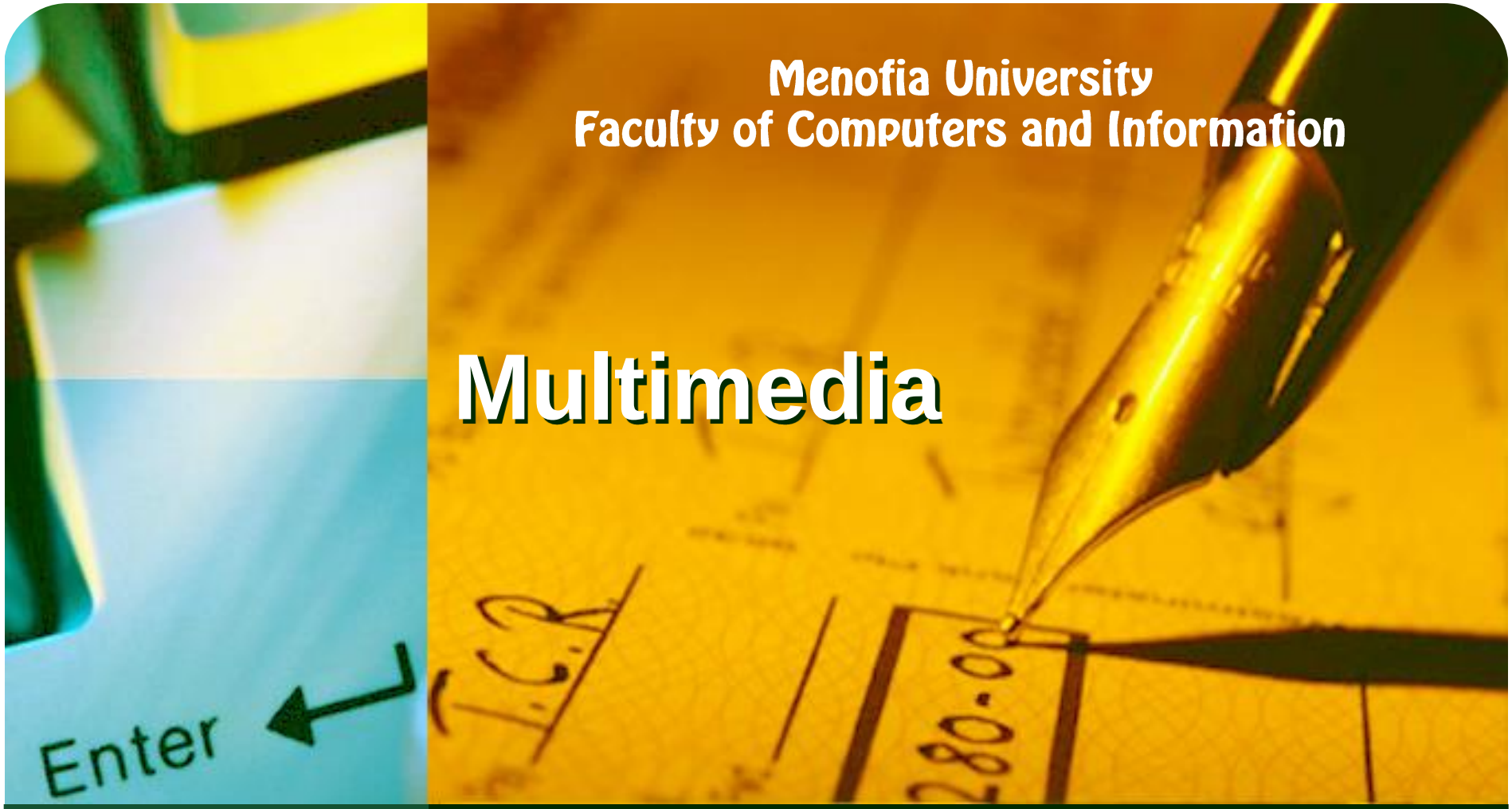


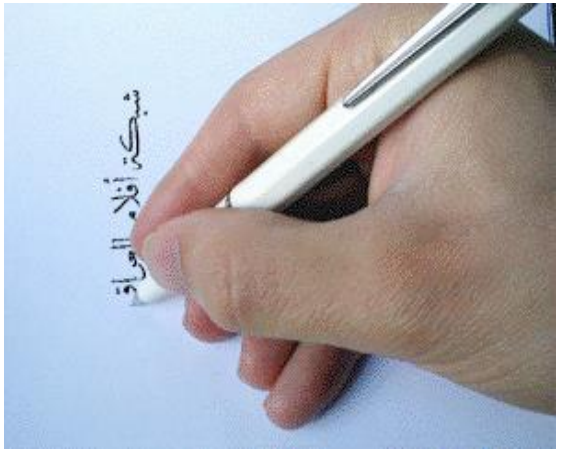
Menofia University
Faculty of Computers and Information

