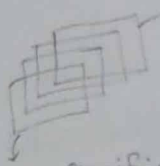
IMAGES

Types of Images

- Bilevel (or) Monochrome Image
- Greyscale Image ↓ Quality depends on the no. of bits to represent each pixel → Number of Bit planes
- Continuous Tone Image ↓ Adjacent pixels differ by a very small value (except for edges) Eg: Natural images
- Discrete Tone Image Eg: Artificial images ↓ Cartoon Image



→ Most Significant Bit Plane (First bit of all pixels)

Least Significant Bit Plane (Last bit of all pixels)

*. Compression can be achieved by removing the least significant bit planes

*. Sampling → Converting continuous data to discrete / digital data
Eg: Taking a photo with a camera

Quantisation → Assigning values to each pixel of an image

* Digital Image ⟶ Transform ⟶ Quantisation ⟶ Coding ↓ Compressed Image

Scalar Quantisation:

8-bit image

11001010 (203)   11001100 (204) (205)   (214)

*. In scalar quantisation, we just remove the last few bits from each pixel.

*. Not an optimal quantisation technique → Causes the formation of prominent edges ⇓ Artificial appearance

11001011   11001100   11001101   11001110   11001111   11010000   11010001

↓ Compression

1100   1100   1100   1100   1100   1101   1101

↓ Decompression

11000000   11000000   11000000   11000000   11000000 ⁝ Strict Boundary 11010000   11010000

Improved Grayscale Quantisation:

S.No.	VALUE	RSM	COMPRESSED STREAM
1	11010101	0000 0000	1101
2	10010111	1001 0111	1001
3	10101010	1011 0001	1011
4	10101111	1011 0000	1011
5	10110000	1011 0000	1011

10101010
0111
10110001

*. If the value of the form 1111 xxxx, simply assign the value to the RSM

Lossless Compression:

→ Based on the assumption: Neighbourhood pixels are the same
⇓
Difference between values is low
→ The difference is stored

Eg: 212 214 216 216 217 215

212 → -2 +2 +2 +1 -2

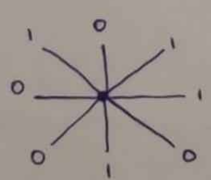
Bilevel Image Compression (Run Length Encoding):

Eg:

000000
011100
111100
001110

Compress → $0^*7 + 1^*3 + 0^*2 + 1^*4 + 0^*4 + 1^*3 \cdots$

Bilevel Image Context Compression:



*. The neighbourhood of a pixel is grouped into a context and represented by n-bits.
↳ No. of neighbours considered

*. Contexts tend to be repeated in an image. The more they are repeated, the lesser the number of bits in the compressed stream to represent that context (Shannon - Fano, Huffman Coding

*. Run length encoding and context encoding can be applied to any image by considering each bit plane of the image to be a bilevel image and encoding all the n- bit planes.

*. Find the average of the context and encode the difference between the average and the pixel.

*. The difference can range from $-(m-1)$ to $+(m-1)$ (Eg: 256)

$$\Delta = \text{Average} - \text{Pixel}$$

*. The difference (Δ) can be coded by arithmetic coding

*. Corner pixels are not averaged and are coded by their pixel values as it is

Transform Based Compression

*. The input spatial domain data is converted into frequency domain

*. This removes the inter-pixel redundancy

Continuous tone coloured images are split into R, G and B planes on which greyscale compression techniques are applied

	R	G	B
Pixel 1	1000	0011	0000
Pixel 2	1001	0011	0000

R: Different values

G, B: Redundancy → Can be compressed

Discrete Tone Image Compression

i)

*. In discrete - toned images, due to self - similarity property, multiple regions are similar.

*. The first unique ~~b~~ region is coded and the following similar regions are made to point to the coded region

ii) Block Similarity:

*. The image is divided into blocks

*. If multiple blocks are similar, the newer block points to the other original blocks that are similar.

