

## COMMONWEALTH OF AUSTRALIA

### *Copyright Regulations 1969*

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

**Prepared by:** [Arun Konagurthu]

# FIT3155 S2/2020: Advanced Algorithms and Data Structures

Week 9: Data compression related algorithms

Faculty of Information Technology, Monash University

# What is covered in this lecture?

This week's lecture deals with...

- Introduction to lossless data compression.
- Fixed and variable length codes.
- Prefix-free code words for encoding/decoding
  - ▶ character streams
  - ▶ integer streams
- A simple dictionary-based compression algorithm on text.

## Source material and recommended reading

- MacKay, Information theory, inference and learning algorithms (chapter 1) [\[Link\]](#)
- Sayood, Introduction to Data compression (chapters 3&5)

# Introduction

- Human data streams generate  $\approx$  **2.5 quintillion bytes** of data each day.
- This data is of various descriptions (text, images, movies, structured-tables, etc.)
- Compression of data saves **space** and **time** (in storage/retrieval/transmission etc).

## Modern applications

### For general files

- zip/gzip/7zip
- bzip
- pkzip
- etc.

### For multimedia

- GIF, JPEG
- MPEG, DivX,
- etc.

### Others

- Fax, modem
- skype, zoom
- large databases: Google, Amazon, Twitter etc.

# Compression is a really old idea/technology

- Natural language (as old as humanity) internalizes compression:
  - ▶ Often, **commonly** articulated words
    - ★ 'yes'/'no'
    - ★ 'go'/'come',
    - ★ 'ma'/'pa')are shorter...
  - ▶ ...than words that are **uncommon**.
    - ★ 'piscatorial'
    - ★ 'abstemious'
    - ★ 'floccinaucinihilipilification'
- Mathematics embodies compression.
- Some old technologies (circa 1800s) for communicating textual information (where short codes are assigned to common characters, and long codes for uncommon ones):
  - ▶ Braille (tactile) code for visually impaired
  - ▶ Morse code for telegraph
  - ▶ Baudot code for teletype
  - ▶ etc.

# A simple ('compressed') view of data

$$\begin{aligned} Data &= \text{redundant parts} + \text{random parts} \\ &= \text{compressible parts} + \text{uncompressible parts} \\ &= \text{model} + \text{deviations} \\ &= \text{signal} + \text{noise} \end{aligned}$$

# Lossless compression – encoding and decoding



## Lossless compression

Lossless data compression algorithms allow encoding the original data into a compact (encoded) form, and which in turn can be perfectly reconstructed (decoded) to get back the original data.



# Outcomes and Probability of outcomes



## Random variable and outcomes

- A **random variable**  $x$  is a variable...
- ...that can take a set of *possible* values,  $o_1, o_2, \dots, o_n$ .
- Each such possible value is an **outcomes** of some (random) event/phenomenon. Examples:
  - ▶ Throw of dice
  - ▶ Coin tosses
  - ▶ Occurrence of a character in text.
  - ▶ etc.
- Associated with each outcome  $x = o_i$ , there is a **probability** of that outcome, denoted by  $\Pr(x = o_i)$ .

# How do we measure information content of an outcome

$x = o_i$ ?

Among the outstanding contributions of Claude E. Shannon is his work on **Mathematical theory of communication**

## Shannon's **information content** of an outcome

The measure of the information content of an **outcome**  $x = o_i$  of a random variable  $x$  is given by:

$$I(o_i) = -\log(\Pr(x = o_i))$$

---

If the base of the log is 2,  $I(\cdot)$  is measured in **bits**.

## Shannon's **entropy** over all outcomes

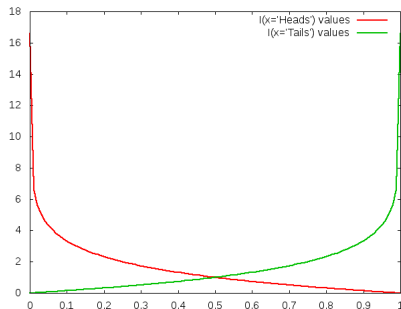
Shannon's **entropy** gives the measure of the **average** information content across all outcomes of the random variable  $x$ :

$$H(x) = \sum_{i=1}^n \Pr(x = o_i) I(o_i)$$

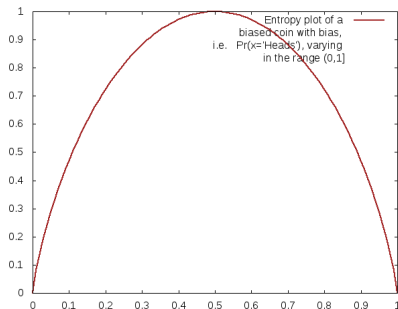
## Example: Coin with a bias

- Assume we have coin with a bias  $p$ .
- That is,  $\Pr(x = \text{Heads}) = p$ ;  $\Pr(x = \text{Tails}) = 1 - p$ .
- Shannon's information content:
  - $I(\text{Heads}) = -\log_2(p)$
  - $I(\text{Tails}) = -\log_2(1 - p)$
- Shannon's Entropy:  $H(x) = p \log_2(1/p) + (1 - p) \log_2(1/(1 - p))$ .

