CSY3010 Media Technology

Second half of the module

- Lossless compression
- Lossy compression (JPEG)
- Video encoding (Moving pictures)
- Advanced Video Codec (H.264/AVC)
- Build a HTML5 video service
- QoE in video streaming
- HTTP Adaptive Streaming (Build your own streaming service)
- Media Synchronisation
- Revision

Lossless Compression

Introduction

- Compression is the process of **transforming** raw data (in binary form) to compressed (or encoded) format.
- The resulting data being somewhat **smaller** than the original.
- The original data can be recreated with no data loss (hence lossless).
- The ratio between the two sizes is called the compression ratio; if the original is 200 KB, and the compressed is 50 KB, then we say a compression ratio of 4:1 has been achieved.

Introduction continued

Most compression systems are symmetric, in that the same algorithm that is used to compress the data is used 'in reverse' to uncompress it.

In general, compression is the reduction in size of data in order to save space or transmission time.

5

Standard Compression methods

- Lossless Compression
  - **RLE** run-length encoding method
  - Huffman coding method
  - Arithmetic Coding method
  - **LZW** algorithm
- Lossy Compression (to be covered in one of the coming lectures)
  - JPEG compression with Fourier transformation method

6

Run-length encoding method

- RLE is a compression techniques that are used in the most common **bitmap** graphics formats.
- It looks for repeating strings of the same character, which it replaces with a much shorter code; decoding the image is just a matter of replacing the code with the string.

7

RLE method continued

For example, a sequence of 12 A's (AAAAAAAAAAAA) might be replaced by 12A.

In another words, we are replacing 12 bytes by only 2 bytes.

To uncompress the image, each occurrence of 12A is simply replaced by AAAAAAAAAAAA.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

8

RLE method continued

AAAAAAbbbXXXXXt

Using run-length encoding this could be compressed into four 2-byte packets:

6A3b5X1t

Thus, after run-length encoding, the 15-byte string would require only eight bytes of data to represent the string, as opposed to the original 15 bytes.

In this case, run-length encoding yielded a compression ratio of almost 2 to 1.

The last run (containing the character t) was only a single character in length; a 1-character run is still a run. Both a run count and a run value must be written for every 2-character run.

To encode a run in RLE requires a minimum of two characters worth of information; therefore, a run of single characters actually takes more space. For the same reasons, data consisting entirely of 2-character runs remains the same size after RLE encoding.

9

RLE method continued

Xtmprsqzntwlfb

After RLE encoding, this string becomes:

1X1t1m1p1r1s1q1z1n1t1w1l1f1b

Coding efficiency depends on content

10

Huffman Coding

- Huffman is a coding algorithm presented by David Huffman in 1952.
- It's an algorithm which works well to provide optimal coding solution for integer length codes.
- For fixed length code, each codeword uses the same number of bits.
- Huffman employs the *variable-length* coding strategy which maps source symbols to a variable number of bits.

11

Huffman Coding continued

character	frequency	fixed length code	variable length code
a	.45	000	0
b	.13	001	101
c	.12	010	100
d	.16	011	111
e	.09	100	1101
f	.05	101	1100

Code word lengths vary and will be shorter for the more frequently used characters.

12

Huffman Coding continued

1. Scan text to be compressed and examine the occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.
6. Decoding is more or less the reverse process, based on the probabilities and the coded data, it outputs the decoded byte.

