reduced interval is partitioned recursively as more symbols are processed.
- Observation: once the tag falls into an interval, it never gets out of it

Some Questions to think about:

- Why compression is achieved this way?
- How to implement it efficiently?
- How to decode the sequence?
- Why is it better than Huffman code?

Example:

Symbol	Prob.
1	0.8
2	0.02
3	0.18

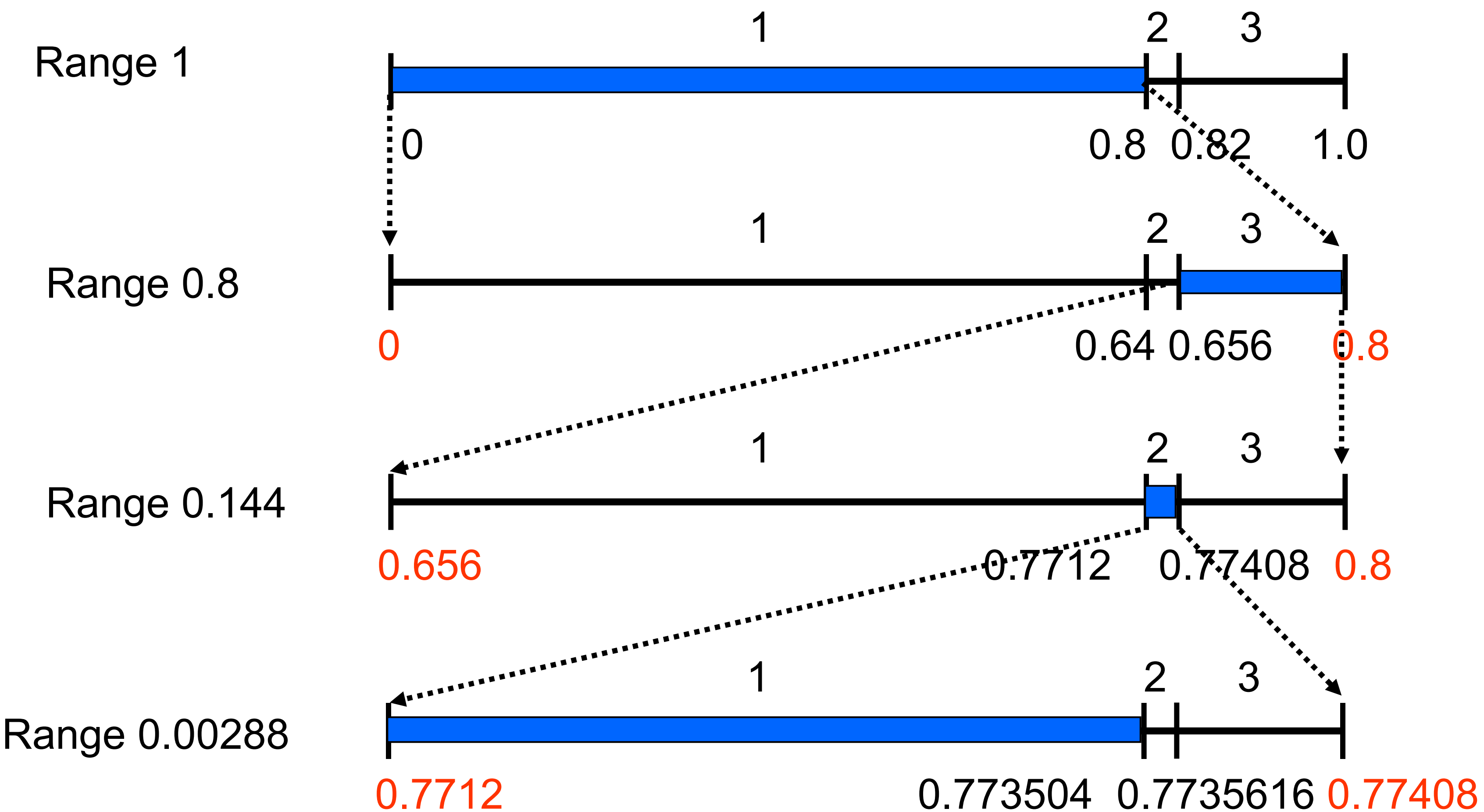


- Map to real line range [0, 1)
- Order does not matter
  - Decoder need to use the same order

- Disjoint but complete partition:
  - 1: [0, 0.8): 0, 0.799999…9
  - 2: [0.8, 0.82): 0.8, 0.819999…9
  - 3: [0.82, 1): 0.82, 0.999999…9
  - (Think about the impact to integer implementation)

Encoding

- Input sequence: "1321"



Final range: [0.7712, 0.773504): Encode 0.7712

Difficulties: 1. Shrinking of interval requires high precision for long sequence.
2. No output is generated until the entire sequence has been processed.

