Multimedia Data Compression

Part I

Chapter 7
Lossless Compression Algorithms

Chapter 7
Lossless Compression Algorithms

1. Introduction
2. Basics of Information Theory
3. Lossless Compression Algorithms
    - ❑ Fix-Length Coding
        - ❑ Run-length coding
        - ❑ Differential coding
        - ❑ Dictionary-based coding
    - ❑ Variable - Length Coding
        - ❑ Shannon-Fano Algorithm
        - ❑ Huffman Coding Algorithm

Introduction

- A digital file (Video, Image and Audio) can easily become very large

  → ***we need Huge volume of multimedia data***

- It can be useful or necessary to compress them for ease of storage or delivery

  → ***we need more efficient data storage, processing and transmission***

- However, while compression can save space or assist delivery, it can *slow or delay the opening of the file*, since it must be decompressed when displayed.

- Some forms of compression will also *compromise the quality* of the file.

Introduction

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

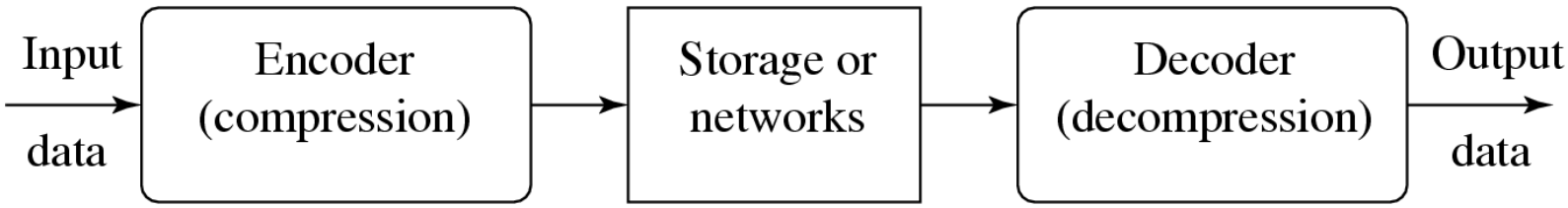


Fig. 7.1: A General Data Compression Scheme.

Introduction

- Compression ratio denotes the relation between the size of the original data before compression and the size of the compressed data. Compression ratio therefore rates the effectivity of a compression system in terms of data reduction capability.

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

B_0 – number of bits before compression

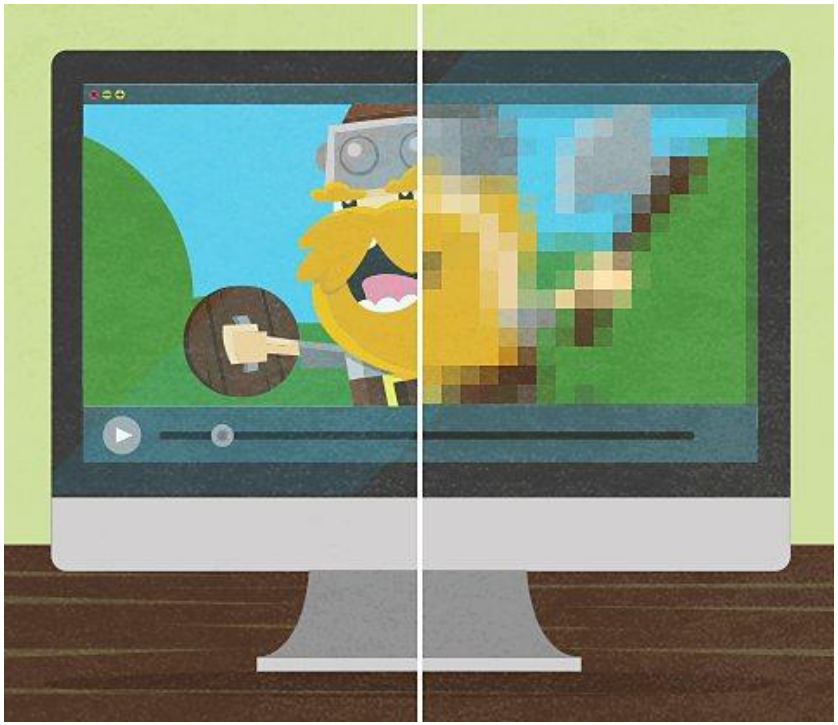
B_1 – number of bits after compression

- In general, we would desire any codec (encoder/decoder scheme) to have a compression ratio much larger than 1. The higher the compression ratio, the better the lossless compression scheme.

Introduction

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

- Lossy compression means that the quality of your file will be reduced. The right side of this image has been saved using lossy compression and doesn't look as good as the left.



Introduction

Lossless Compression Algorithms

- There will be no data loss in this type of compression as it is defined by the name. Both original data and the compressed data are the same in this compression. The algorithms for the compression and decompression are exact inverse of each other in the Lossless Compression. The main mechanism in this compression is removing the redundant data in the compression and adding them in the decompression.

- **Advantages**: The original format of the data remains even it is compressed.
- **Disadvantages**: Reduction of the size of the data is a small. Sometimes the size can increase instead of decrease.

Lossy Compression Algorithms

- It is the compression technique which will lose data in the original source while trying to keep the visible quality at the almost same amount. The compression ratio will be very high. Most probably the ratio will be a value near 10. It reduces non sensitive information to the human eyes and the compressed media will not be the media that was available before compression.

- **Advantages**: Can reduce the file size more than in the Lossless Compression
- **Disadvantages**: The original file cannot be taken after the decompression