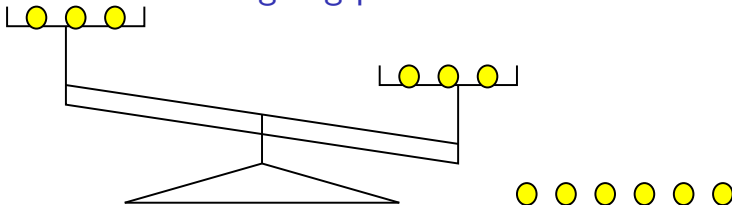
Information content plot



Entropy plot



## Detour: The weighing puzzle



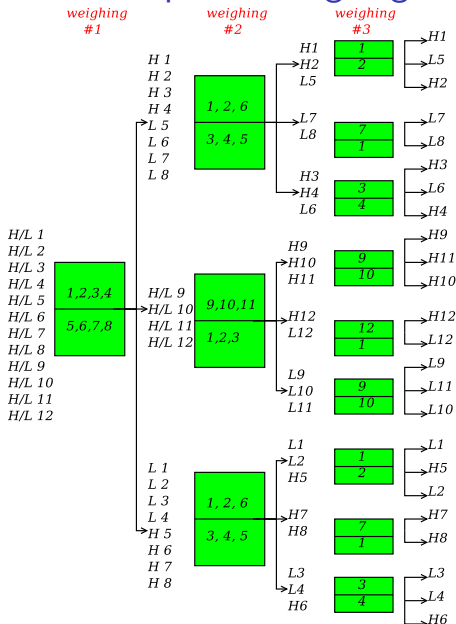
### Puzzle statement

We have 12 identically looking balls. All have equal weight, **except one**. In addition, we do **not** know if this defective ball is **heavier** or **lighter** than the rest.

At your disposal is a simple two-pan balance you can use to weigh the balls (putting any number of balls in each pan, per weighing). The **outcome** of each such weighing is: **heavy**, **equal**, or **light**.

What is the **optimal weighing strategy** to determine the defective ball **and also** whether it is heavier or lighter, **in as few weighings** of the balance as possible?

# Detour: Optimal weighing solution for the puzzle



# Fixed length code words

## Fixed length code words

- Each symbol's code word takes the **same** number of bits to state
- Convenient to encode and decode each symbol.
- With fixed-length code words, we do **not** care about the underlying probability/frequency of each symbol's occurrence.
  - ▶ Underlying assumption is that the distribution is **uniform**.

# Fixed-length code words example – ASCII

## Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

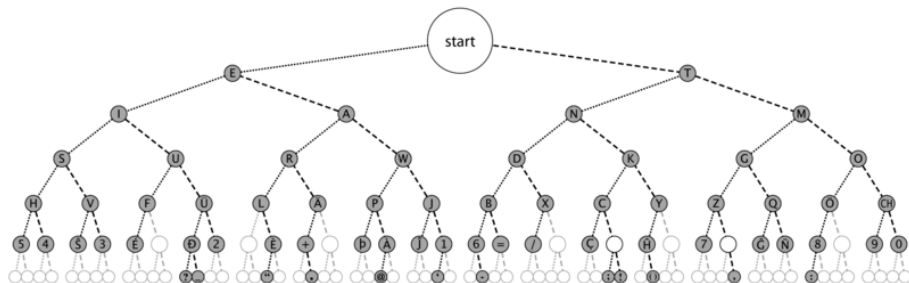
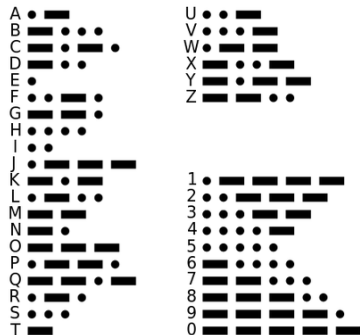
# Variable length code words

## Variable length code words

- Each symbol's code word takes **varying** number of bits
- Takes **more effort** to encode and decode each symbol.
- Here, we have to consider the underlying probability of each symbol's occurrence, when designing the code words.



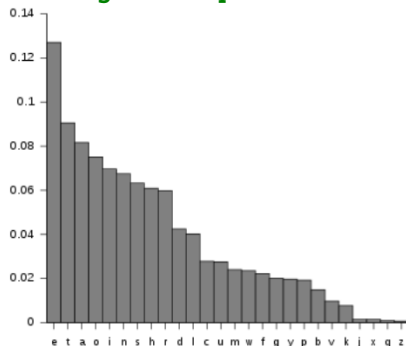
# Variable-length code words example – Morse code



# Variable length codes are central to compression

- If **frequent symbols** in some text are assigned **shorter code words**...
- ...and **infrequent symbols** are assigned **longer code words**...
- ...then the **total number of bits** to encode the source text/data will be **smaller**...
- ...i.e., **compressed**, compared to encoding using fixed-length code words.
- These variable-length code words provide the **building blocks** for data compression.

Frequency of letters in the English alphabet:



# Problem with variable-length codes – decoding can become ambiguous!

## Example of ambiguous variable-length code words

Assume we are using these code words: **A** = 0, **B** = 1, **C** = 10, **D** = 101

Encoding of **BABA** using this code is: 1010

# Problem with variable-length codes – decoding can become ambiguous!

## Example of ambiguous variable-length code words

Assume we are using these code words: **A** = 0, **B** = 1, **C** = 10, **D** = 101

Encoding of **BABA** using this code is: 1010

Decoding of this encoded message is **not** unique, since we don't know where each code word starts and ends. In the above example, the message can be decoded as:

- 1 0 1 0 = **B A B A**, or

# Problem with variable-length codes – decoding can become ambiguous!

## Example of ambiguous variable-length code words

Assume we are using these code words: **A** = 0, **B** = 1, **C** = 10, **D** = 101

Encoding of **BABA** using this code is: 1010

Decoding of this encoded message is **not** unique, since we don't know where each code word starts and ends. In the above example, the message can be decoded as:

- 1 0 1 0 = **B A B A**, or
- 10 10 = **C C**, or

# Problem with variable-length codes – decoding can become ambiguous!

## Example of ambiguous variable-length code words

Assume we are using these code words: **A** = 0, **B** = 1, **C** = 10, **D** = 101

Encoding of **BABA** using this code is: 1010

Decoding of this encoded message is **not** unique, since we don't know where each code word starts and ends. In the above example, the message can be decoded as:

- 1 0 1 0 = **B A B A**, or
- 10 10 = **C C**, or
- 101 0 = **D A**

(spaces between code words added above for convenience in parsing)

# Prefix-free (variable-length) codes provide unique decodability

Now consider these variable-length code words

Assume we are using these code words: **A** = 0, **B** = 10, **C** = 110, **D** = 111

Encoding of **BABA** using this code is: 100100

Indeed this encoded message is **uniquely decodable**. Why?