13

Huffman Coding (Building the tree)

Scan the original text (17 characters):

i love river nene

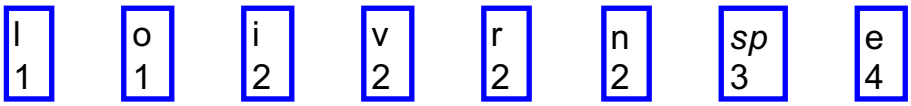
What is the frequency of each character in the text?

char	freq	char	freq
i	2	v	2
space	3	e	4
l	1	r	2
o	1	n	2

14

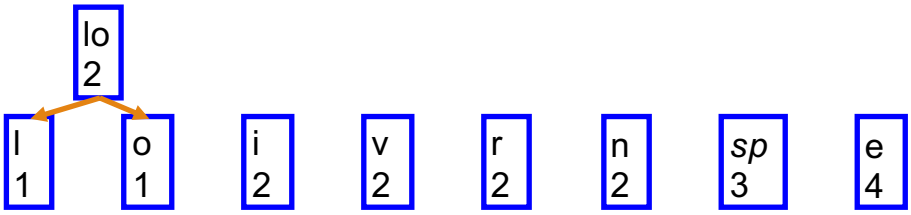
Huffman Coding

Recursively combine two smallest entries:



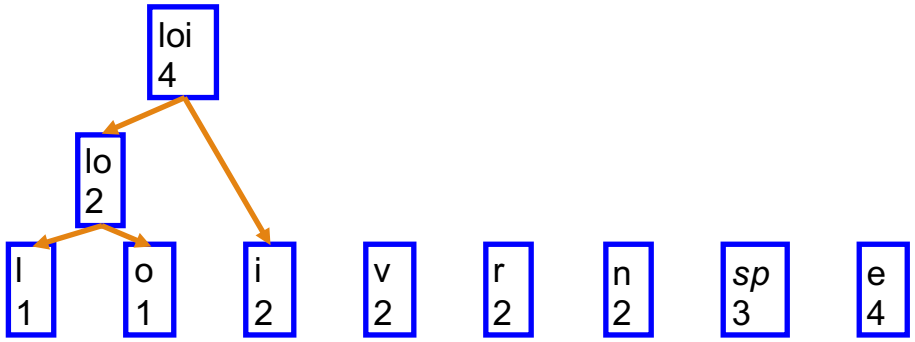
15

Huffman Coding



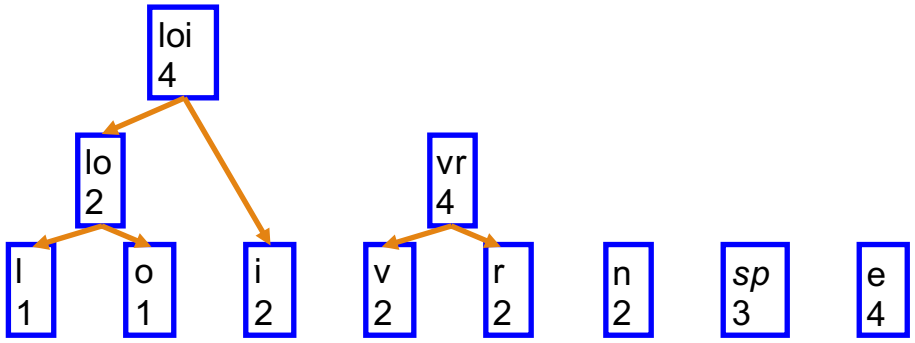
16

Huffman Coding



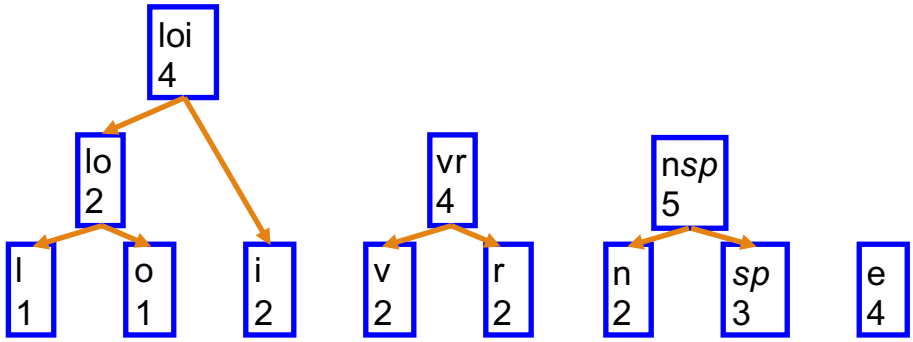
17

Huffman Coding



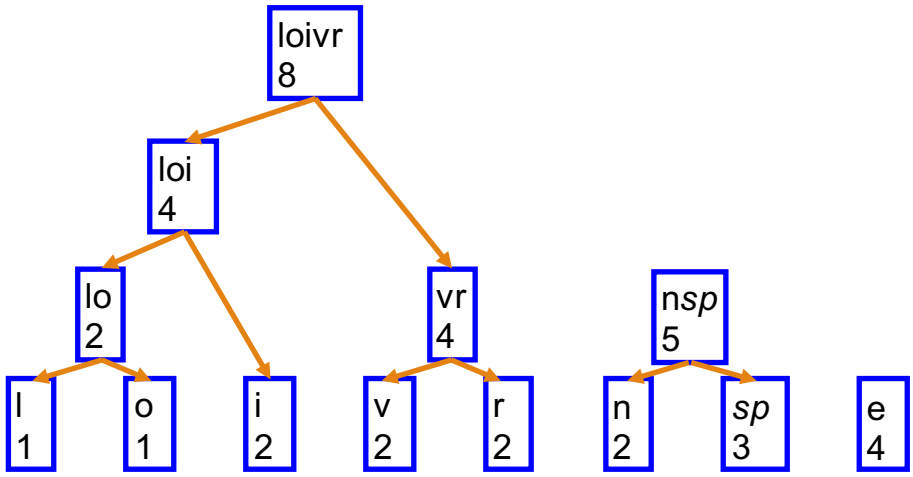
18

Huffman Coding



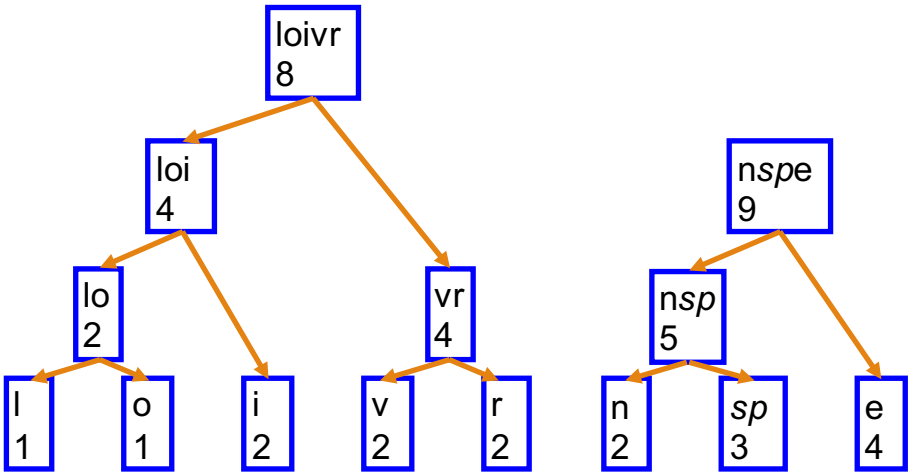
19

Huffman Coding



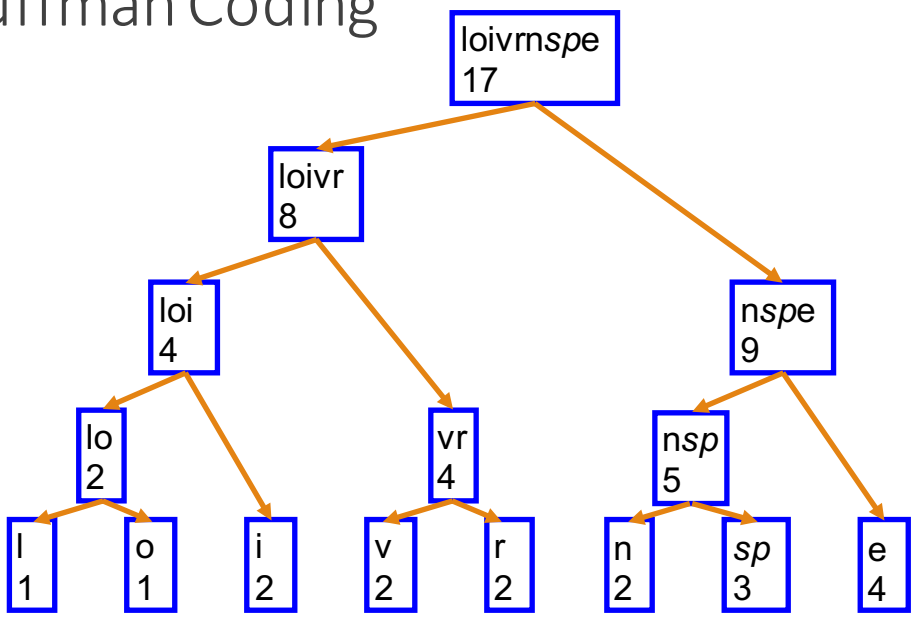
20

Huffman Coding



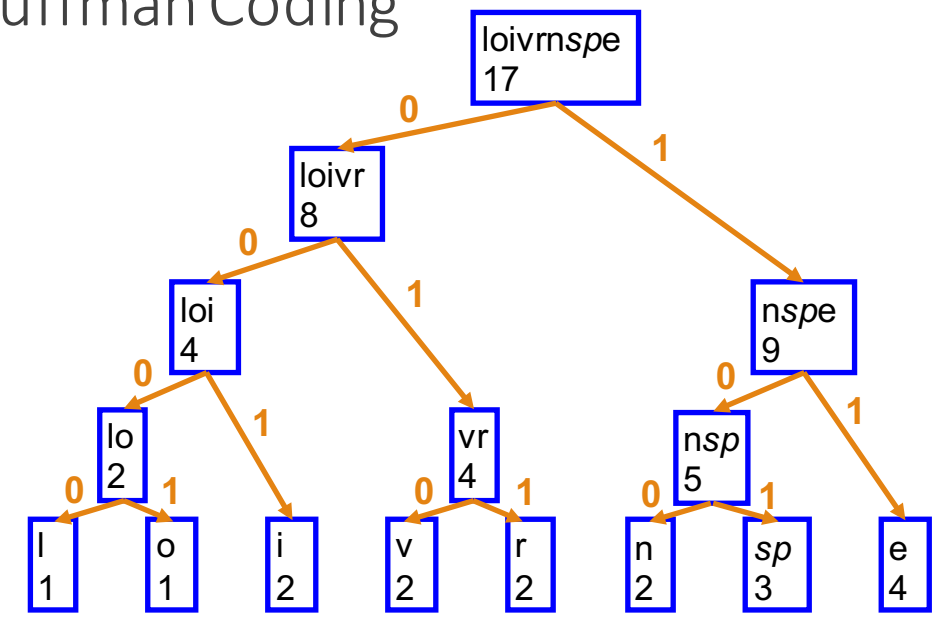
21

Huffman Coding



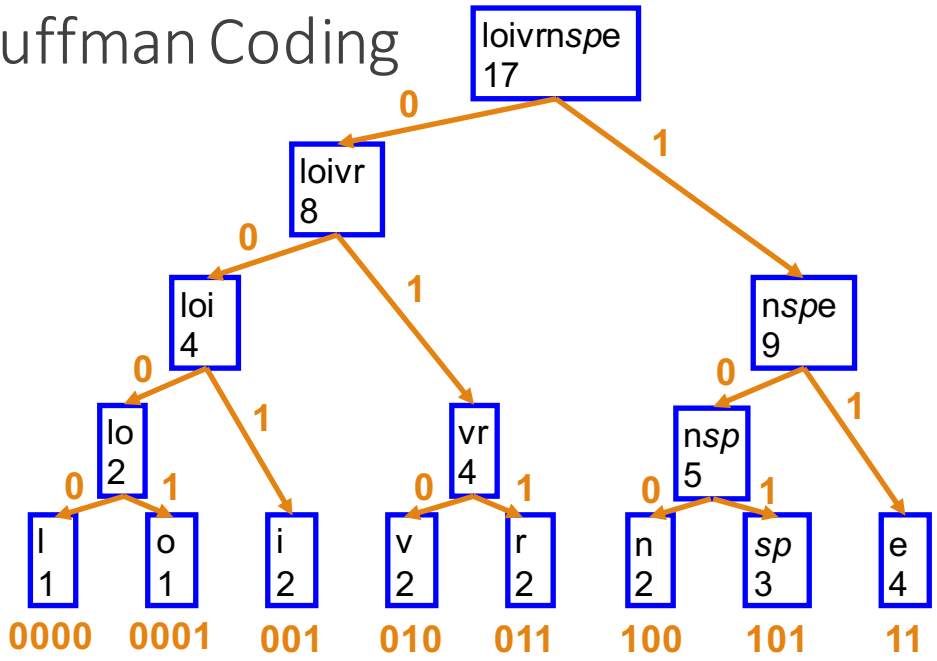
22

Huffman Coding



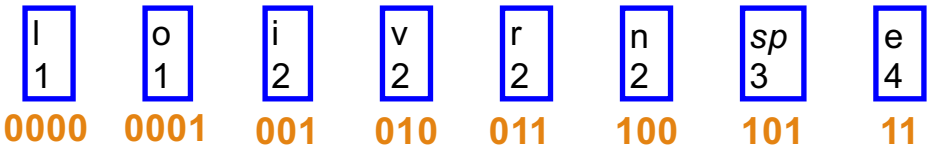
23

Huffman Coding



24

Huffman Coding



"i love river nene" (17 characters including space)

using Huffman:
00110100000001010111010110010101101110110011 0011 (49 bits)