JPEG

*. Cannot be used for bilevel images ~~grayscale images~~ → Only 0's and 1s

*. JPEG compression can be used for grayscale and colour images

Merits:

*. JPEG has parameters to decide the quality of the decompressed image.

*. Images decompressed using JPEG, their evaluation ranges from good to excellent.

*. JPEG works efficiently for all image features (aspect ratio, colour space, size)

*. Easily understandable and sophisticated ~~platform~~ technique for all platforms.

JPEG Compression :

- Colour Image
  - Luminiscence → Brightness (Less compression)
  - Chrominiscence → Hue + Saturation (More compression)
    - Hue → Dominant wavelight of the spectrum
    - Saturation → Amount of white light mixed

Image
↓
Down Sampling → Subtract by each pixel by 127 [0, 255] → [0, 127]
↓
DCT → 2D DCT
↓
Quantisation
↓
Coding

Committee of Illumination (CIE) :

*. RGB $\xrightarrow{\text{For more compression}}$ Y C_r C_b

Y → Luminiscence Component (Grayscale)

*. Y-component alone gives the grayscale image, and gives more information.

*. We focus on compressing $C_r C_b$ as they convey lesser information

Ex. RGB to YC_rC_b :

*. $Y = \left(\frac{77}{256}\right) R + \left(\frac{150}{256}\right) G + \left(\frac{29}{256}\right) B$

*. $C_b = -\left(\frac{44}{256}\right) R - \left(\frac{87}{256}\right) G + \left(\frac{131}{256}\right) B + 128$

*. $C_{br} = \left(\frac{131}{256}\right) R - \left(\frac{110}{256}\right) G - \left(\frac{21}{256}\right) B + 128$

$R, G, B \in [0, 255]$

YC_rC_b to RGB :

* $R = Y + 1.371\,(C_r - 128)$
* $G = Y - 0.698\,(C_r - 128) - 0.366\,(C_b - 128)$
* $B = Y + 1.732\,(C_b - 128)$

Mostly,
$Y \in [16, 235]$
$C_r, C_b \in [16, 240]$

2-D DCT

* The image is broken down ~~it~~ into blocks and 2-D DCT is applied to each block.
* DCT cannot be applied to the whole image as it will lead to a large number of computations and the whole image is not always similar. (Blocks have similar pixels due to neighbourhood property)

Forward DCT:

0 to 7 for 8x8 matrix

$$DCT(i,j) = \frac{1}{4}\, C_i\, C_j \sum_{x=0}^{7} \sum_{y=0}^{7} P(x,y) \cdot \left[\frac{\cos (2x+1)\, i\pi}{16}\right] \cdot \left[\frac{\cos (2y+1) \cdot j\pi}{16}\right]$$

where,

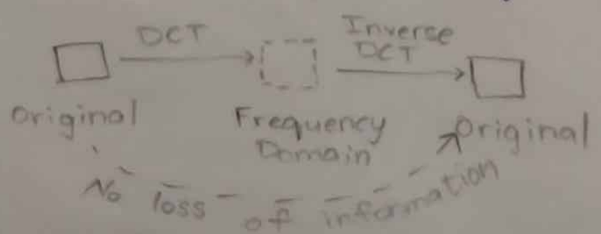
$0 \leq \bigcirc \leq 7$

$$C_f = \begin{cases} \frac{1}{\sqrt{2}} & f = 0 \\ 1 & f > 0 \end{cases}$$

Converting spatial domain data to frequency domain :

$$DCT(i,j) = \frac{1}{\sqrt{2N}}\, C_i C_j \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} P(x,y) \cos\left[\frac{(2x+1) \cdot i\pi}{2N}\right] \cdot \cos\left[\frac{(2y+1) \cdot j\pi}{2N}\right]$$

* DCT does not perform any compression.