Motivating Example

Transmit the data {250, 251, 251, 252, 253, 253, 254, 255} by the network

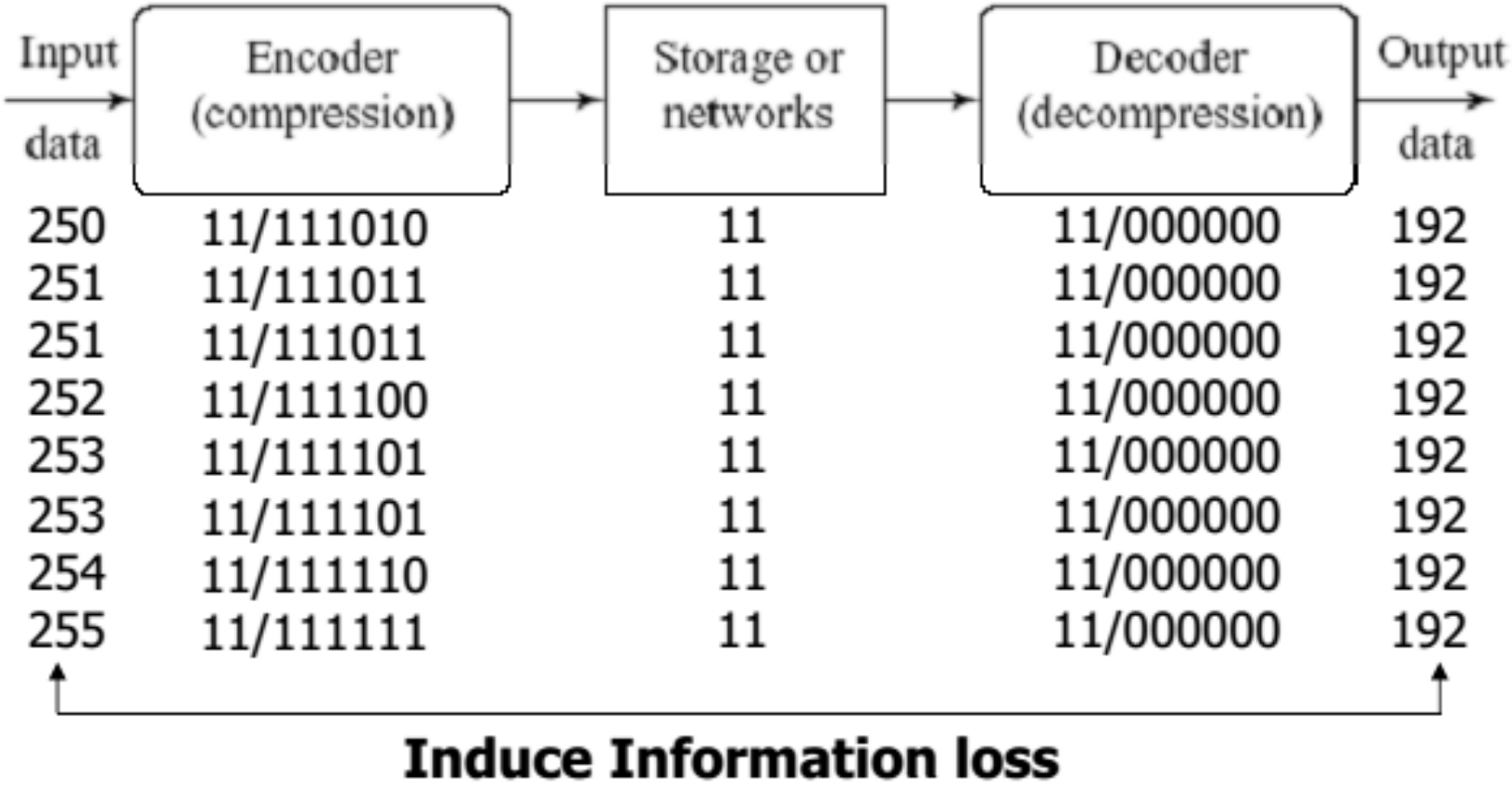
- Rewrite the data sequence using binary vector {11111010,
- 11111011 ,11111011 ,11111100 ,11111101 ,11111101,11111110, 11111111}
- Totally require 8*8-bit = **64 bits** for transmission

The available bandwidth is limited!!

- **Only 16 bits available** for the transmission of the data
- → Compression is necessary!!

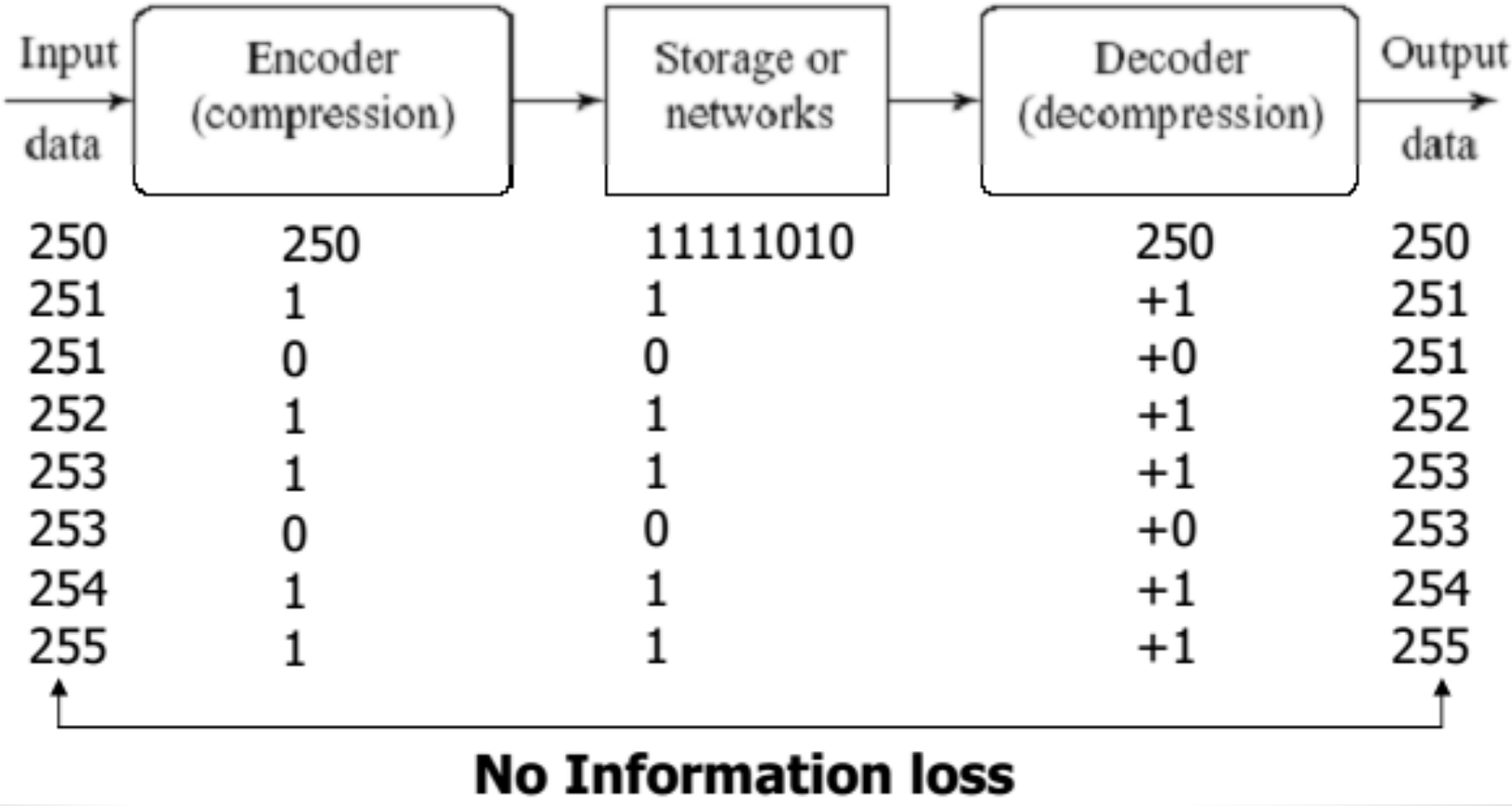
Lossy Compression

- Encode: drop the least significant bits
- Encode data: 8 * 2-bit = **16 bits**



Lossless Compression

- **Encode**: encode the difference
- **Encode data**: 8-bit + 7*1-bit = **15 bits**