---

**Prefix-free** codes are sometimes shorthand to '**prefix codes**', without explicitly adding '**-free**'. They are also called **instantaneous codes**

# Prefix-free (variable-length) codes provide unique decodability

Now consider these variable-length code words

Assume we are using these code words: **A** = 0, **B** = 10, **C** = 110, **D** = 111

Encoding of **BABA** using this code is: 100100

Indeed this encoded message is **uniquely decodable**. Why?

**Answer:** No letter's code word is a **prefix** of another letter's code word. Such code words are called **prefix-free** codes.

---

**Prefix-free** codes are sometimes shorthanded to '**prefix codes**', without explicitly adding '**-free**'. They are also called **instantaneous codes**



## Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.

## Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.

## Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.
- It yields:

## Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.
- It yields:
  - ▶ the shortest code word for the most frequent character.

## Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.
- It yields:
  - ▶ the shortest code word for the most frequent character.
  - ▶ the second shortest code word for the second most frequent character.

# Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.
- It yields:
  - ▶ the shortest code word for the most frequent character.
  - ▶ the second shortest code word for the second most frequent character.
  - ▶ ...and so on.

# Huffman coding yields **prefix-free** codes

- Huffman coding is a method of generating reliable **prefix-free** code words.
- It **requires** as input, the **frequencies** of characters.
- It yields:
  - ▶ the shortest code word for the most frequent character.
  - ▶ the second shortest code word for the second most frequent character.
  - ▶ ...and so on.
- the coding algorithm falls in the class of **greedy algorithm**.

# Huffman coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



# Huffman coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDD\_A\_BAD\_BABE\_A\_BEADDED\_ABACA\_BED"

2.

C	:	2
B	:	6
E	:	7
_	:	10
D	:	10
A	:	11

# Huffman coding by example – from [wikipedia]

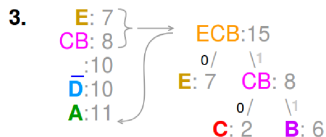
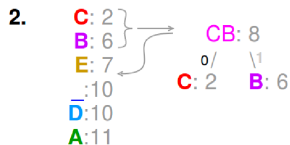
1. "A\_DEAD\_DAD\_CEDD\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"

2.

C: 2	}	→	CB: 8	
B: 6			0/	\1
E: 7				
_: 10			C: 2	B: 6
D: 10				
A: 11				

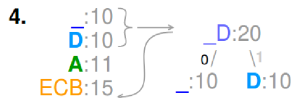
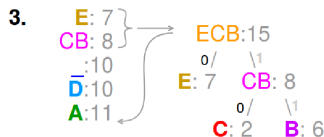
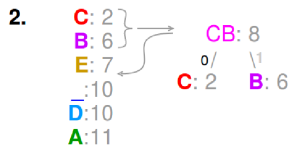
# Huffmanmann coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



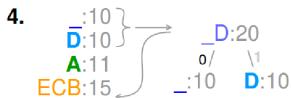
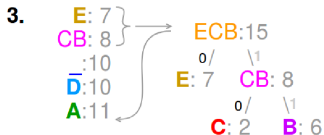
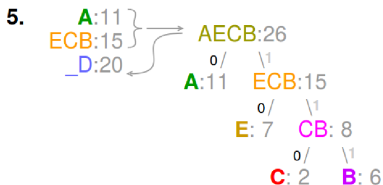
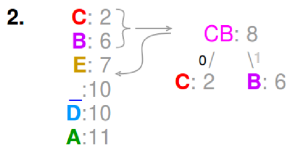
# Huffmanmann coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



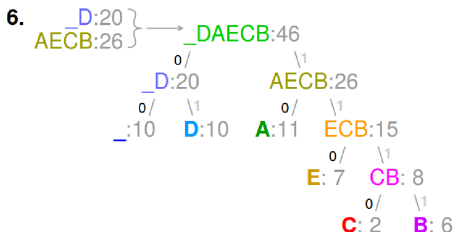
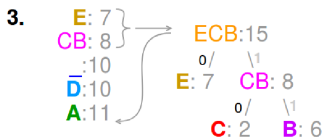
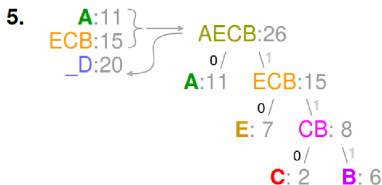
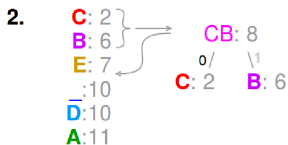
# Huffman coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDD\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



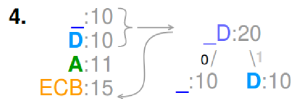
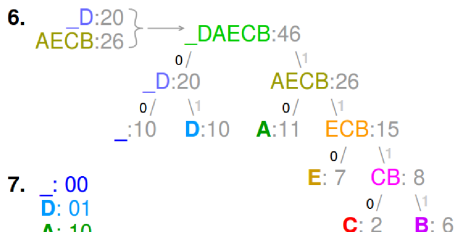
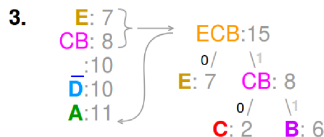
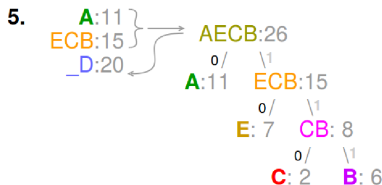
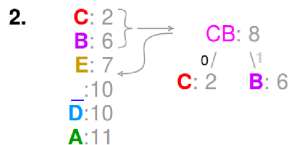
# Huffman coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