using ASCII: 17x8=136 bits
01101001 00100000 01101100 01101111 01110110 01100101
00100000 01110010 01101001 01110110 01100101 01110010
00100000 01101110 01100101 01101110 01100101

using 5-bit: 17x5=85 bits

25

Arithmetic coding

Arithmetic coding is a data compression technique that encodes data (the data string) by creating a code string which represents a **fractional value** on the number line between 0 and 1.

The coding algorithm is symbol wise recursive; i.e. it operates upon and encodes (decodes) one data symbol per iteration or recursion.

On each recursion, the algorithm successively partitions an interval of the number line between 0 and 1, and retains one of the partitions as the new interval.

http://domino.research.ibm.com/tchjr/journalindex.nsf/0/aa3c6bd1731fbf0485256bfa0067f5bb?OpenDocument

26

Arithmetic coding continued

- Thus, the algorithm successively deals with smaller intervals, and the code string, viewed as a magnitude, lies in each of the nested intervals.
- The data string is recovered by using magnitude comparisons on the code string to recreate how the encoder must have successively partitioned and retained each nested subinterval.

http://domino.research.ibm.com/tchjr/journalindex.nsf/0/aa3c6bd1731fbf0485256bfa0067f5bb?OpenDocument

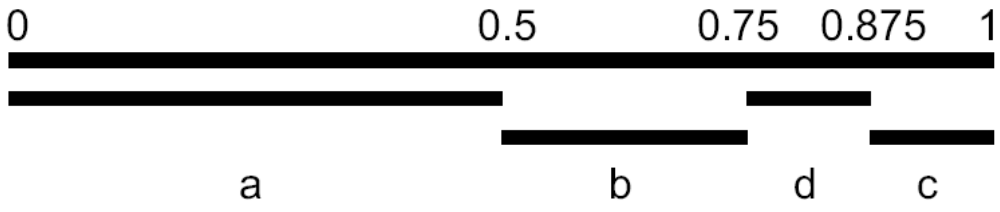
27

Example of Arithmetic Coding

Let $P_M(a)=0.5$ *is the probability of character "a" appearing in a text to be compressed:* abaabcda