Encoder Pseudo Code

- Keep track of LOW, HIGH, RANGE
  - Any two are sufficient, e.g., LOW and RANGE.

```
BEGIN
low = 0.0;  high = 1.0;  range = 1.0;
while (symbol != terminator)
{
   get (symbol);
   low = low + range * Range_low(symbol);
   high = low + range *
   Range_high(symbol);
   range = high - low;
}
output a code so that low <= code < high;
END
```

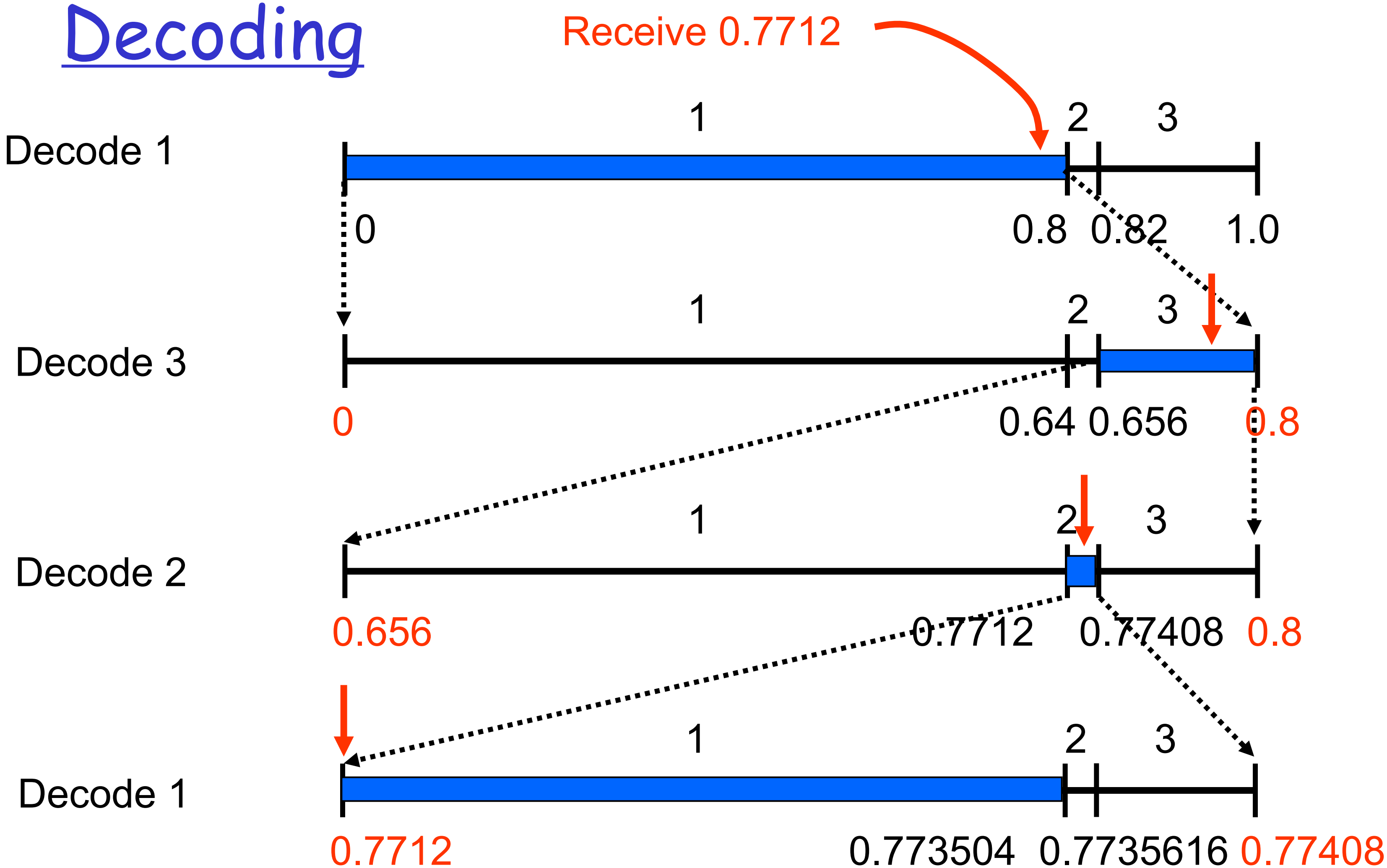
Input	HIGH	LOW	RANGE
Initial	1.0	0.0	1.0
1	0.0+1.0*0.8=0.8	0.0+1.0*0 = 0.0	0.8
3	0.0 + 0.8*1=0.8	0.0 + 0.8*0.82=0.656	0.144
2	0.656+0.144*0.82=0.77408	0.656+0.144*0.8=0.7712	0.00288
1	0.7712+0.00288*0.8=0.773504	0.7712+0.00288*0=0.7712	0.002304