Multimedia

Dr. Sameh Zarif

Lecture #9

General Rules



Chapter 6

BASIC COMPRESSION ALGORITHMS

Need for Compression

Raw video, image, and audio files can be very large.

Example: One minute of uncompressed audio.

Audio Type	*44.1 KHz*	*22.05 KHz*	*11.025 KHz*
16 Bit Stereo:	10.1 MB	5.05 MB	2.52 MB
16 Bit Mono:	5.05 MB	2.52 MB	1.26 MB
8 Bit Mono:	2.52 MB	1.26 MB	630 KB

Example: Uncompressed images.

Image Type	*File Size*
512 x 512 Monochrome	0.25 MB
512 x 512 8-bit colour image	0.25 MB
512 x 512 24-bit colour image	0.75 MB

Need for Compression

Example: Videos (involves a stream of audio plus video imagery).

- Raw Video — uncompressed image frames 512x512 True Colour at 25 FPS = 1125 MB/min.
- HDTV (1920×1080) — Gigabytes per minute uncompressed, True Colour at 25 FPS = 8.7 GB/min.

- Compression HAS TO BE part of the representation of audio, image, and video formats.

Data Compression

- **Data compression:** is the science of representing data in a compact form.

- A compression problem involves finding an efficient algorithm to remove various redundancy from a certain type of data.

Many questions related to data compression

Why do we compress?

What do we mean by source data?

How would we know if there is any redundancy in a source?

Why do we Compress

- **Conserve storage space**
- **Reduce time for transmission**
- **Progressive transmission**
  - Some compression techniques allow us to send the most important bits first so we can get a low resolution version of some data before getting the high version.
- **Reduce computation**
  - Use less data to achieve an approximate answer

Compression in Multimedia Data

Compression basically employs redundancy in the data:

Temporal in 1D data, 1D signals, audio, between video frames etc.

Spatial correlation between neighbouring pixels or data items.

Spectral *e.g.* correlation between colour or luminescence components. This uses the frequency domain to exploit relationships between frequency of change in data.

Psycho-visual exploit perceptual properties of the human visual system.

Lossless vs Lossy Compression

Compression can be categorised in two broad ways:

Lossless Compression: after decompression gives an exact copy of the original data.

Example: Entropy encoding schemes (Shannon-Fano, Huffman coding), arithmetic coding, LZW algorithm (used in GIF image file format).

Lossy Compression: after decompression gives ideally a "close" approximation of the original data, ideally perceptually lossless.

Example: Transform coding — FFT/DCT based quantisation used in JPEG/MPEG differential encoding, vector quantisation.

Why Lossy Compression?

- Lossy methods are typically applied to high resoultion audio, image compression.
- Have to be employed in video compression (apart from special cases).

Basic reason:

- Compression ratio of lossless methods (*e.g.* Huffman coding, arithmetic coding, LZW) is not high enough for audio/video.
- By cleverly making a small sacrifice in terms of fidelity of data, we can often achieve very high compression ratios.
  - Cleverly = sacrifice information that is psycho-physically unimportant.

Lossless Compression Algorithms

- Repetitive Sequence Suppression.
- Run-Length Encoding (RLE).
- Pattern Substitution.
- Entropy Encoding:
  - Shannon-Fano Algorithm.
  - Huffman Coding.
  - Arithmetic Coding.
- Lempel-Ziv-Welch (LZW) Algorithm.

Simple Repetition Suppression

If a sequence a series on n successive tokens appears:

- Replace series with a token and a count number of occurrences.
- Usually need to have a special flag to denote when the repeated token appears.

Example:

894000000000000000000000000000000

we can replace with:

894f32

where `f` is the flag for zero.

Simple Repetition Suppression

- Fairly straight forward to understand and implement.
- Simplicity is its downfall: poor compression ratios.

Compression savings depend on the content of the data.