$$P_M(a) = 0.5, P_M(b) = 0.25, P_M(c) = 0.125, P_M(d) = 0.125.$$

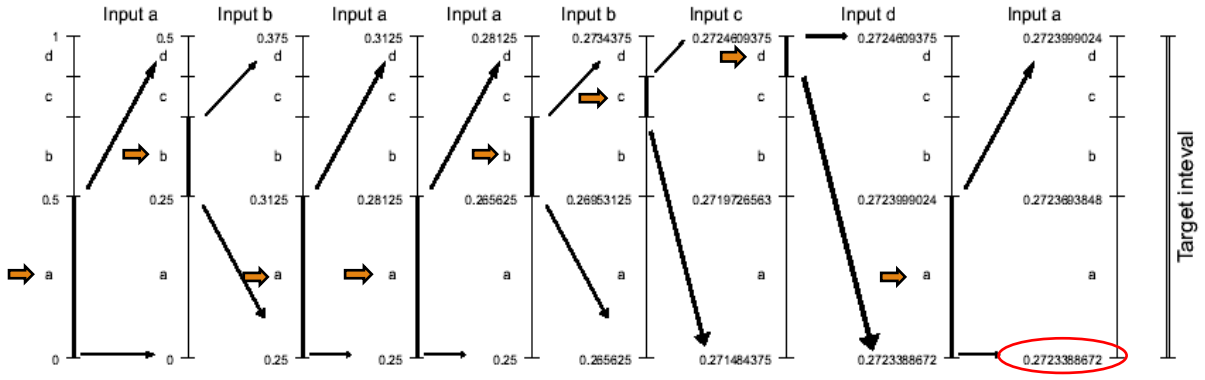
Now the interval $[0, 1)$ would be split as emphasized in Figure 1.



28

Example of Arithmetic Coding continued

To compress the input "abaabcda"



Example 2

Consider encoding the name BILL GATES

Again, we need the frequency of all the characters in the text.

char	freq.
space	0.1
A	0.1
B	0.1
E	0.1
G	0.1
I	0.1
L	0.2
S	0.1
T	0.1

Example 2

character	probability	range
space	0.1	[0.00, 0.10)
A	0.1	[0.10, 0.20)
B	0.1	[0.20, 0.30)
E	0.1	[0.30, 0.40)
G	0.1	[0.40, 0.50)
I	0.1	[0.50, 0.60)
L	0.2	[0.60, 0.80)
S	0.1	[0.80, 0.90)
T	0.1	[0.90, 1.00)

31

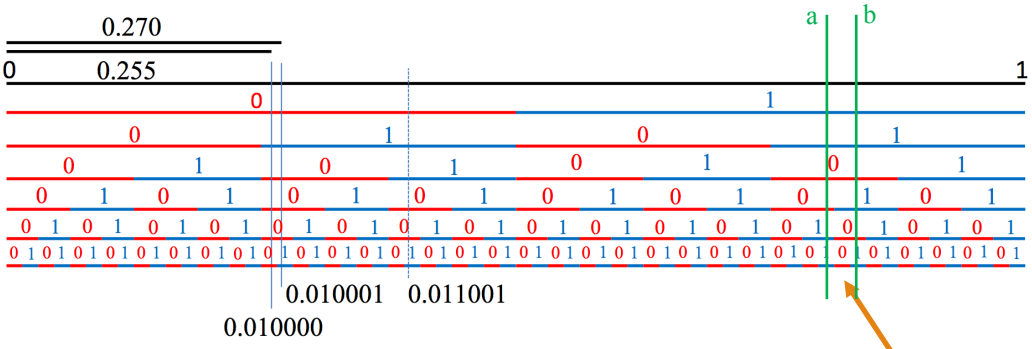
Example 2

chr	low	high
	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

The final range will uniquely encode the name BILL GATES

32

Binary representation of numbers in the unit segment



The binary unit segment

Use binary to divide the space until there is a **binary segment** that falls within the arithmetic range (e.g., 0.876073 - 0.876082).