Generating Codeword for Encoder

```
BEGIN
   code = 0;
   k = 1;
   while (value(code) < low)
   {
        assign 1 to the kth binary fraction bit
        if (value(code) >= high)
               replace the kth bit by 0
        k = k + 1;
   }
END
```

- The final step in Arithmetic encoding calls for the generation of a number that falls within the range [*low*, *high*). The above algorithm will ensure that the shortest binary codeword is found.

Decoding

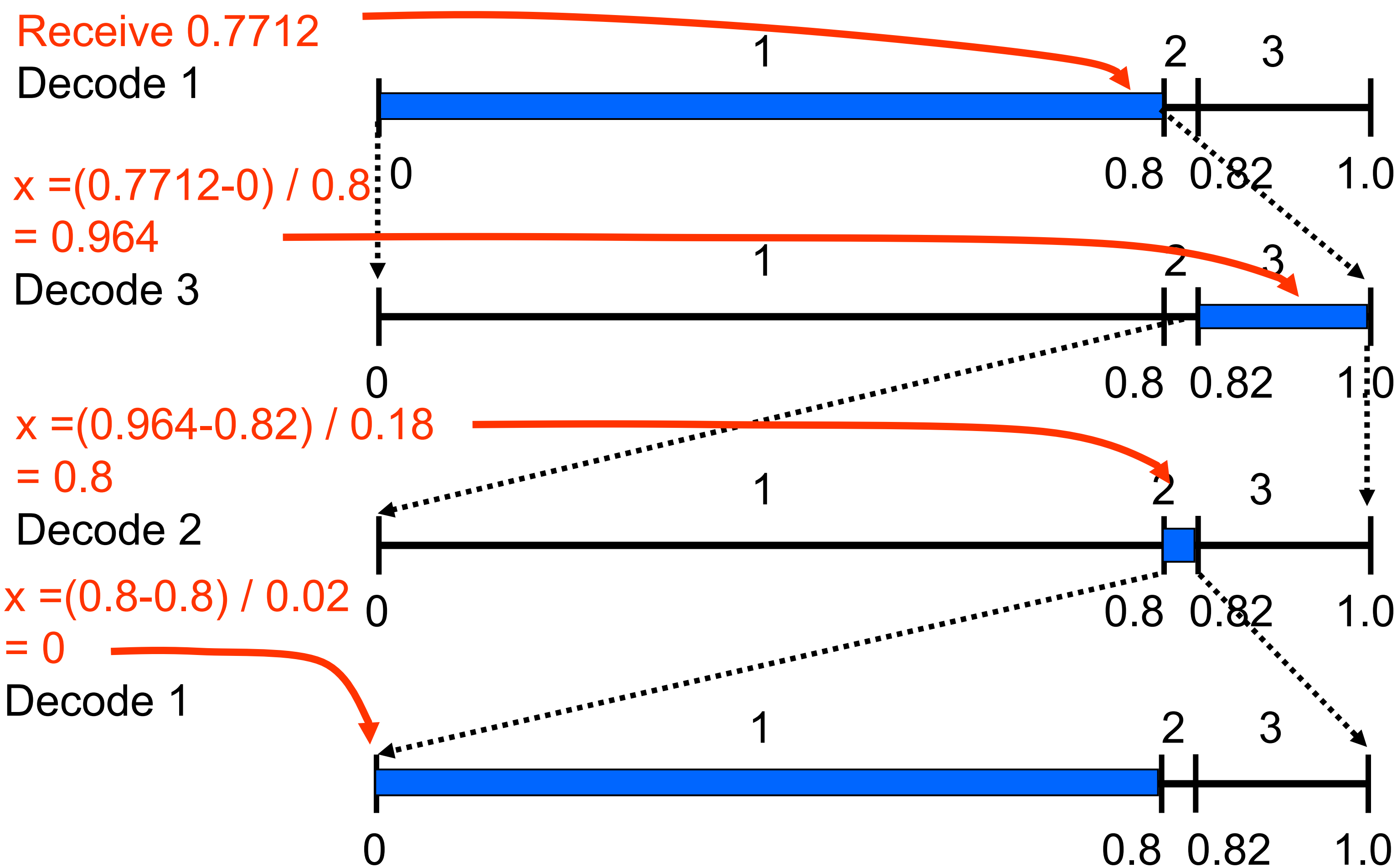


Drawback: need to recalculate all thresholds each time.