# Huffmanmann coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"

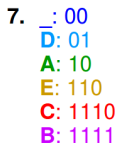
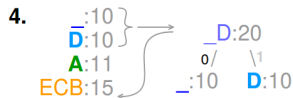
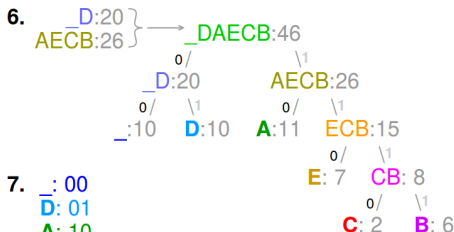
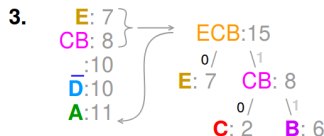
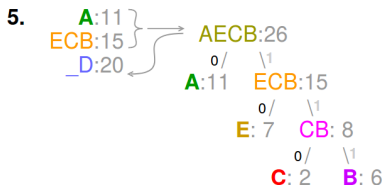
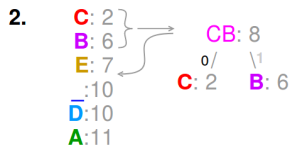


7.

\_: 00  
 D: 01  
 A: 10  
 E: 110  
 C: 1110  
 B: 1111

# Huffmanmann coding by example – from [wikipedia]

1. "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED"



8. "100001110100100011001001110110011100100100011111001001111101111110  
0010001111110100111001001011111011101000111111001"



# Huffman coding algorithm summary

- ➊ Compute frequencies of each unique character in the given text/data.
- ➋ Imagine each character as a leaf node in a binary tree you are constructing.
- ➌ Repeatedly join two characters/nodes with the smallest frequencies to form a new node.
  - ▶ this new node represents the sum of frequencies of nodes that were joined.
  - ▶ Assign a bit symbol **0** to the left branch...
  - ▶ ... and the bit symbol **1** to the right branch.
- ➍ Stop repeating step 3 when all nodes are joined into a rooted binary tree.
- ➎ Each character's code word is then the sequence of **0**s and **1**s generated from **root**-to-**leaf** traversal on that binary tree.

# Huffman coding algorithm summary

- ❶ Compute frequencies of each unique character in the given text/data.
- ❷ Imagine each character as a leaf node in a binary tree you are constructing.
- ❸ Repeatedly join two characters/nodes with the smallest frequencies to form a new node.
  - ▶ this new node represents the sum of frequencies of nodes that were joined.
  - ▶ Assign a bit symbol **0** to the left branch...
  - ▶ ... and the bit symbol **1** to the right branch.
- ❹ Stop repeating step 3 when all nodes are joined into a rooted binary tree.
- ❺ Each character's code word is then the sequence of **0**s and **1**s generated from **root**-to-**leaf** traversal on that binary tree.

## Decoding

Decoding Huffman encoded bitstream back into text is simple. Why?

## What about prefix-free codes for **integers**?

- Assume we have a data stream of integers coming from the set of natural numbers.
- Huffman coding is not effective for such data...
- ...especially when integers are spread over large ranges, and each integer in the data stream is nearly unique.
- This motivates designing variable-length, prefix-free codes for integers.

## Definition: minimal binary code of a number

The **minimal binary code** of a number is the binary representation of that number such that the most significant digit is always **1** (that is, there are no 0's padded on the most-significant side).

### Example

$(561)_{dec} = \dots$  <sup>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</sup>  
0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 .

The **minimal binary code** of  $(561)_{dec} =$  <sup>9 8 7 6 5 4 3 2 1 0</sup>  
1 0 0 0 1 1 0 0 0 1 .

# Elias (Omega) code for universal integers

By universal integers, we are considering positive integers from  $1, 2, \dots, \infty$ .

## Overarching strategy

The strategy to design a code word for an integer  $N$  is:

- Encode  $N$
- Encode  $L_1 = \text{length}(N)-1$
- Encode  $L_2 = \text{length}(L_1)-1$
- Encode  $L_3 = \text{length}(L_2)-1$
- ...
- .. and so on until the final encoded length is 1.

We will call the encodings of  $L_1, L_2, \dots$  as the **length component**, and the encoding of  $N$  as the **code component**.

## Elias (Omega) code for universal integers..continued

### Decoding can be problematic

Encoding  $N, L_1, L_2, \dots$  integers using directly their **minimal binary codes** poses a problem.

- During decoding, we cannot differentiate between the **length components** and the actual **code component** of  $N$ .

# Elias (Omega) code for universal integers..continued

## Decoding can be problematic

Encoding  $N, L_1, L_2, \dots$  integers using directly their **minimal binary codes** poses a problem.

- During decoding, we cannot differentiate between the **length components** and the actual **code component** of  $N$ .

## Example

Decoding...

11110011000110001

# Elias (Omega) code for universal integers..continued

## Decoding can be problematic

Encoding  $N, L_1, L_2, \dots$  integers using directly their **minimal binary codes** poses a problem.

- During decoding, we cannot differentiate between the **length components** and the actual **code component** of  $N$ .

## Example

Decoding...

11110011000110001

....requires its interpretation as the following length and code components

$\underbrace{1}_{L_3} \underbrace{11}_{L_2} \underbrace{1001}_{L_1} \underbrace{1000110001}_N$

is **problematic**.



# Elias (Omega) code for universal integers..continued

## Decoding can be problematic

Encoding  $N, L_1, L_2, \dots$  integers using directly their **minimal binary codes** poses a problem.

- During decoding, we cannot differentiate between the **length components** and the actual **code component** of  $N$ .