Lossless Compression Algorithms

Basics of Information Theory

Entropy

- The ***entropy*** η of an information *source* with alphabet $S = \{s_1, s_2, \ldots, s_n\}$ is:

$$\eta = H(S) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \qquad (7.2)$$

$$= -\sum_{i=1}^{n} p_i \log_2 p_i \qquad (7.3)$$

p_i – probability that symbol s_i will occur in S.

$\log_2 \frac{1}{p_i}$ – indicates the amount of information (self information as defined by Shannon) contained in s_i, which corresponds to the number of bits needed to encode s_i.

Basics of Information Theory
Distribution of Gray-Level Intensities Example

- For example, in an image with uniform distribution of gray-level intensity, i.e. pi = 1/256, then the number of bits needed to code each gray level is 8 bits. The entropy of this image is 8.

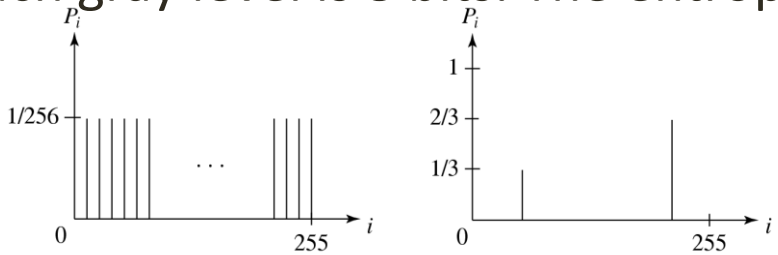


Fig. 7.2 Histograms for Two Gray-level Images.

- Fig. 7.2(a) shows the histogram of an image with *uniform* distribution of gray-level intensities, i.e., $\forall i\ p_i = 1/256$. Hence, the entropy of this image is:

$$\eta = \sum_{i=0}^{255} \frac{1}{256} \cdot \log_2 256 = 8 \tag{7.4}$$

- Fig. 7.2(b) shows the histogram of an image with two possible values. Its entropy is 0.92.

Basics of Information Theory

Entropy and Code Length

- The most significant interpretation of entropy in the context of lossless compression is that entropy represents **the *average* amount of information contained (in bits) per symbol in the source *S*. (bits/symbol),** i.e. to conduct lossless compression.

- Entropy therefore gives a **lower bound** on the number of bits required to represent the source information, i.e. the minimum amount of bits can be used to encode a message without loss.

Basics of Information Theory

Entropy Example

A. What is the entropy of the image below, where numbers (0, 20, 50, 99) denote the gray-level intensities?

99	99	99	99	99	99	99	99
20	20	20	20	20	20	20	20
0	0	0	0	0	0	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	0	0	0	0	0	0

Entropy Example

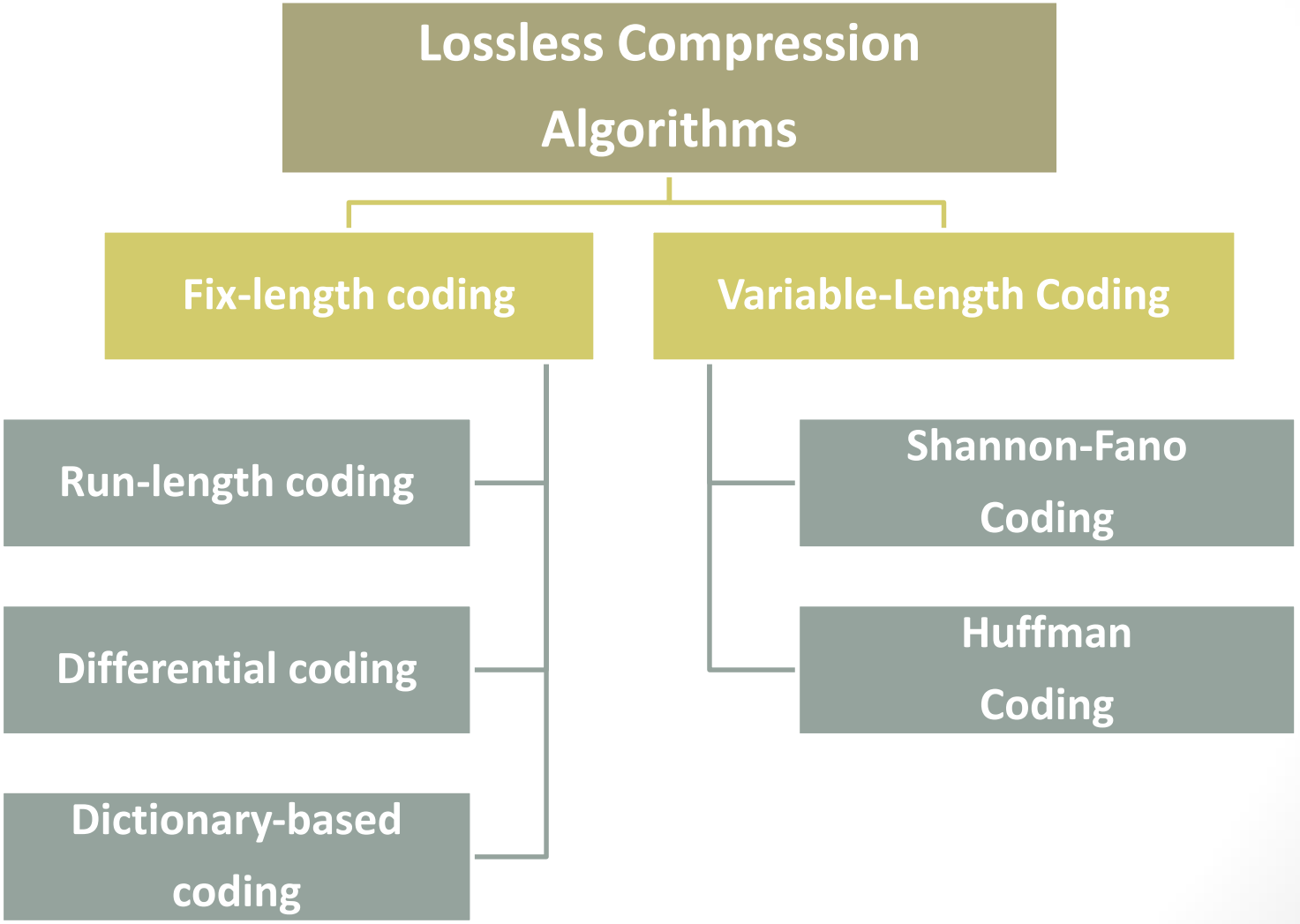
99	99	99	99	99	99	99	99
20	20	20	20	20	20	20	20
0	0	0	0	0	0	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	0	0	0	0	0	0