Simplified Decoding

- Normalize RANGE to [0, 1) each time
- No need to recalculate the thresholds.

$$x \leftarrow \frac{x - low}{range}$$



Decoder Pseudo Code

```
BEGIN
get binary code and convert to
decimal value = value(code);
DO
{
   find a symbol s so that
        Range_low(s) <= value < Range_high(s);
   output s;
   low = Rang_low(s);
   high = Range_high(s);
   range = high - low;
   value = [value - low] / range;
}
UNTIL symbol s is a terminator
END
```

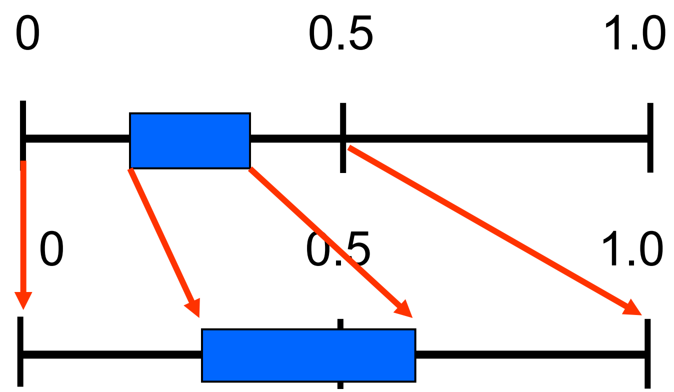
Scaling and Incremental Coding

- Problems of Previous examples:
  - Need high precision
  - No output is generated until the entire sequence is encoded

- Key Observation:
  - As the RANGE reduces, many MSB's of LOW and HIGH become identical:
    - Example: Binary form of 0.7712 and 0.773504:
      0.11000100.., 0.11000111..,
  - We can output identical MSB's and re-scale the rest:
    - ➔ Incremental encoding
  - This also allows us to achieve infinite precision with finite-precision integers.

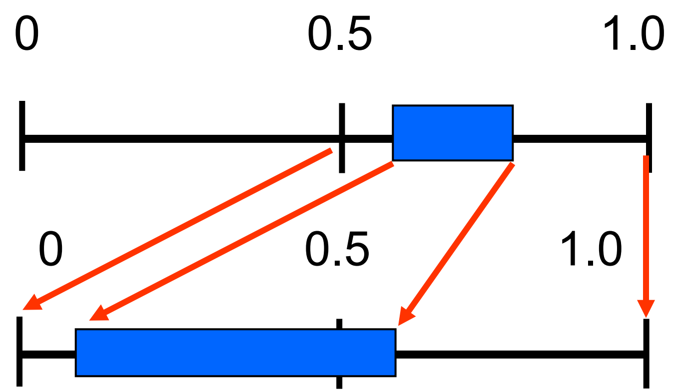
E1 and E2 Scaling

- E1: [LOW HIGH) in [0, 0.5)
  - LOW: 0.0xxxxxxx (binary),
  - HIGH: 0.0xxxxxxx.