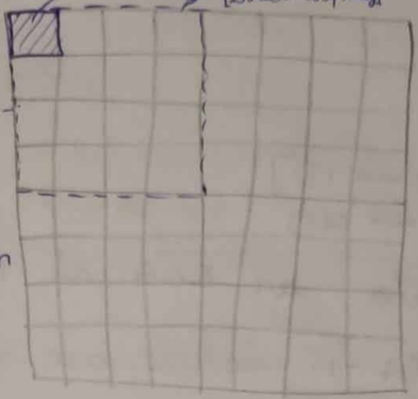
2D Inverse DCT:

$$P(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C_i C_j \; DCT(i,j) \cdot \cos\left[\frac{(2x+1)\, i\pi}{2N}\right] \cdot \cos\left[\frac{(2y+1)\, j\pi}{2N}\right]$$

Average of the 8×8 pixels obtained by DCT

DC (Rest 63 are AC)
[Direct Coupling]

Significant Data
⇓
Lesser Quantisation



8×8

Quantisation:

*. JPEG → Default Quantisation Table
→ $1 + (i+j) * R$

*. When $R = 2$,

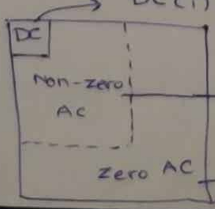
Lesser Quantisation

1	3	5	7	9	11	13	15
3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29

*. After quantisation, more non-zero AC coefficients will appear in the upper left corner and more zero AC coefficients will appear ~~around~~ the towards the bottom left corner.

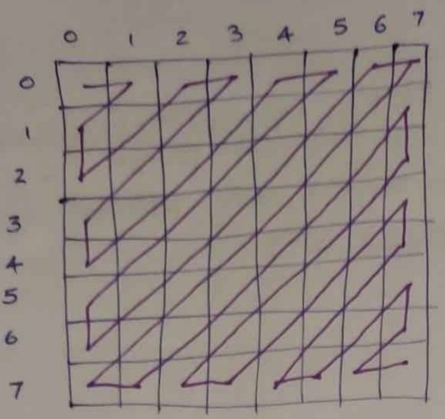
*. Run length encoding is performed to this data.

$DC(i) = DC(i) - DC(i-1)$, $i = 2, 3, \dots$ {DPCM} Difference Pulse Code Modulation



→ Huffman Coding

→ Run length encoding



Zig-Zag Scan [2D → 1D]

8x8 → 64x1

*. In the one-dimensional data, an EOB at n^{th} position (1010) implies that the remaining 64 - n (minus) values are zeroes.

$\{-(2^i-1) \text{ to } +(2^i-1)\} - \{-(2^{i-1}-1) \text{ to } +(2^{i-1}-1)\}$

→ The values in the range will be code with i bits

i	Possible Values for difference between DC components	Unary Code
0	0	0
1	-1, +1	10
2	-3, -2, 2, 3	110
3	-7, -6, -5, -4, 4, 5, 6, 7	1110
4		
⋮		
16	32768	1111111111111111

Column Number

Eg: 1118, -2, -7

1118 will be present in some i^{th} row of the above table

∴ The unary code will have i 1's followed by a 0.

R = i [Row Number]

~~C = Unary Code [i] Colum~~

C = Column Number [1118] in ~~ith row~~ the row corresponding to i

-2 is present at i = 2

∴ ~~C = 110~~ C = 2

R = i = 2

110|010

For -7, $i = 3$

~~C = 1110~~

$R = i = 3$

$C = 1$

$(R, C) = (3, 1)$

1110 | 001

Unary code | R-bit representation of C

For AC components,

Row	0	1	2	3	4
0	10	011	01	110	
1	110	101	101	1110	
2	10101	1110	110	1111	
3					
4					
⋮					
16					

(R, C)
↓
(R, Z) → No. of zeros preceding the non-zero AC component

Eg: $\overset{DC}{1118}$ $\overbrace{-2 \quad 0 \quad 2 \quad 0 \quad 0 \quad 0 \quad 2}^{RC}$

For -2,

$i = 2 \Rightarrow R = 2$ } From previous table
$C = 2$

$Z = 0$ [No zeros before -2]