## Example

Decoding...

11110011000110001

....requires its interpretation as the following length and code components

$\underbrace{1}_{L_3} \underbrace{11}_{L_2} \underbrace{1001}_{L_1} \underbrace{1000110001}_N$

is **problematic**.

Elias (omega) encoding of integers addresses this problem by changing the most-significant **1** in the **minimal binary code** of each **length component** to **0**. Let's see how the encoding and decoding works.

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The minimal binary code of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$
- The **minimal binary code** for  $L_1$  is therefore  $\overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .

# Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$
- The **minimal binary code** for  $L_1$  is therefore  $\overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- **IMPORTANT RULE:** When coding any **length component**, **change** the leading 1 of its minimal binary code to 0.

# Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$
- The **minimal binary code** for  $L_1$  is therefore  $\overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- **IMPORTANT RULE:** When coding any **length component**, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .



## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$
- The **minimal binary code** for  $L_1$  is therefore  $\overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- IMPORTANT RULE:** When coding any **length component**, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .

## Elias (omega) code for universal integers – Encoding via an example

- Let  $N = (561)_{dec}$ .
- The **minimal binary code** of  $N$  is  $\overset{9}{1} \overset{8}{0} \overset{7}{0} \overset{6}{0} \overset{5}{1} \overset{4}{1} \overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$
- The **length** of minimal binary code of  $N$  is  $(10)_{dec}$ .
- Therefore, the **length component** to encode is  $L_1 = (10 - 1 = 9)_{dec}$
- The **minimal binary code** for  $L_1$  is therefore  $\overset{3}{1} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- IMPORTANT RULE:** When coding any **length component**, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .

Encoded string so far

$\underbrace{0001}_{L_1} \underbrace{1000110001}_N$

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1} \text{ .}$

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .
- The **minimal binary code** for  $L_2$  is therefore  $\overset{1}{1} \overset{0}{1}$  .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .
- The **minimal binary code** for  $L_2$  is therefore  $\overset{1}{1} \overset{0}{1}$  .
- Again, since this is a **length** component, as per the rule stated earlier, **change** the leading 1 of its minimal binary code to 0.

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .
- The **minimal binary code** for  $L_2$  is therefore  $\overset{1}{1} \overset{0}{1}$  .
- Again, since this is a **length** component, as per the rule stated earlier, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .



## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .
- The **minimal binary code** for  $L_2$  is therefore  $\overset{1}{1} \overset{0}{1}$  .
- Again, since this is a **length** component, as per the rule stated earlier, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_1$  is now  $\overset{3}{0} \overset{2}{0} \overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_1$  is 4.
- The next **length component** to encode is therefore  $L_2 = (4 - 1 = 3)_{dec}$ .
- The **minimal binary code** for  $L_2$  is therefore  $\overset{1}{1} \overset{0}{1}$  .
- Again, since this is a **length** component, as per the rule stated earlier, **change** the leading 1 of its minimal binary code to 0.
- That is, the code to encode (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .

### Encoded string so far

$\underbrace{01}_{L_2} \underbrace{0001}_{L_1} \underbrace{1000110001}_N$

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_2$  is 2.

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .
- The **minimal binary code** for  $L_3$  is therefore  $\overset{0}{1}$  .

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .
- The **minimal binary code** for  $L_3$  is therefore  $\overset{0}{1}$  .
- Since this is a **length component**, as per the rule stated above, **change** the leading 1 of the **minimal binary code** to 0.

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$  .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .
- The **minimal binary code** for  $L_3$  is therefore  $\overset{0}{1}$  .
- Since this is a **length component**, as per the rule stated above, **change** the leading 1 of the **minimal binary code** to 0.
- That is, the code to encode (modified)  $L_3$  is now  $\overset{1}{0}$  .



## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$ .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .
- The **minimal binary code** for  $L_3$  is therefore  $\overset{0}{1}$ .
- Since this is a **length component**, as per the rule stated above, **change** the leading 1 of the **minimal binary code** to 0.
- That is, the code to encode (modified)  $L_3$  is now  $\overset{1}{0}$ .
- Since the length of (modified)  $L_3$  has reached 1, **STOP encoding!**

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $\overset{1}{0} \overset{0}{1}$ .
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}$ .
- The **minimal binary code** for  $L_3$  is therefore  $\overset{0}{1}$ .
- Since this is a **length component**, as per the rule stated above, **change** the leading 1 of the **minimal binary code** to 0.
- That is, the code to encode (modified)  $L_3$  is now  $\overset{1}{0}$ .
- Since the length of (modified)  $L_3$  has reached 1, **STOP encoding!**

## Elias (omega) code for universal integers – Encoding via an example

- From previous slide, (modified)  $L_2$  is now  $0 \overset{1}{1} \overset{0}{} .$
- The length of (modified)  $L_2$  is 2.
- The next **length component** to encode is therefore  $L_3 = (2 - 1 = 1)_{dec}.$
- The **minimal binary code** for  $L_3$  is therefore  $1 \overset{0}{} .$
- Since this is a **length component**, as per the rule stated above, **change** the leading 1 of the **minimal binary code** to 0.
- That is, the code to encode (modified)  $L_3$  is now  $0 \overset{1}{} .$
- Since the length of (modified)  $L_3$  has reached 1, **STOP encoding!**

### Final encoded bit string

$\underbrace{0}_{L_3} \underbrace{01}_{L_2} \underbrace{0001}_{L_1} \underbrace{1000110001}_N$

## Elias (omega) code words for the first few integers

integer $N$	Components (incl. $N$ )	code word
1	1	1
2	1,2	0 10
3	1,3	0 11
4	1,2,4	0 00 100
5	1,2,5	0 00 101
6	1,2,6	0 00 110
7	1,2,7	0 00 111
8	1,3,8	0 01 1000
9	1,3,9	0 01 1001
10	1,3,10	0 01 1010
...		
15	1,3,15	0 01 1111
16	1,2,4,16	0 00 000 10000
...		

## Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - ▶ how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ▶ ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

# Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - ▶ how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ▶ ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

## Decoding method: Input variable-length codeword of $N$

- 1 Input: codeword[1...]

Example worked out during the lecture.

## Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - ▶ how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ▶ ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

### Decoding method: Input variable-length codeword of $N$

- 1 Input: `codeword[1...]`
- 2 Initialize: `readlen` =  $(1)_{dec}$ , **component** = <EMPTY>, `pos` = 1

Example worked out during the lecture.

# Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - ▶ how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ▶ ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

## Decoding method: Input variable-length codeword of $N$

- 1 Input: `codeword[1...]`
- 2 Initialize: `readlen` =  $(1)_{dec}$ , **component** = <EMPTY>, `pos` = 1
- 3 **component** = `codeword[pos...pos + readlen - 1]`.

Example worked out during the lecture.



# Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - ▶ how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ▶ ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

## Decoding method: Input variable-length codeword of $N$

- 1 Input: `codeword[1...]`
- 2 Initialize: `readlen` =  $(1)_{dec}$ , **component** = <EMPTY>, `pos` = 1
- 3 **component** = `codeword[pos...pos + readlen - 1]`.
- 4 If the **most-significant** bit of **component** is **1**, then  
 $N = (\text{component})_{dec}$ . **STOP**.

Example worked out during the lecture.

# Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

## Decoding method: Input variable-length codeword of $N$

- Input: codeword[1...]
- Initialize:  $readlen = (1)_{dec}$ , **component** = <EMPTY>,  $pos = 1$
- component** = codeword[ $pos \dots pos + readlen - 1$ ].
- If the **most-significant** bit of **component** is **1**, then  $N = (\text{component})_{dec}$ . **STOP**.
- Else, if the **most-significant** bit of **component** is **0**, then flip **0**  $\rightarrow$  **1** and reset  $pos = pos + readlen$ ,  $readlen = (\text{component})_{dec} + 1$ .

Example worked out during the lecture.

# Elias code for universal integers – Decoding example

- When decoding a variable-length encoded integer, we do **not** know *a priori*:
  - how many **length components**,  $L_k, L_{k-1}, \dots, L_1$ , the code word has...
  - ...and what  $N$  is.
- We also know is that the very last **length component**,  $L_k$ , in the code word has length  $= (1)_{dec}$ , and it is encoded with 0 bit. (See slide 30.).

## Decoding method: Input variable-length codeword of $N$

- Input: codeword[1...]
- Initialize:  $readlen = (1)_{dec}$ , **component** = <EMPTY>,  $pos = 1$
- component** = codeword[ $pos \dots pos + readlen - 1$ ].
- If the **most-significant** bit of **component** is **1**, then  $N = (\text{component})_{dec}$ . **STOP**.
- Else, if the **most-significant** bit of **component** is **0**, then flip **0**  $\rightarrow$  **1** and reset  $pos = pos + readlen$ ,  $readlen = (\text{component})_{dec} + 1$ .
- Repeat from step (2), until  $N$  is decoded (when step (3) is true).

Example worked out during the lecture.

## Lempel Ziv algorithms

# Lempel-Ziv (LZ77) algorithm

## LZ77

- LZ77 is a **sliding window** based algorithm.
- Since original publication, it inspired many variants (eg. LZSS)
- Used in many applications: gzip, **PKZIP** etc.

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - 1 search window (also called the 'dictionary')



## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - 1 search window (also called the 'dictionary')
  - 2 lookahead buffer (sometimes simply called the 'buffer')

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the dictionary (i.e., search window).

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the dictionary (i.e., search window).
  - ▶ This yields three pieces of information:

## LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the **dictionary** (i.e., search window).
  - ▶ This yields three pieces of information:
    - ① the **offset** (i.e., distance of the match from the current char/substring being encoded).

# LZ77 basic strategy

- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - 1 search window (also called the 'dictionary')
  - 2 lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the **dictionary** (i.e., search window).
  - ▶ This yields three pieces of information:
    - 1 the **offset** (i.e., distance of the match from the current char/substring being encoded).
    - 2 the **length** of the match.

# LZ77 basic strategy

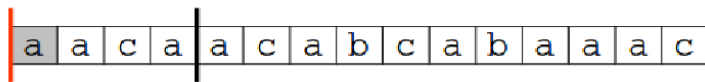
- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the **dictionary** (i.e., search window).
  - ▶ This yields three pieces of information:
    - ① the **offset** (i.e., distance of the match from the current char/substring being encoded).
    - ② the **length** of the match.
    - ③ the next character **char** in the lookahead buffer, after the matched char/substring.


## LZ77 basic strategy


- The LZ77 encoding involves examining the input text through a sliding window.
- The window consists of 2 consecutive parts:
  - ① search window (also called the 'dictionary')
  - ② lookahead buffer (sometimes simply called the 'buffer')
- To encode any char/substring in the lookahead buffer:
  - ▶ find the **largest** matched substring in the **dictionary** (i.e., search window).
  - ▶ This yields three pieces of information:
    - ① the **offset** (i.e., distance of the match from the current char/substring being encoded).
    - ② the **length** of the match.
    - ③ the next character **char** in the lookahead buffer, after the matched char/substring.
- Using this search, the char/substring at the current position is encoded as a **triple**  $\langle \text{offset}, \text{length}, \text{char} \rangle$

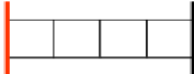



## LZ77 encoding example – explained during lecture



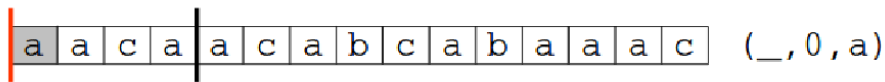
 Dictionary (size = 6)