Answer:

(a) $P_{20} = P_{99} = 1/8$, $P_{50} = 1/4$, $P_0 = 1/2$.

$$\eta = 2 \times \frac{1}{8}\log_2 8 + \frac{1}{4}\log_2 4 + \frac{1}{2}\log_2 2 = \frac{3}{4} + \frac{1}{2} + \frac{1}{2} = 1.75$$

Lossless Compression Algorithms



Fix-length coding

- The length of the codeword is **fixed**

1. **Run-length coding**
2. **Differential coding**
3. **Dictionary-based coding**

Fix-length coding

1. Run length coding

- **Run-length coding** is a data compression algorithm that helps us encode large runs of repeating items by only sending one item from the run and a counter showing how many times this item is repeated.

- Unfortunately this technique is *useless* when trying to compress *natural language texts*, because they don't have long runs of repeating elements. In the other hand it is useful when it comes to *image compression*, because images happen to have long runs pixels with identical color.

Fix-length coding

1. Run length coding

- **For example**:

The string **AAABBCDDDD** would be encoded as → **3A2B1C4D**.

aaaabbaba → ?? 4a2b1a1b1a

- A real life example where run-length encoding is quite effective is the **fax machine**. Most faxes are white sheets with the occasional black text. So, a run-length encoding scheme can take each line and transmit a code for while then the number of pixels, then the code for black and the number of pixels and so on.

- This method of compression must be used carefully. If there is not a lot of repetition in the data then it is possible the run length encoding scheme would actually increase the size of a file.

Fix-length coding

1. Run length coding (Example)

- **Original data**: AAAAABBBAAAAAAAABBBB

Using ASCII code

- B = 20*8bit = 20 * 1Byte = **20 Bytes**

Character	ASCII Code	Character	ASCII Code	Character	ASCII Code
0	00110000	A	01000001	N	01001110
1	00110001	B	01000010	O	01001111
2	00110010	C	01000011	P	01010000
3	00110011	D	01000100	Q	01010001
4	00110100	E	01000101	R	01010010
5	00110101	F	01000110	S	01010011
6	00110110	G	01000111	T	01010100
7	00110111	H	01001000	U	01010101
8	00111000	I	01001001	V	01010110
9	00111001	J	01001010	W	01010111
		K	01001011	X	01011000
		L	01001100	Y	01011001
		M	01001101	Z	01011010

Run length coding result: 5A3B8A4B

- **Compressed data** = 8 * 1-Byte = **8 Bytes < 20 Bytes** ☺
- **Compression ratio**: 20/8 = 2.5 > 1 → good ☺

Fix-length coding

1. Run length coding (Example)

Extreme cases:

- **Best case**: AAAAAAAA ~ 8A
  - **Compression ratio**: 8/2 = 4
- **Worst case**: ABABABAB ~1A1B1A1B1A1B1A1B
  - **Compression ratio**: 8/16 = 0.5
  - ***Negative compression***: the resulting compressed file is larger than the original one.

Example: Original data: ABABBABCABABBA, use RLC and compute Compression ratio?

Run-length coding: 1A1B1A2B1A1B1C1A1B1A2B1A
Compression ratio: 14/24 ~ negative compression

Fix-length coding

2. Differential coding

- In this method first a reference symbol is placed. Then for each symbol in the data, we place the difference between that symbol and the reference symbol used.

- **For example**, using symbol **A** as reference symbol, the string:

AAABBC DDDD would be encoded as → **A0001123333**

(since A is the same as reference symbol, B has a difference of 1 from the reference symbol and so on.)

Fix-length coding

3. Dictionary based coding

- One of the best known dictionary based encoding algorithms is **Lempel-Ziv (LZ)** compression algorithm.
- In this method, a dictionary (table) of variable length strings (common phrases) is built.
- This dictionary contains almost every string that is expected to occur in data.
- When any of these strings occur in the data, then they are replaced with the corresponding index to the dictionary.

Fix-length coding

3. Dictionary based coding

- In this method, instead of working with individual characters in text data, we treat each word as a string and output the index in the dictionary for that word.

- **For example**, let us say that the word "compression" has the index 4978 in one particular dictionary.

- To compress a body of text, each time the string "compression" appears, it would be replaced by 4978.

Variable-length coding

- The length of the codeword is **variable**

1. **Shannon-Fano Algorithm**
2. **Huffman Coding Algorithm**

Variable-length coding

1. Shannon-Fano Algorithm

Shannon-Fano Algorithm — a t**op-down** approach

1. **Sort** the symbols according to the frequency count of their occurrences.
2. Recursively **divide the symbols into two parts**, each with approximately the same number of counts, until all parts contain only one symbol.

An Example: coding of “HELLO”

Symbol	H	E	L	O
Count	1	1	2	1

Frequency count of the symbols in ”HELLO”.