$(R, Z) = (2, 0) \rightarrow 1011$ (From current table)

$(R, C) = (2, 2) \rightarrow 10$

$\therefore$ 1011 | 10

For 2,

$R = 2$

$c = 3$

$z = 1$ [1 zero before 2]

~~(R, C) → (2, 3) => 11~~

$(R, Z) \rightarrow (2, 1) \Rightarrow 1110$

$(R, C) \rightarrow (2, 3) \Rightarrow 11$

$\therefore$ 1110|11

For next 2,

$R = 2$

$C = 3$

$z = 3$

$(R, Z) \rightarrow (2, 3) \Rightarrow 1111$

$(R, C) \rightarrow (2, 3) \Rightarrow 11$

$\therefore$ 1111|11

---

*. Without JPEG compression,

$$\underbrace{64}_{\text{No. of pixels}} \times \underbrace{24}_{\text{No. of bits for 1 pixel}} = 1536 \text{ bits}$$

*. with JPEG,

$$\underbrace{23}_{\text{23 bits to encode DC}} + \underbrace{23}_{\text{23 bits to encode remaining 63AC components}} = 46 \text{ bits}$$

For 3 planes,

$46 \times 3 = 138$ bits

*. Compression Ratio = 1536/138 ≈ 11

VIDEO COMPRESSION

Video Compression

- Temporal Redundancy
- Spatial Redundancy
  - Coding
  - Inter-Pixel
  - Psychovisual

emporal Dependency:

Independent
*. I - frame → Coded independently

Predicted
*. P- frame → Coded based on previous I or P-frame

*. B - frame → Coded with previous and successive P or I-frames

Sub-Sampling:

*. Selecting certain frames (Alternate frames)

Differencing:

*. Difference between two consecutive frames is calculated in order to find the difference matrix.

*. Difference matrix will have lots of zeros (Consecutive frames are generally very similar).

*. The difference value and its position in the matrix alone need to be shared.

~~The image~~ is

Block Differencing:

*. The image is divided into blocks and differencing is applied to these blocks.

Motion Compensation: (For object translation) [Cannot be used for rotation and scaling]

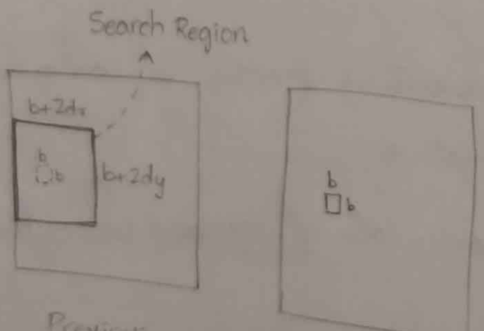
$$(c_x - b_x,\ c_y - b_y) = \underbrace{(\Delta x, \Delta y)}_{\text{Motion Vector}}$$

c_x: x-coordinate in previous frame
b_x: x-coordinate in current frame
c_y: y-coordinate in previous frame
b_y: y-coordinate in current frame

*. The image is divided into blocks of optimal size.
*. Large blocks do not match often.
*. Small blocks do not offer much compression
*. Threshold → Error tolerance between similar blocks

Block Search Procedure:

*. The block might have translated from its position in the previous frame.
*. Therefore, the block is searched in the range: $(b + 2dx,\ b + 2dy)$.



Previous Frame

Current Frame

*. Distortion Measure

$$\text{Mean Absolute Pixel Difference} = \frac{1}{b^2} \sum_{i=1}^{b} \sum_{j=1}^{b} (c_{ij} - b_{ij})$$

c_{ij}: Pixel in previous frame
b_{ij}: Pixel in current frame

(x_2-x_1, y_2-y_1)

*. If a match is found for the block, a motion vector is calculated by comparing the top left corners of the current and previous frams.

*. A frame is predicted with the help of the motion vector

*. If there is no match for a block, then it is coded independently.