33

Decoding

Suppose we have to decode 0.64

The decoder needs symbol probabilities, as it simulates what the encoder must have been doing

It starts with *low*=0 and *high*=1 and divides the interval exactly in the same manner as the encoder:

- *a* in [0, 1/3),
- *b* in [1/3, 2/3),
- *c* in [2/3, 1)

34

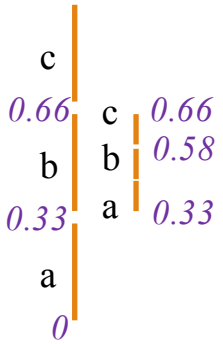
Decoding

The trasmitted number falls in the interval corresponding to *b*, so *b* must have been the first symbol encoded

Then the decoder evaluates the new values for *low* (0.3333) and for *high* (0.6667), updates symbol probabilities and divides the range from *low* to *high* according to these new probabilities

Decoding proceeds until the full string has been reconstructed

Decoding



0.64 in [0.3333, 0.6667) *b*

0.64 in [0.5834, 0.6667) *c...*

and so on...

but when to stop?
The encoded string often includes the initial character length:
Char_length, encoded_messange

Further reading: https://www.youtube.com/watch?v=I6OeaKdlNF4

LZW method

The most widely-used alternative is the **LZW** algorithm, *which was developed by **L**empel, **Z**iv and **W**elch (hence its name) in the 1970s*.

Some of the most efficient imagefile formats (such as **GIF** and **TIFF**) use this encoding method.

Universal coding schemes, like LZW, do not require prior knowledge and can build such knowledge on-the-fly.

LZW is the foremost technique for general purpose data compression due to its simplicity and versatility.

- It is the basis of many PC utilities that claim to *"double the capacity of your hard drive"*
- LZW compression uses a code table, with 4096 as a common choice for the number of table entries.

37

LZW method continued

Codes 0-255 in the code table are always assigned to represent single bytes from the input file.

When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks.

Compression is achieved by using codes 256 through 4095 (12 bits) to represent sequences of bytes.

As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table.

Decoding is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents.

http://www.ascii-code.com/

38

LZW Encoding Algorithm

```
1   Initialize table with single character strings
2   P = first input character
3   WHILE not end of input stream
4           C = next input character
5           IF  P + C  is in the string table
6                   P = P + C
7           ELSE
8                   output the code for P
9                           add P + C to the string table
10          P = C
11  ENDWHILE
12  output code for P
```

39

Example 1: Compression using LZW

Example 1: Use the LZW algorithm to compress the string

BABAABAAA

40

Example 1: LZW Compression Step 1

BABAABAAA
↑

P=A
C=empty

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA

41

Example 1: LZW Compression Step 2

BABAABAAA
↑

P=B
C=empty

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB

42

Example 1: LZW Compression Step 3

BABAABAAA

P=A
C=empty

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA

43

Example 1: LZW Compression Step 4

BABAABAAA

P=A
C=empty

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA

44

Example 1: LZW Compression Step 5

BABAABAAA
↑

P=A
C=A

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA
65	A	260	AA

45

Example 1: LZW Compression Step 6

BABAABAAA
↑

P=AA
C=empty

ENCODER	OUTPUT	STRING	TABLE
output code	representing	codeword	string
66	B	256	BA
65	A	257	AB
256	BA	258	BAA
257	AB	259	ABA
65	A	260	AA
260	AA		

46

LZW Decompression

The LZW decompressor creates the same string table during decompression.

It starts with the first 256 table entries initialized to single characters.

The string table is updated for each character in the input stream, except the first one.

Decoding achieved by reading codes and translating them through the code table being built.