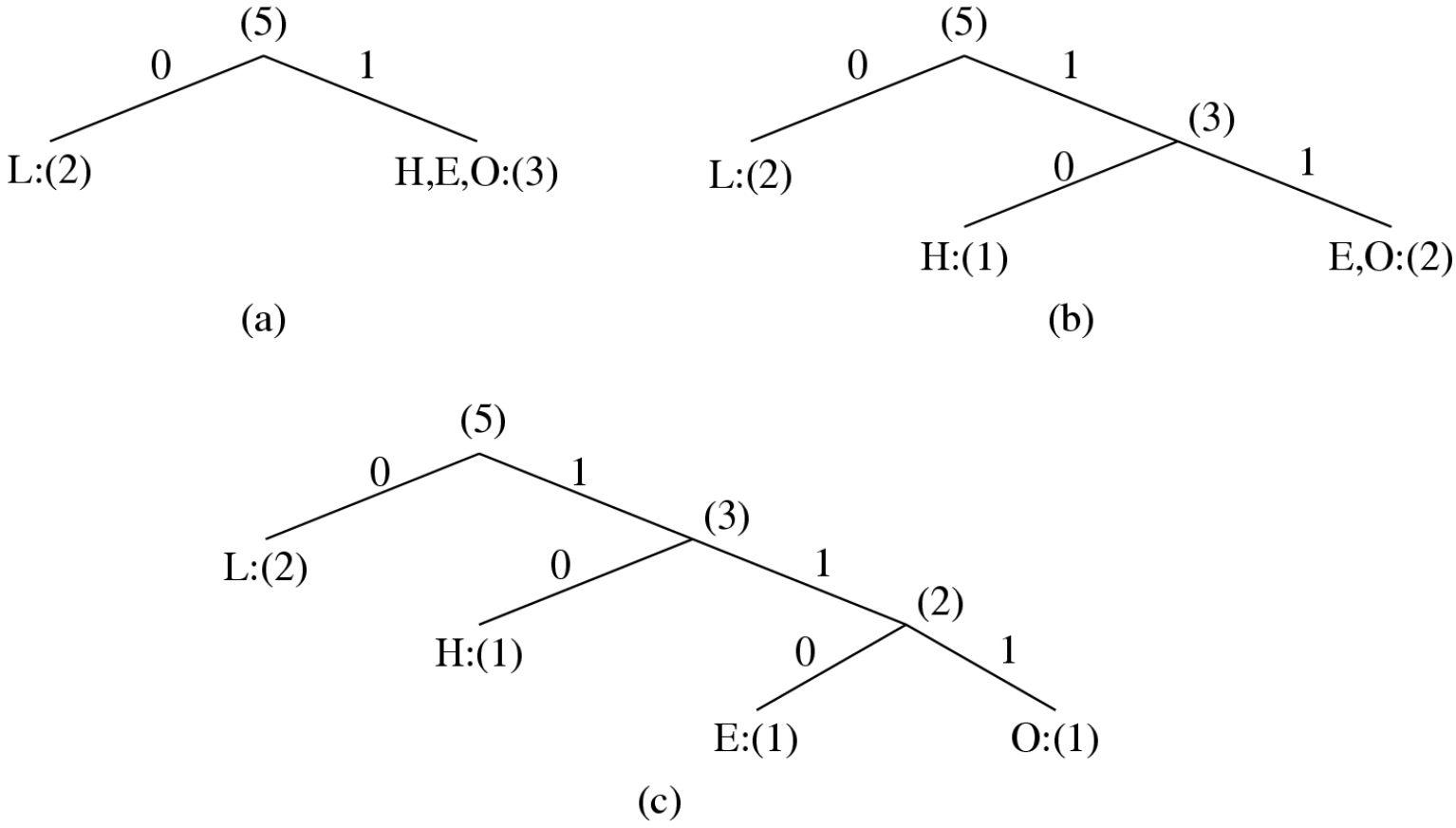


Fig. 7.3: Coding Tree for HELLO by Shannon-Fano.

Table 7.1: Result of Performing Shannon-Fano on HELLO

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

On average it uses 10/5 = 2 bits to code each symbol → close to the lower bound of 1.92 → the result is satisfactory ☺

p_i – probability that symbol s_i will occur in S.

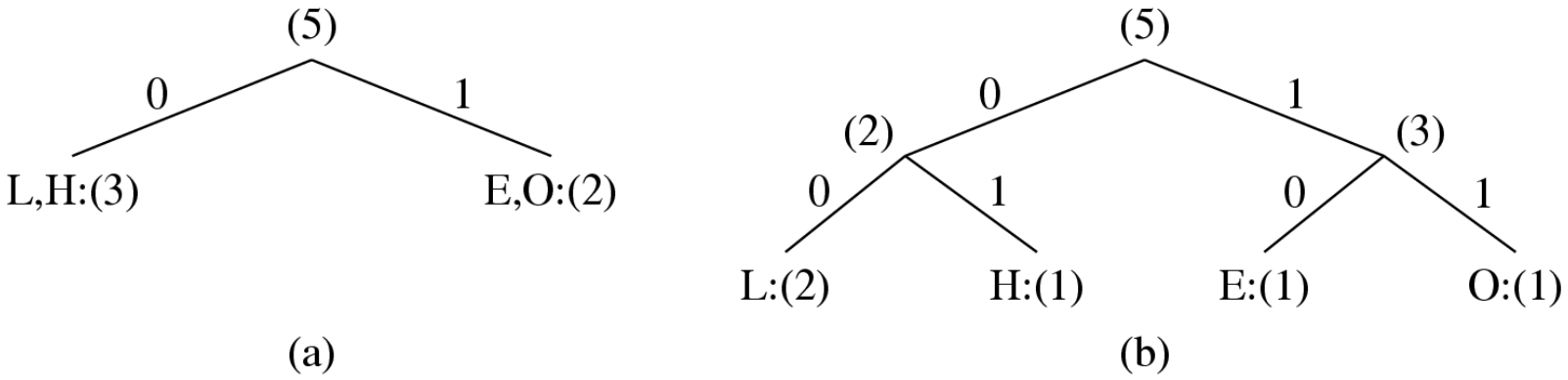
$\log_2 \frac{1}{p_i}$ – indicates the amount of information (self-information as defined by Shannon) contained in s_i, which corresponds to the number of bits needed to encode s_i.

Entropy ŋ ==??

PL . Log 1/PL + PH . Log 1/PH + PE . Log 1/PE + PO . Log 1/PO =

0.4 X 1.32 + 0.2 X 2.32 + 0.2 X 2.32 + 0.2 X 2.32 = 1.92

Fig. 7.4 Another coding tree for HELLO by Shannon-Fano.