H261 Standard (First video compression technique)

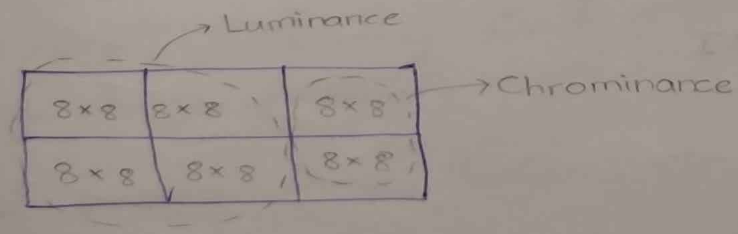
*. The image is divided into blocks

*. The predicted image is calculated.

*. If the block size is too large, ~~we~~ multiple objects might be in the same block, but search computation is reduced.

*. If the search area is reduced, more number of blocks will be coded independently.

*. H-261 makes use of macro blocks.



Macro block

*. H-261 is used in video calling and video conferencing.

*. In H-261, all subsequent frame depend on the first I and P frames. Therefore H-261 is not suitable for storing videos (More I-frames can solve this problem)

Filtering:

*. Remove the noise in the image and abrupt frequencies.

Abnormal Frequencies / Noise

Eg:

110	218	110
111	222	111
101	222	112

$$\frac{1}{4} \times 110 + \frac{1}{2} \times 218 + \frac{1}{4} \times 110$$

MPEG-I (Moving Picture Expert Group)

I → Independent frame or Intraframe (Intracoding)

P → Predictive Coded frame (Coded based on previous I/P frame)

B → Bidirectional Predictive Coded frame (Coded based on previous I/P frame as well as successive I/P frame)

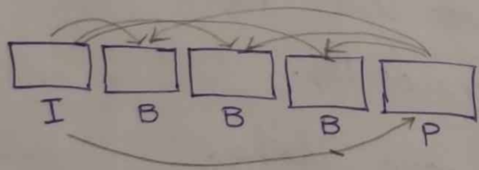
↓

Not used for prediction

⇓

Prevents error propagation

Group of Pictures (GOP):



Display Order:

Eg: I B B P B B P B B P B B I

1 2 3 4 5 6 7 8 9 10 11 12 13

Processing Order:

1, 4, 2, 3, 7, 5, 6, 10, 8, 9, 13, 11, 12

I P B B P B B P B B I B B

*. Decoder gets the frames in the processing order.

*. More B frames ⇒ Consecutive frames are very similar for a longer duration

*. As the distance of the B frame increases from the reference frame (I/P), the search area needs to be increased

*. In MPEG-I, the frames must be of dimensions 768 × 576

*. 396 macro-blocks for 25 FPS or lesser

*. 330 macro-blocks for 30 FPS or lesser

*. MPEG-I is suitable for slow moving images. It is not suitable for quick scene changes

STATISTICAL MODELLING

More compression | More probability tables ⇓ Adaptive Modelling (More compression but lots of computations)

order 0 → Consider only current symbol ~~(1 table)~~ (1 row)

order 1 → Consider previous symbol also (256 rows)

order 2 → Consider 2 previous symbols (256^2 rows)

Finite Context Modelling:

*. The probability of occurrence of a symbol is calculated based on the context.

Dictionary Based Compression:

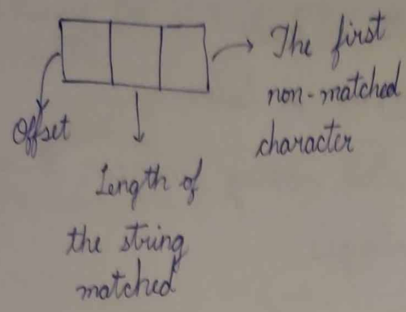
Dictionary → Static (No deletions), Dynamic

Static Dictionary:

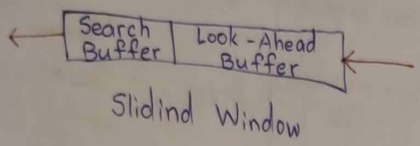
*. Can compress to the entropy when the data is large.
(If lesser data, dictionary itself will occupy large space nullyfying the compression)