 Longest match


 Buffer (size = 4)

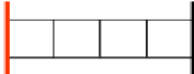
 Next character


## LZ77 encoding example – explained during lecture



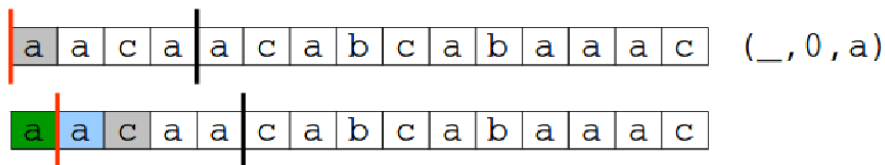
 Dictionary (size = 6)


 Longest match


 Buffer (size = 4)

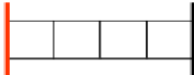
 Next character


## LZ77 encoding example – explained during lecture



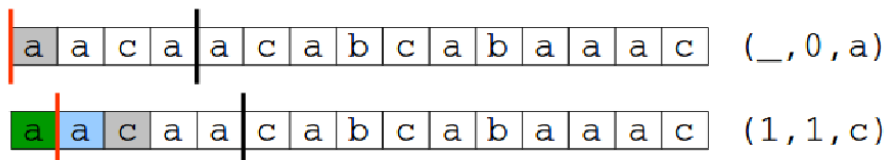
 Dictionary (size = 6)


 Longest match


 Buffer (size = 4)

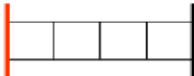
 Next character


## LZ77 encoding example – explained during lecture



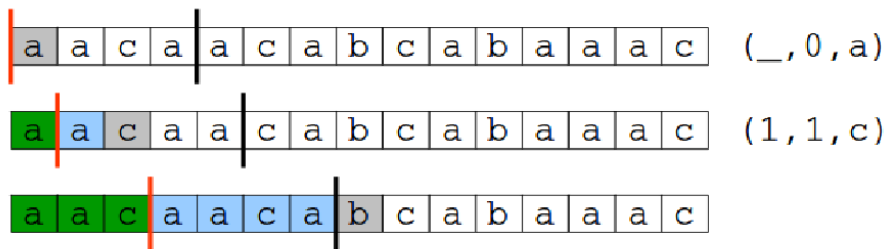
 Dictionary (size = 6)


 Longest match


 Buffer (size = 4)

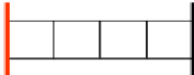
 Next character


## LZ77 encoding example – explained during lecture



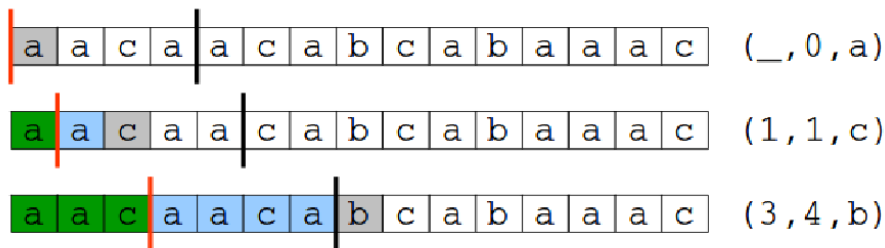
 Dictionary (size = 6)

 Longest match

 Buffer (size = 4)

 Next character

## LZ77 encoding example – explained during lecture



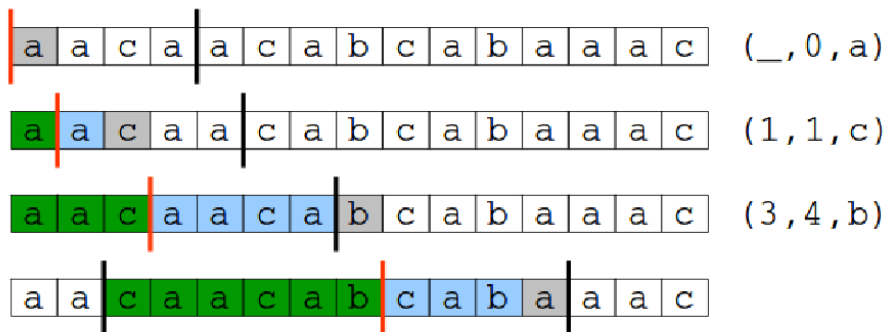
**Dictionary (size = 6)**


**Longest match**


**Buffer (size = 4)**

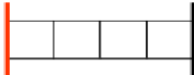
**Next character**


## LZ77 encoding example – explained during lecture



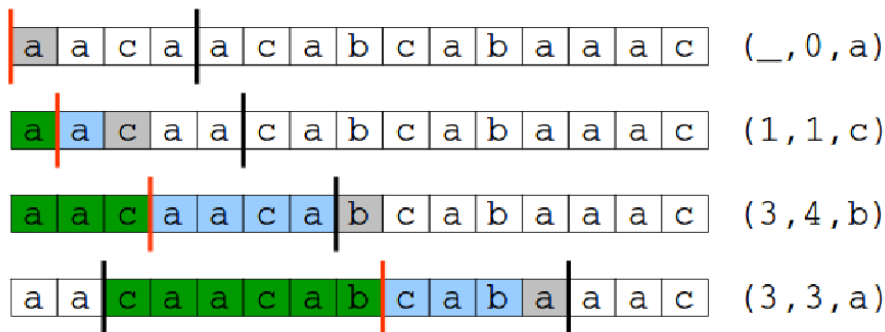
 Dictionary (size = 6)


 Longest match


 Buffer (size = 4)

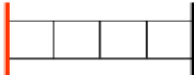
 Next character


## LZ77 encoding example – explained during lecture



 Dictionary (size = 6)