Symbol	Count	$\text{Log}_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	00	4
H	1	2.32	01	2
E	1	2.32	10	2
O	1	2.32	11	2
			TOTAL # of bits:	**10**

Example

- Suppose we have a message consisting :

[A(4), B(2), C(2), D(1), E(1)]

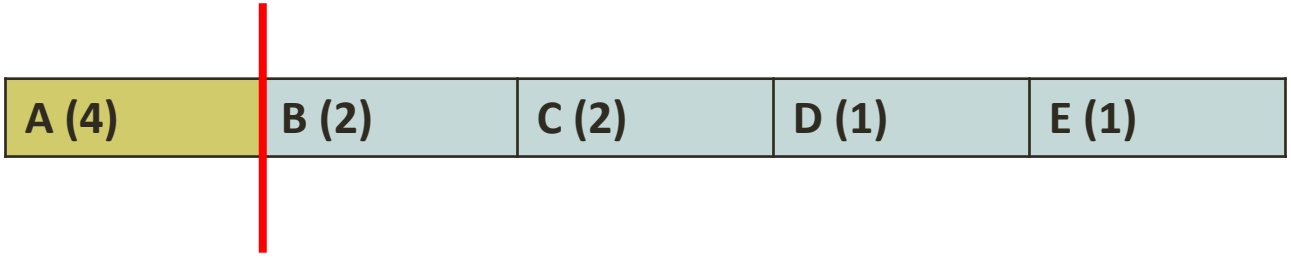
- Draw a Shannon for this distribution.

steps

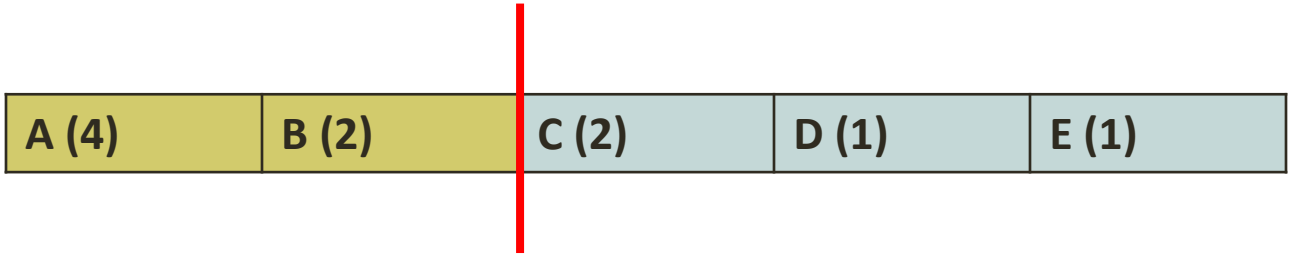
- first you need to **sort** the symbols based on the frequency.
- Second, try to spilt the collection into 2 groups, where the collection with large frequency in the right.
- Third, repeat (1,2) in each sub tree.
- “splitting is after sorting”

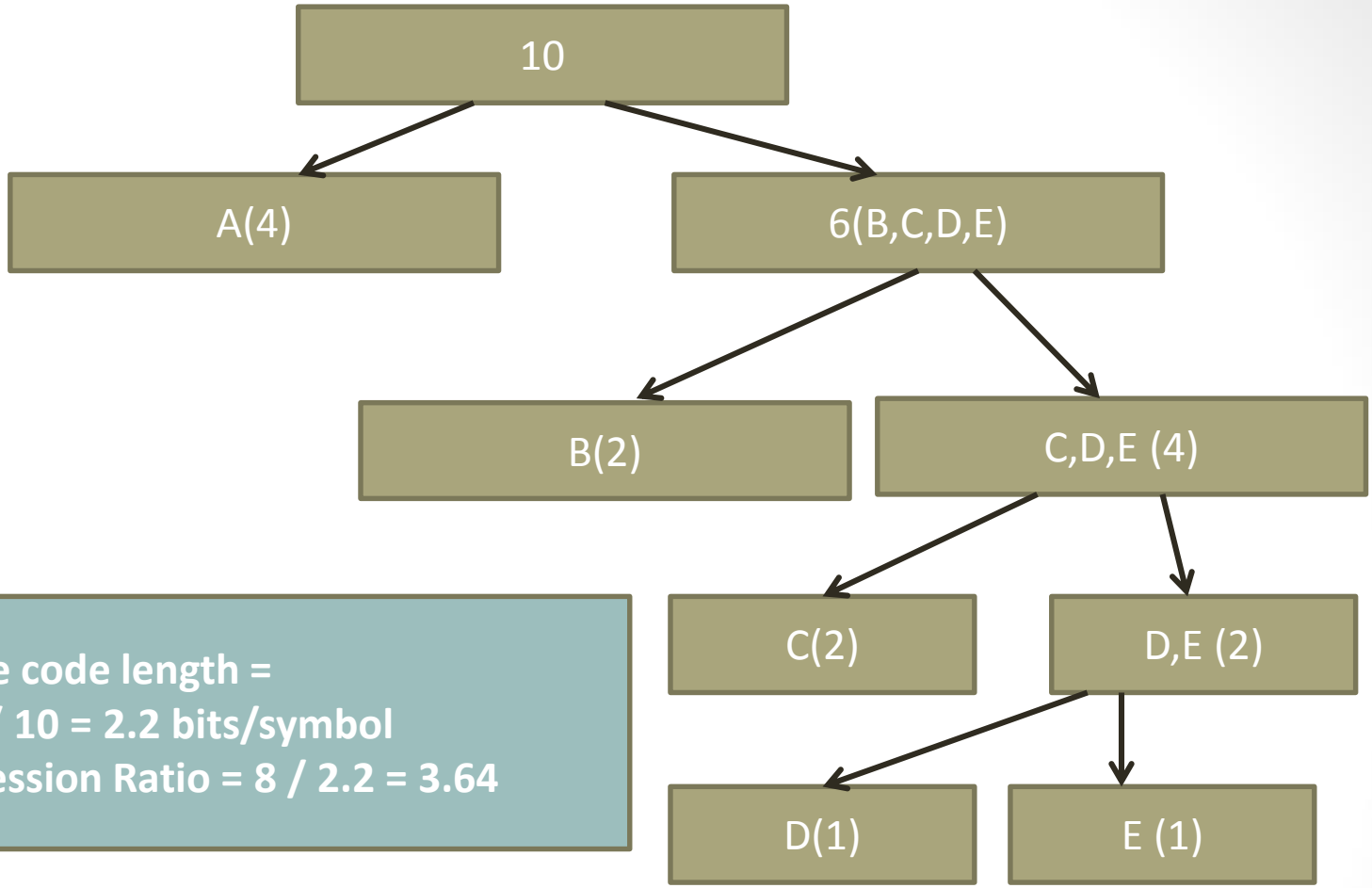
steps

- Only we can create 2 combinations:



OR





- **Average code length =**
  22 / 10 = 2.2 bits/symbol
- **Compression Ratio = 8 / 2.2 = 3.64**

Symbol	Count	pi	Code	Subtotal (#of bits)
A	4	0.4	0	4*1 bit = 4
B	2	0.2	10	2*2 bits = 4
C	2	0.2	110	2*3 bits = 6
D	1	0.1	1110	1*4 bits = 4
E	1	0.1	1111	1*4 bits = 4
Totals	10	1		22 bits

Variable-length coding

2. Huffman Coding Algorithm

Huffman Coding Algorithm— a ***bottom-up*** approach

1. **Initialization**: Put all symbols on a list sorted according to their frequency counts.
2. **Repeat until the list has only one symbol left:**
   a. From the list pick two symbols with the lowest frequency counts. Form a Huffman subtree that has these two symbols as child nodes and create a parent node.
   b. Assign the sum of the children's frequency counts to the parent and insert it into the list such that the order is maintained.
   c. Delete the children from the list.
   d. Assign a codeword for each leaf based on the path from the root.