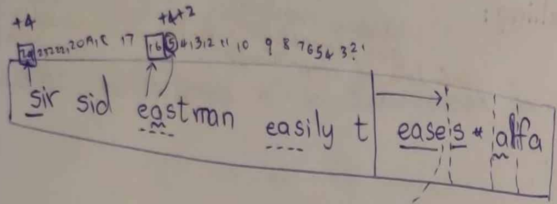
*. General purpose large static dictionaries may not provide compression as good as domain specific static dictionaries

Dynamic (Adaptive) Dictionary:



Sliding Window Algorithm (LZ77 Algorithm):



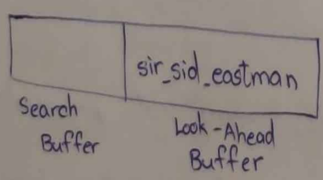


16	3	'e'

28	1	'_'

21	1	'l'

Eg: sir_sid_eastman



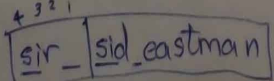
0	0	's'

∵ 's' is not present in search buffer

0	0	'i'

0	0	'r'

0	0	'_'



4	2	'd'

4 3 2 1

sir_sid	_eastman

4	1	'e'

sir_sid_e	astman

0	0	'a'

10	1	't'

0	0	'm'

4 3 2 1

sir_sid_eastm	an

4	1	'n'

sir_sid_eastman	

* Search buffer → 1000 bytes
* Look-ahead buffer → 10 bytes

Example Values

Search Buffer → 8000 bits ⇒ $\underbrace{\text{Offset max value} = 8000}_{\text{Bits required} = \log_2(8000)}$

Look Ahead Buffer → 80bits ⇒ $\underbrace{\text{Length of match max value} = 80}_{\text{Bits required} = \log_2(80)}$

Non-match character ⇒ 8

* This method provides good compression when the same sequence of symbols occur in a concentrated manner

Ⓠ ALFALFALFAL

0	0	A

0	0	L

0	0	F

3	7	L

∵ Search buffer can never be the complete string

Ⓠ AAAAAAAAAA

0	0	A

1	8	A

LZSS ALGORITHM:

Fixed Size

Search Buffer - 1 byte
Look-Ahead Buffer - 1 byte } Example

~~Token (off~~

*Repeating symbols (len_match > 2)

Token(1, Offset, Length) $\longrightarrow$ $1 + 8 + 8 = 17$ bits

*New symbol:

(0, 'x') $\longrightarrow$ $1 + 8 = 9$ bits

*Repeating symbol (len_match = 1)

(0, 'x') $\longrightarrow$ 9 bits < 17 bits

*For len_match = 2,

If coded as ~~rep~~, new symbols,

(0, 'x') + (0, 'y') $\longrightarrow$ $(1+8) + (1+8) = 18$ bits

If coded as repeating symbols,

(1, n, 2) $\longrightarrow$ $1 + 8 + 8 = 17$ bits

The advantage of coding as repeated symbols is only bit. This advantage is insignificant when compared to the computations involved in encoding and decoding the matching symbols. Therefore, if the match is of length only 2, the symbols will be encoded ~~only~~ as new symbols.

Eg: LZSS

"MISS_MISSISSIPPI"

Search Buffer → 16 bytes ⇒ 4 bits

LA Buffer → 16 bytes ⇒ 4 bits

|MISS_MISSISSIPPI

| 0 | M | (~~5~~ 9 bits)

| 0 | I | (~~5~~ 9)

| 0 | S | (~~5~~ 9)

len_match = 2

New: 9+9=18 ✓

~~5+5=10~~

Repeat:

1+4+4 = 9 ✓

MIS|S_MISSISSIPPI

~~| | | |~~ | 0 | S | (~~5~~ 9)

More Compression

len_match = 3

New: 9+9+9 = 27

~~5+5+5=15~~

Repeat:

1+4+4 = 9 ✓

MISS|_MISSISSIPPI

| 0 | _ | (~~5~~ 9)

MISS_|MISSISSIPPI

| 1 | 5 | 4 | (9 bits)

4, 4, 1

MISS_MISS|ISSIPPI

| 1 | 8 | 3 | (9)

MISS_MISSISS| IPPI

| 0 | I | (~~5~~ 9)

| 0 | P | 9 (~~5~~)

$\boxed{O|P}$ $\overset{9}{(5)}$

$\boxed{O|I}$ $\overset{9}{(5)}$

LZ 78 Algorithm

Eg: Good morning

Dictionary

Index	String
0	NULL
1	G
2	o
3	od
4	_
5	m
6	or
7	n
8	i
9	ng

Output String:

(0, 'G')

(0, 'o')

(2, 'd')

(0, '_')

(0, 'm')

(2, 'r')

(0, 'n')

(0, 'i')

(7, 'g')

(Index, Character)
↳ 8 bits

*. The dictionary is not sent to the decoder. Only the output string is sent to the decoder.

* Decoding,

→ (0, 'G')

⇒ G

→ (0, 'o')

⇒ Go

→ (2, 'd')

⇒ Good

Dictionary

0	NULL
1	G
2	o
3	od

→ (0, '_')

⇒ Good_

→ (0, 'm')

⇒ Good m

→ (2, 'r')

⇒ Good mor

→ (0, 'n')

⇒ Good morn

→ (0, 'i')

⇒ Good morni

→ (7, 'g')

⇒ Good morning

Dictionary

Index	String
0	NULL
1	G
2	o
3	od
4	_
5	m
6	or
7	n
8	i
9	ng

*. If the last symbol matches, the corresponding output string will have the index alone (There is no character to follow).

Eg: aababacbacacabac → $8 \times 16 = 128$ bits

Dictionary

Index	String
0	NULL
1	a
2	ab
3	aba
4	c
5	b
6	ac
7	aca
8	ba

Output String

$\lceil \log_2 9 \rceil$ $\times 8$

(0, 'a')

(1, 'b')

(2, 'a')

(0, 'c')

(0, 'b')

(1, 'c')

(6, 'a')

(5, 'a')

(4)

$\overset{4}{\Vert}$

$(\lceil \log_2 9 \rceil + 8) \times 9$

$= 12 \times 9$

$= 108$

$108 - 8$

∵ Last token does not have a character

∴ 100 bits

Decoding.

→ (0, 'a')

⇒ a

→ (1, 'b')

⇒ aab

→ (2, 'a')

⇒ aababa

→ (0, 'c')

⇒ aababac

→ (0, 'b')

⇒ aababacb

→ (1, 'c')

⇒ aababacbac

→ (6, 'a')

⇒ aababacbac aca

→ (5, 'a')

⇒ aababacbac aca ba

→ (4)

⇒ aababacbacacabac

Dictionary

Index	String
0	NULL
1	a
2	ab
3	aba
4	c
5	b
6	ac
7	aca
8	ba

LZW Algorithm

Eg : ababac

The dictionary initially has 255 entries for the unique symbols.

a	0
b	1
c	2
.	.
.	.
ab	255
ba	256

→ a ✓

⇒ 0

→ ab ✗

⇒ Add 'ab' to dictionary at ~~256~~ 255

→ b ✓

⇒ 0 1

→ ba ✗

⇒ Add 'ba' to dictionary at 256

→ a ✓

→ ab ✓ ⇒ 0 1 255

→ aba ✗ ⇒ Add aba to dictionary at 257

→ a ✓ ⇒ 0 1 255 0

→ ac ✗ ⇒ Add ac to dictionary at 258

→ c ✓ ⇒ 0 1 255 0 2

∴ Encoded string : 0 1 255 0 2

Decoding :