- Output 0, then shift left by 1 bit
  - [0, 0.5) →[0, 1): E1(x) = 2 x

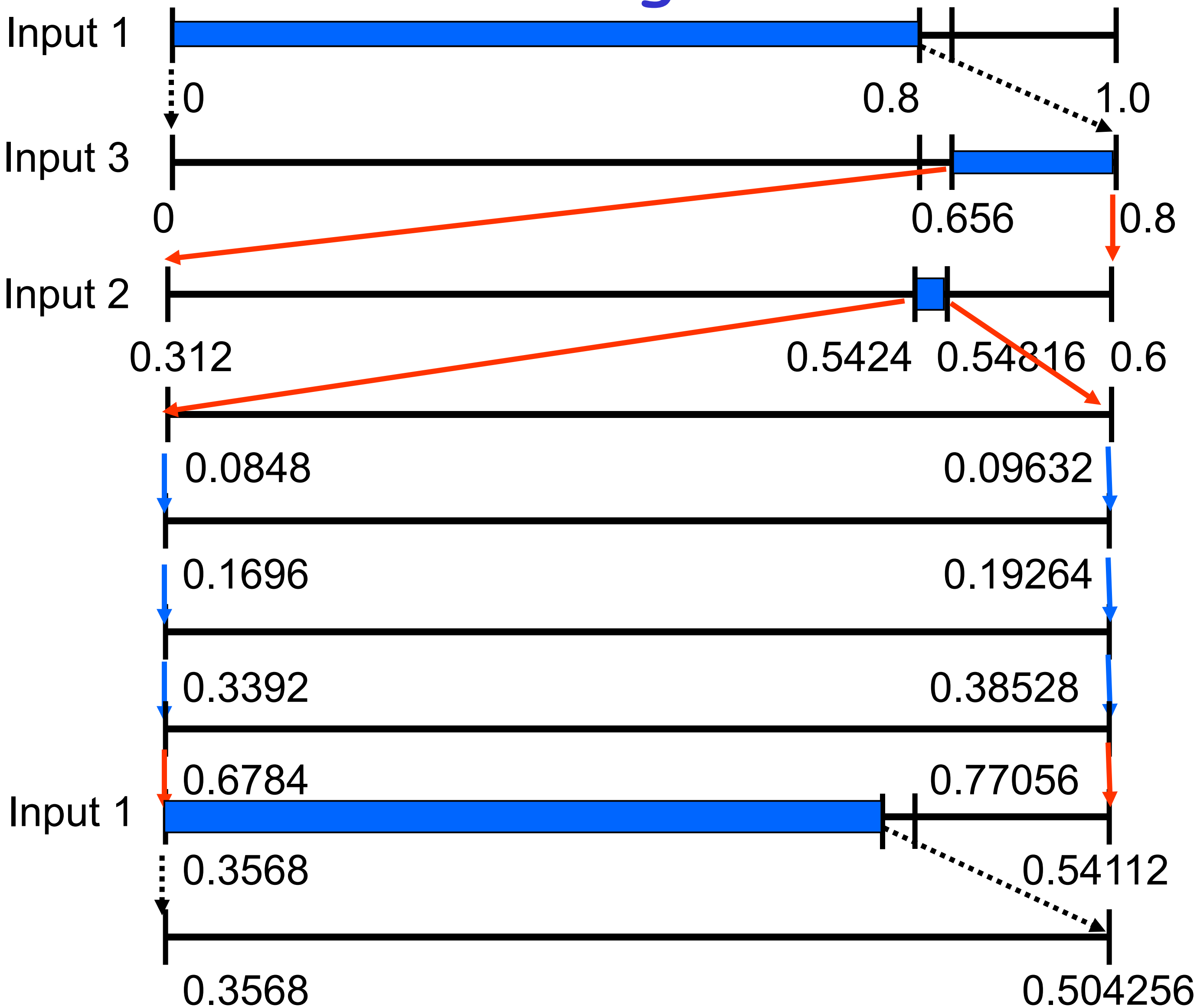
- E2: [LOW HIGH) in [0.5, 1)
  - LOW: 0.1xxxxxxx,
  - HIGH: 0.1xxxxxxx.



- Output 1, subtract 0.5, shift left by 1 bit
  - [0.5, 1) →[0, 1): E2(x) = 2(x - 0.5)

Encoding with E1 and E2

Symbol	Prob.
1	0.8
2	0.02
3	0.18



E2: Output 1
2(x – 0.5)

E2: Output 1

E1: 2x, Output 0

E1: Output 0

E1: Output 0

E2: Output 1

Encode any value in the tag, e.g., 0.5

Output 1

All outputs: 1100011

To verify

- LOW = 0.5424 (0.10001010... in binary),
  HIGH = 0.54816 (0.10001100... in binary).
- So we can send out 10001 (0.53125)
  - Equivalent to E2→E1→E1→E1→E2
- After left shift by 5 bits:
  - LOW = (0.5424 – 0.53125) x 32 = 0.3568
  - HIGH = (0.54816 – 0.53125) x 32 = 0.54112
  - Same as the result in the last page.

- Note: Complete all possible scaling before encoding the next symbol

Symbol	Prob.
1	0.8
2	0.02
3	0.18

Comparison with Huffman

- Input Symbol 1 does not cause any output
- Input Symbol 3 generates 1 bit
- Input Symbol 2 generates 5 bits

- Symbols with larger probabilities generates less number of bits.
  - Sometimes no bit is generated at all
    → Advantage over Huffman coding
- Large probabilities are desired in arithmetic coding
  - Can use context-adaptive method to create larger probability and to improve compression ratio.