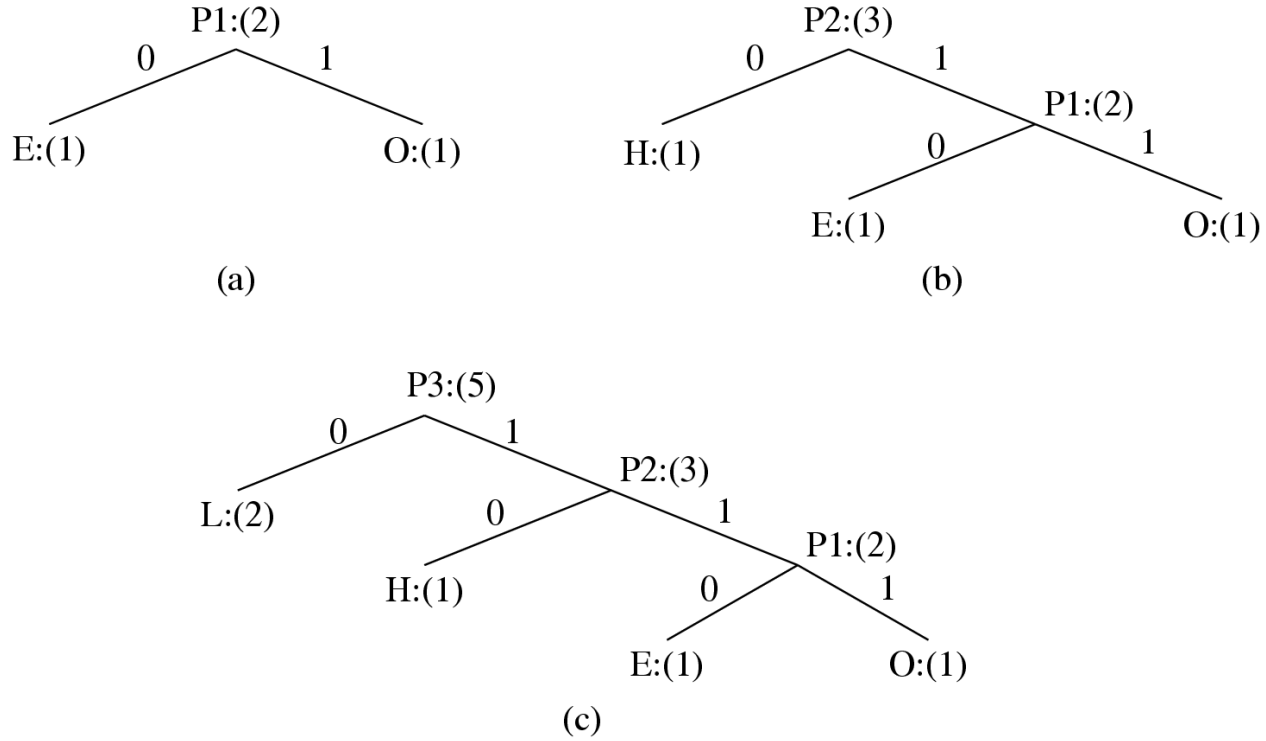


Fig. 7.5: Coding Tree for "HELLO" using the Huffman Algorithm.

In the Figure, new symbols P1, P2, P3 are created to refer to the parent nodes in the Huffman coding tree. The contents in the list are illustrated below:

After initialization:	L	H	E	O
After iteration (a):	L	P1	H	
After iteration (b):	L	P2		
After iteration (c):		P3		

Properties of Huffman Coding

- **Unique Prefix Property**: No Huffman code is a prefix of any other Huffman code - precludes any ambiguity in decoding.

- **Optimality**: shortest tree → *minimum redundancy code* - proved optimal for a given data model (i.e., a given, accurate, probability distribution):
  - The two least frequent symbols will have the same length for their Huffman codes, differing only at the last bit.
  - Symbols that occur more frequently will have shorter Huffman codes than symbols that occur less frequently.

Shannon-Fano VS Huffman Encoding - Tutorial

Huffman Example (1)

A. What is the **entropy** () of the image below, where numbers (0, 20, 50, 99) denote the gray-level intensities?
B. Show step by step how to construct the Huffman tree to encode the above four intensity values in this image. Show the resulting code for each intensity value.
C. What is the **average number of bits** needed for each pixel, using your Huffman code? How does it compare to ?

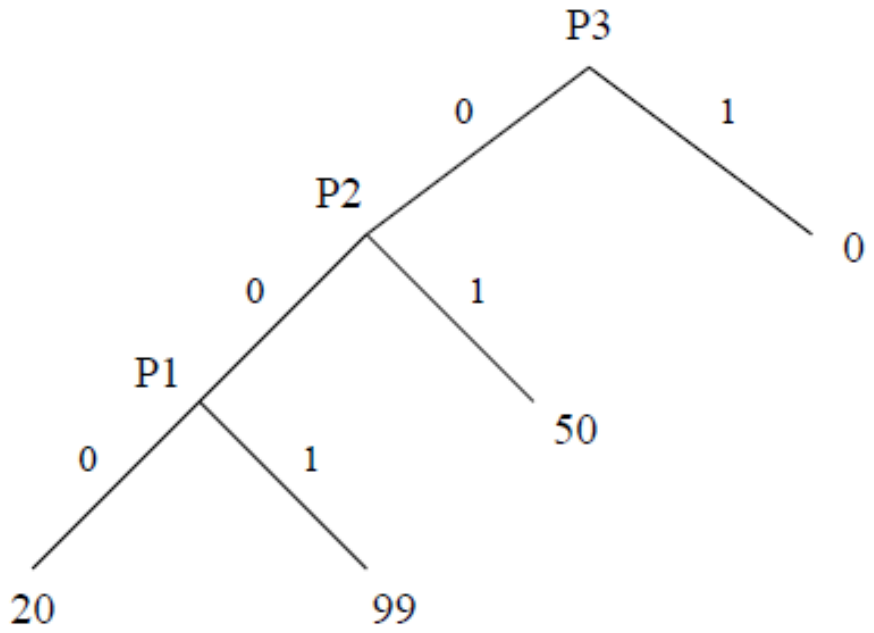
99	99	99	99	99	99	99	99
20	20	20	20	20	20	20	20
0	0	0	0	0	0	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	50	50	50	50	0	0
0	0	0	0	0	0	0	0