→ 0 ⇒ a

→ 1 ⇒ ab ~~→ Add 'ab' to dictionary at 255~~

→ 255 ~~⇒ abab~~ $\xrightarrow{\text{Not in dictionary}}$ Add 'ab' to dictionary at 255

⇒ abab

→ 0 ⇒ ababa

→ 2 ⇒ ababc

Eg: ababbabbbabbbb

→ a ✓
→ ab × → Add 'ab' to the dictionary at 255
⇒ 0
→ b ✓
→ ba × 256 : ba
⇒ 0 1

~~→ b ✓~~
~~→ bb × 257 : bb~~
~~⇒ 0 1 1~~
~~→~~

→ a ✓
→ ab ✓
→ abb × 257 : abb
⇒ 0 1 255

→ b ✓
→ ba ✓
→ bab × 258 : bab
⇒ 0 1 255 256
→ b ✓
→ bb × 259 : bb
⇒ 0 1 255 256 1
→ b ✓
* → bb ✓
→ bba × 260 : bba
* ⇒ 0 1 255 256 1 259
→ a ✓
→ ab ✓
* → abb ✓
→ abbb × 261 : abbb
⇒ 0 1 255 256 1 259 257
* → b ✓
→ bb ✓
⇒ 0 1 255 256 1 259 257 259

*. LZW may lead to a lot of additional entries in the dictionary for each new substring.

*. If the dictionary size becomes very large, the number of bits required to represent string will also increase, resulting in lesser or even no compression.

LZMW Algorithm:

Assume, Existing Dictionary

S	100
W	101
I	102
_	103
M	104

Eg: SWISS_MISS

INPUT SYMBOL	CURRENT SYMBOL	OUTPUT	ADD TO DICTIONARY
		100	—
S	S		
W	W	101	256 : SW
I	I	102	257 : WI
S	S	100	258 : IS
S	S	100	259 : SS
_	_	103	260 : S_
M	M	104	261 : _M
I	IS	258	262 : MIS
S	S	100	263 : ISS

Decoding: 100 101 102 100 100 103 104 258 100

INCOMING SYMBOL	OUTPUT STRING	ADD TO DICTIONARY
100	S	256 : SW
101	W	257 : WI
102	I	258 : IS
100	S	259 : SS
100	S	260 : S_
103	_	261 : _M
104	M	262 : MIS
258	IS	263 : ISS
100	S	

All Prefix

LZAP Algorithm :

Existing Dictionary

100 : S
101 : W
102 : I
103 : _
104 : M

Eg: SWISS_ MISS

INPUT	CURRENT	OUTPUT	ADD TO DICTIONARY
		100	–
S	S	101	256 : SW
W	W	102	257 : WI
I	I	100	258 : IS
S	S ~~100~~	100	259 : SS
S	S ~~100~~	103	260 : S_
_	_	104	261 : _M
M	M	258	262 : MI~~S~~ 263 : MIS
I	IS		
			264 : ISS
S	S	100	

Q Compress YABBADABBADABBADOO using LZMW and LZAP

Let the existing dictionary be :

12 bits

100 : Y
101 : A
102 : B
103 : D
104 : O

LZMW :

INPUT	CURRENT	OUTPUT	ADD
Y	Y	100	–
A	A	101	256 : YA
B	B	102	257 : AB
B	B	102	258 : BB
A	A	101	259 : BA
D	D	103	260 : AD
A	AB	257 ~~101~~	261 : DAB
B	BA	259	262 : ABBA

D	DA	103	263 : BAD
A	ABBA	262	264 : DABBA
D	D	0103	265 : ABBAD
O	O	104	266 : DO
O	O	104	267 : OO

LZAP :

INPUT	CURRENT	OUTPUT	ADD
Y	Y	100	—
A	A	101	256 : YA
B	B	102	257 : AB
B	B	102	258 : BB
A	A	101	259 : BA
D	D	103	260 : AD
A	AB	257	261 : DAB 262 : DAB
B	BA	259	263 : ABA 264 : ~~ABAB~~ ABBA 265 : ABBA
D	DAB	262	