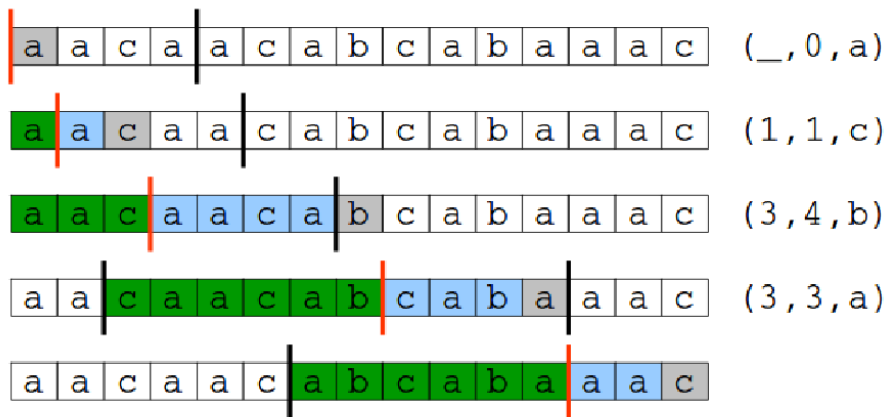
 Longest match


 Buffer (size = 4)


 Next character

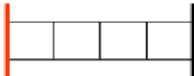



## LZ77 encoding example – explained during lecture



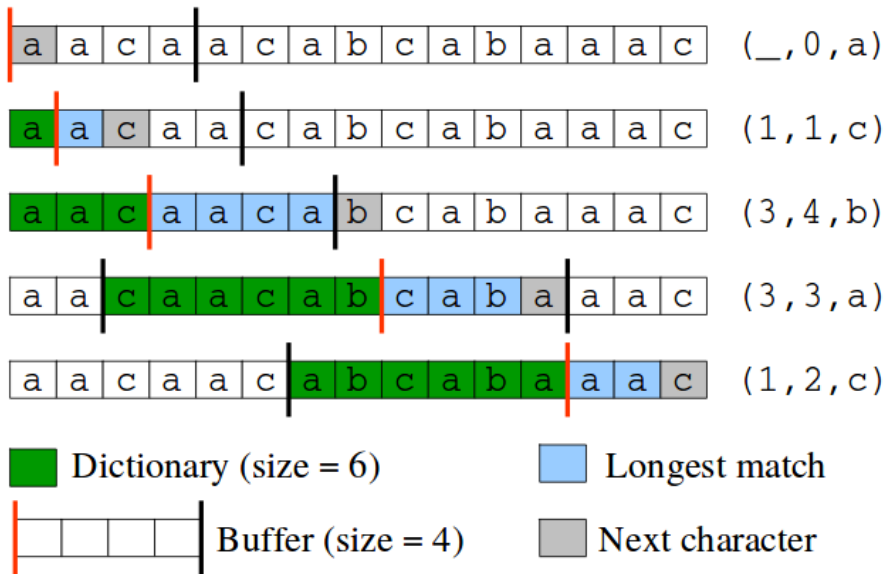
 Dictionary (size = 6)

 Longest match

 Buffer (size = 4)

 Next character

## LZ77 encoding example – explained during lecture



# LZ77 decoding

- Decoding is straightforward using the triple encoding we just saw.
- Using the same sliding window size (=dictionary size + lookahead buffer size), decode the text left to right, using one triple at a time

## Example of decoding one triple

- Assume we are in the middle of decoding, and we have already decoded the string w x y z a b c d  
8 7 6 5 4 3 2 1
- Assume the triple we have at our disposal is  $\langle 2, 9, e \rangle$
- character by character, copy a substring from offset=2 of length=9.
- This further decodes 9 additional characters:  
w x y z a b c d c d c d c d c d c e
- To be discussed in detail during the lecture.

---

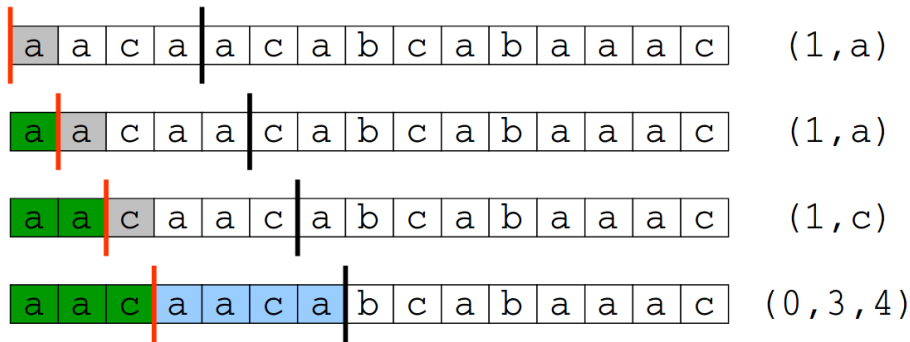
Note: Red numbers on the top indicate offsets in the dictionary.

# LZSS (Lempel-Ziv-Storer-Szymanski) variation of LZ77 algorithm

LZSS variant of LZ77, improves the original LZ77 by reducing the amount of space required to encode short substrings as triples. This is achieved by using two formats for encoding character/substrings during encoding.

- 1 Format 0: When the **length** of the matched substring in the **dictionary**  $\geq 3$ , use:  $\langle 0\text{-bit, offset, length} \rangle$
- 2 Format 1: When the **length** of the matched substring in the **dictionary**  $< 3$ , use:  $\langle 1\text{-bit, char} \rangle$

## LZSS encoding example



... and so on.

## Self-study

During self-study, explore what optimizations can be applied towards implementing an encoder and decoder for LZ77 and LZSS variant, using the variable-length codes we have studied here.

Ask yourself during this if the algorithms we have learnt so far could help optimize your implementation.

Next week

Linear Programming (algebraic and tableau simplex methods)

--o0o--

END

--o0o--