Answer:

(a) $P_{20} = P_{99} = 1/8$, $P_{50} = 1/4$, $P_0 = 1/2$.

$$\eta = 2 \times \frac{1}{8}\log_2 8 + \frac{1}{4}\log_2 4 + \frac{1}{2}\log_2 2 = \frac{3}{4} + \frac{1}{2} + \frac{1}{2} = 1.75$$

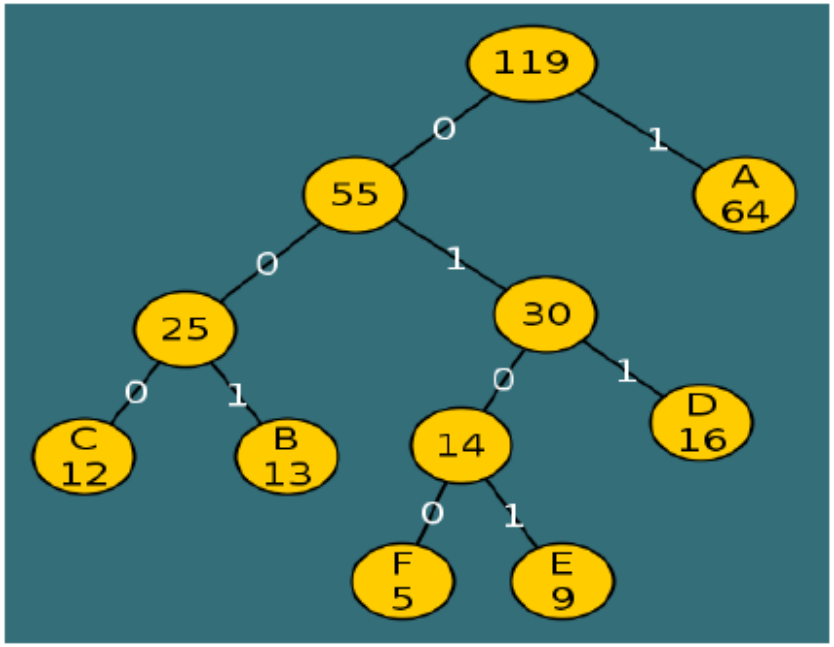
(b) Only the final tree is shown below. Resulting code: 0: "1", 50: "01", 20: "000", 99: "001"



(c) Average number of bits $= 0.5 \times 1 + 0.25 \times 2 + 2 \times 0.125 \times 3 = 1.75$.

This happens to be identical to η — it only happens when all probabilities are 2^{-k} where k is an integer. Otherwise, this number will be larger than η.

Huffman Example (2)



Given The following table from counting characters frequencies from a string

A	B	C	D	E	F
64	13	12	16	9	5

Probability Table

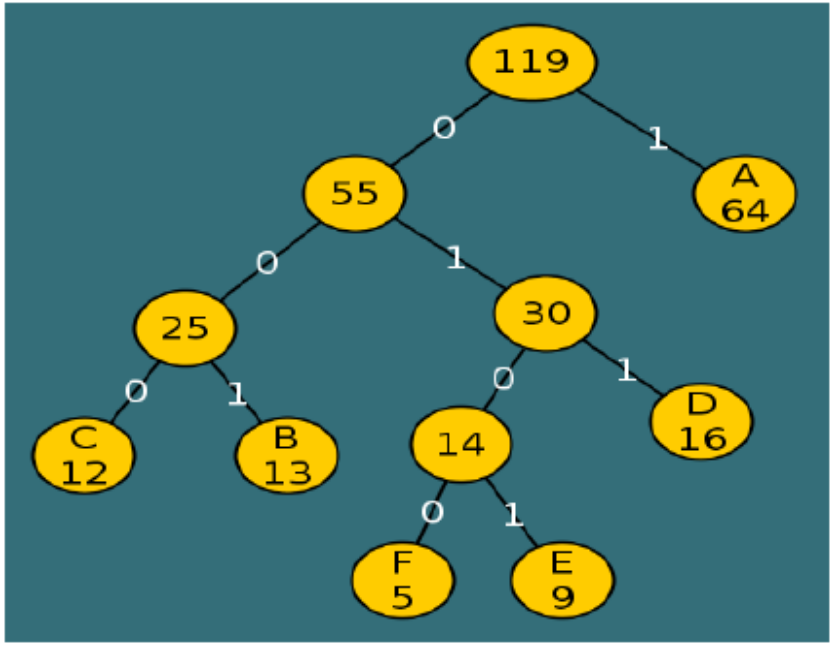
A	B	C	D	E	F	Total
64	13	12	16	9	5	119
0.538	0.1092	0.1008	0.1345	0.0756	0.042	1.0

Code Table

A	B	C	D	E	F
1	001	000	011	0101	0100

Huffman Example (2)

- A(64), D(16), B(13), C(12), E(9), F(5)
- A(64), D(16), P1(14), B(13), C(12)
- A(64), P2(25), D(16), P1(14)
- A(64), P3(30), P2(25)
- A(64), P4(55)
- P5(119)



Huffman Example (2)

- **Average code length =** $\sum_{i=0}^{n} P_i L_i$

where P is the probability of the symbol and L is the code length of the symbol

- **avg** = 0.538 * 1 + 0.1092 * 3 + 0.1008 * 3 + 0.1345 * 3 + 0.0756 * 4 + 0.042 * 4 = **2.042**

- ~~**Entropy** = Average code length (bits/symbol)~~

~~**Entropy = 2.042 bits/symbol**~~

- **Compression Ratio** = Original Symbol Length before Compression / Average Code Length after compression
- **Compression Ratio = 8 / 2.042 = 3.918 ≈ 4**
- 8 because we are using ASCII which stores each symbol in 1 byte , which is 8 bits

Decompression

Decompression is straight forward using the tree given :

- “100001010101011” →
- **1**00001010101011
- A
- **1 000** 01010101011
- A C
- **1 000 0101** 0101011
- A C E
- **1 000 0101 0101** 011
- A C E E
- **1 000 0101 0101 011**
- A C E E D

→ 100001010101011 = **ACEED**