47

LZW Decompression Algorithm

```
1   Initialize table with single character strings
2   OLD = first input code
3   output translation of OLD
4   WHILE not end of input stream
5      NEW = next input code
6      IF NEW is not in the string table
7           S = translation of OLD
8           S = S + C
9      ELSE
10          S = translation of NEW
11     output S
12     C = first character of S
13     OLD + C to the string table
14     OLD = NEW
15  END WHILE
```

48

Example 2: LZW Decompression 1

Example 2: Use LZW to decompress the output sequence of Example 1:

<66><65><256><257><65><260>.

Example 2: LZW Decompression Step 1

<66><65><256><257><65><260>

Old = 65 S = A
New = 66 C = A

ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA

Example 2: LZW Decompression Step 2

<66><65><256><257><65><260> Old = 256 S = BA

New = 256 C = B

ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA
BA	257	AB

51

Example 2: LZW Decompression Step 3

<66><65><256><257><65><260> Old = 257 S = AB

New = 257 C = A

ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA
BA	257	AB
AB	258	BAA

52

Example 2: LZW Decompression Step 4

<66><65><256><257><65><260> Old = 65 S = A

New = 65 C = A

ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA
BA	257	AB
AB	258	BAA
A	259	ABA

53

Example 2: LZW Decompression Step 5

<66><65><256><257><65><260> Old = 260 S = AA

New = 260 C = A

ENCODER OUTPUT	STRING TABLE	
string	codeword	string
B		
A	256	BA
BA	257	AB
AB	258	BAA
A	259	ABA
AA	260	AA

54

LZW: Some Notes

- This algorithm compresses repetitive sequences of data well.
- Since the code words are 12 bits, any single encoded character will expand the data size rather than reduce it.
- In this example, 72 bits are represented with 72 bits of data. After a reasonable string table is built, compression improves dramatically.
- Advantages of LZW over Huffman:
  - LZW requires no prior information about the input data stream.
  - LZW can compress the input stream in one single pass.
  - Another advantage of LZW its simplicity, allowing fast execution.

55

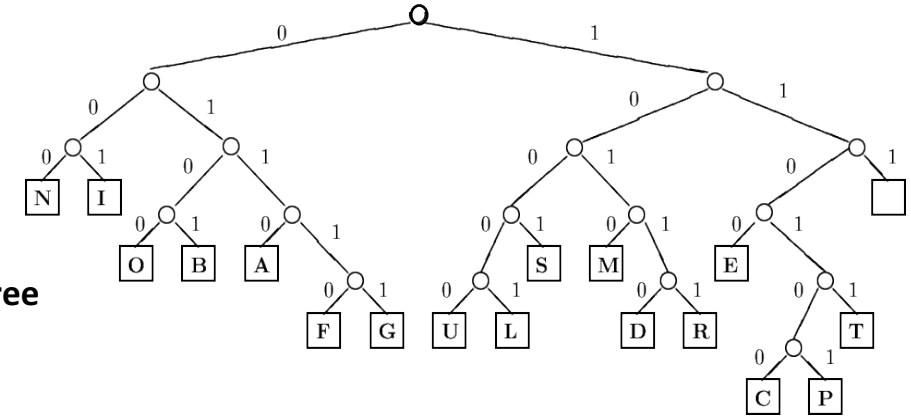
LZW: Limitations

- What happens when the dictionary gets too large (i.e., when all the 4096 locations have been used)?
- Here are some options usually implemented
  - Simply forget about adding any more entries and use the table as is.
  - Throw the dictionary away when it reaches a certain size.
  - Throw the dictionary away when it is no longer effective at compression.
  - Clear entries 256-4095 and start building the dictionary again.
    - Some clever schemes rebuild a string table from the last N input characters.

56

Exercise 1

- Using the Huffman tree shown below, compress and decompress the below phrases:
  - OPEN DOOR
  - BIG FLAG
  - Build your own tree



57

Exercise 2

- Use LZW to trace encoding the strings
  - ABRACADABRA
  - ALAABABA SHARBATLAMON

(answers in slide notes)

58