Applications of this simple compression technique include:

- Suppression of zeros in a file (Zero Length Suppression)
  - Silence in audio data, pauses in conversation etc.
  - Sparse matrices.
  - Component of JPEG.
  - Bitmaps, *e.g.* backgrounds in simple images.
  - Blanks in text or program source files.

Run-length Encoding (RLE)

This encoding method is frequently applied to graphics-type images (or pixels in a scan line) — simple compression algorithm in its own right.
It is also a component used in JPEG compression pipeline.

Basic RLE Approach (*e.g.* for images):

- Sequences of image elements $X_1, X_2, \ldots, X_n$ (row by row).
- Mapped to pairs $(c_1, L_1), (c_2, L_2), \ldots, (c_n, L_n)$, where c_i represent image intensity or colour and L_i the length of the i-th run of pixels.

Run-length Encoding Example

Original sequence:
11112223333311112222

can be encoded as:
(1,4),(2,3),(3,6),(1,4),(2,4)

How Much Compression?

The savings are dependent on the data: In the worst case (random noise) encoding is more heavy than original file: 2×integer rather than 1×integer if original data is integer vector/array.

Run-length Encoding Example

b) ***Binary image***

Original line (hex): 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1

Code (hex) 0 6 3 3 2 1 5 8

Pattern Substitution

- This is a simple form of statistical encoding.
- Here we substitute a frequently repeating pattern(s) with a code.
- The code is shorter than the pattern giving us compression.

The simplest scheme could employ predefined codes:

Example: Basic Pattern Substitution

Replace all occurrences of pattern of characters 'and' with the predefined code '&'. So:

```
and you and I
```

becomes:

```
& you & I
```

Code Assignment

- A predefined codebook may be used, i.e. assign code c_i to symbol s_i. (*E.g.* some dictionary of common words/tokens).
- Better: dynamically determine best codes from data.
- The entropy encoding schemes (next topic) basically attempt to decide the optimum assignment of codes to achieve the best compression.

Example:

- Count occurrence of tokens (to estimate probabilities).
- Assign shorter codes to more probable symbols and vice versa.

Fixed-length vs. Variable length codes

Problem: Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Fixed-length vs. Variable length codes

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,

(1) the fixed-length code requires $100 \times 2 = 200$ bits,

(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

a 25% saving.

The Shannon-Fano Algorithm

Compression: Given a list of symbols, the algorithm involves the following steps:

1. Develop a frequency (or probability) list.

2. Sort the list according to frequency (the most frequent one at the left).

3. Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.

4. Assign the left half of the list a 0 and the right half a 1.

5. Recursively apply the steps 3 and 4 to the two halves, subdividing groups and adding bits to the code words until each symbol has become a corresponding leaf on the tree.

The Shannon-Fano Algorithm

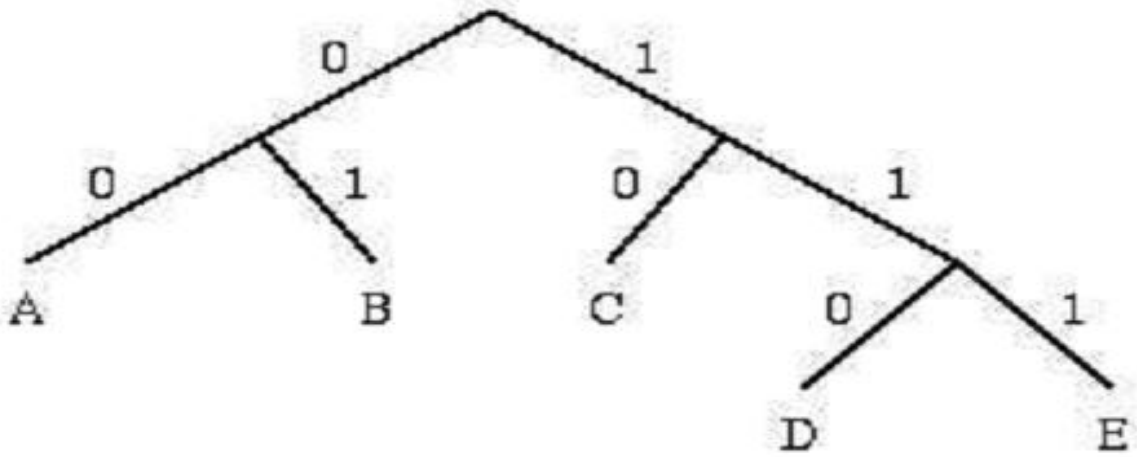
Example:

Consider a finite symbol stream:

ACABADADEAABBAAAEDCACDEAAABCDBBEDCBACAE

Count symbols in stream:

Symbol	A	B	C	D	E
Count	15	7	6	6	5



The Shannon-Fano Algorithm

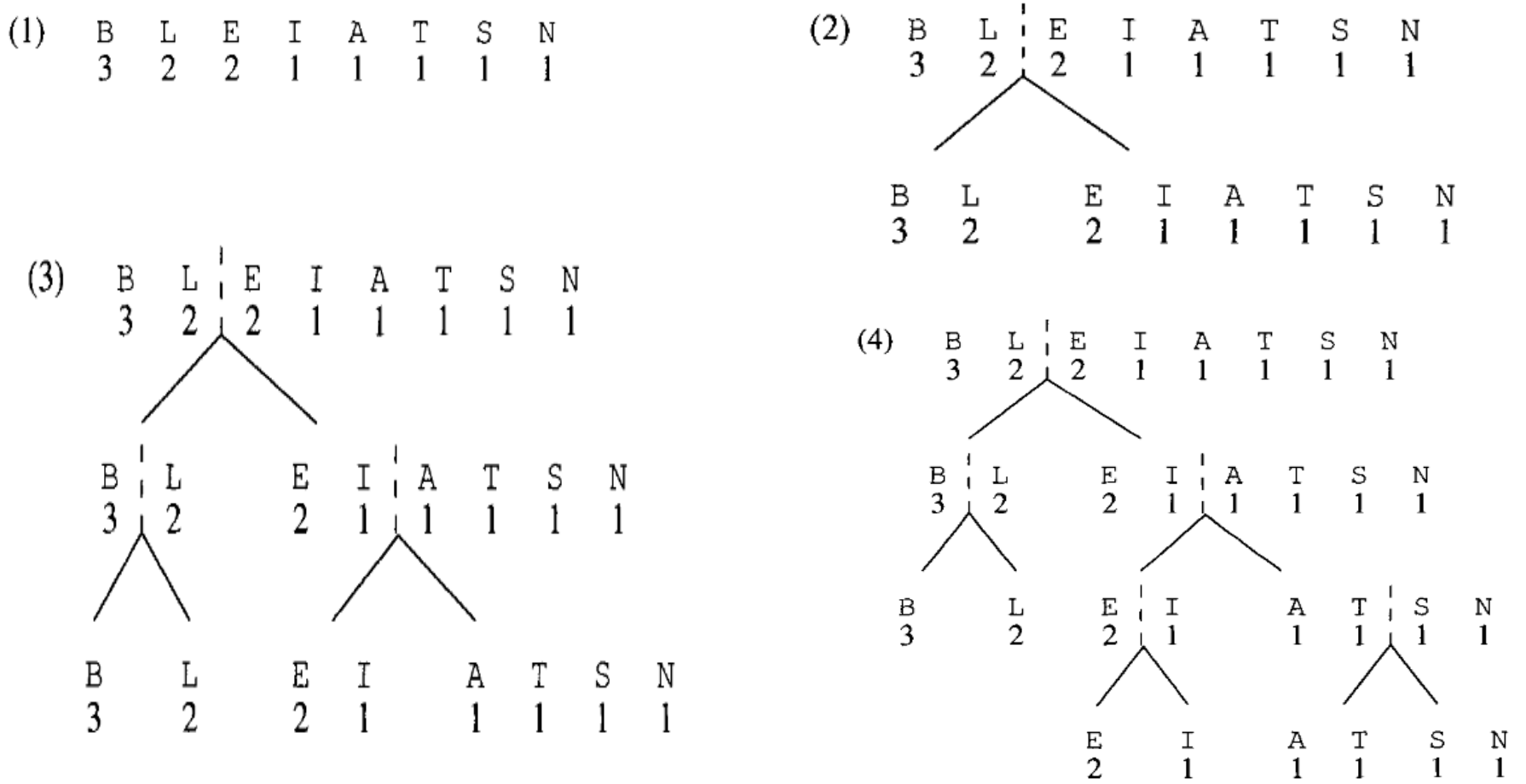
3. Assemble codebook by depth first traversal of the tree:

Symbol	Count	log(1/p)	Code	# of bits
A	15	1.38	00	30
B	7	2.48	01	14
C	6	2.70	10	12
D	6	2.70	110	18
E	5	2.96	111	15
			TOTAL (# of bits):	89

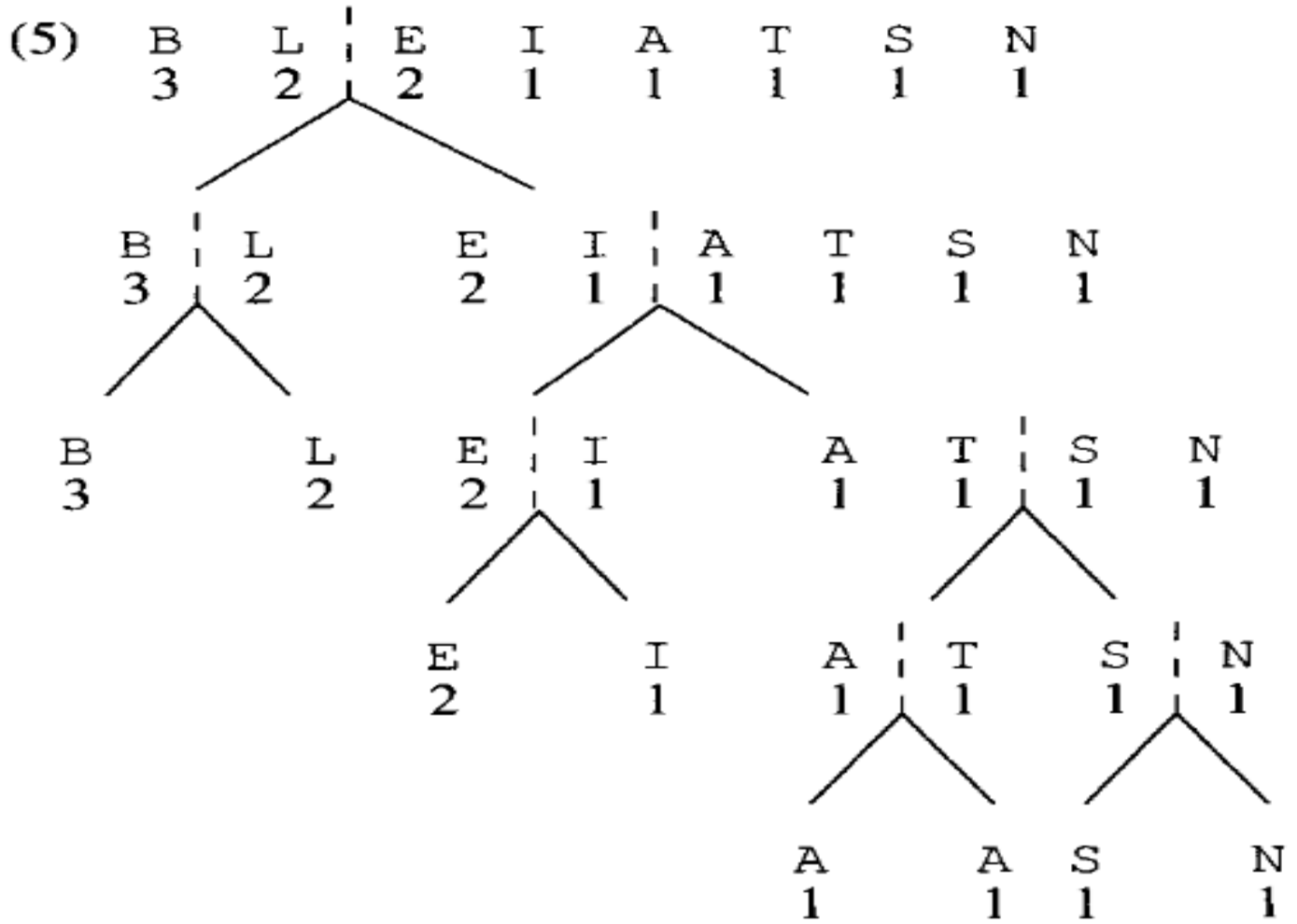
4. Transmit codes instead of tokens.
   In this case:
   - Raw token stream 8 bits per (39 chars) = 312 bits.
   - Coded data stream = 89 bits.

The Shannon-Fano Algorithm

Example: applying the approach to the previous example



The Shannon-Fano Algorithm



Huffman Algorithm

Can we do better than Shannon-Fano?

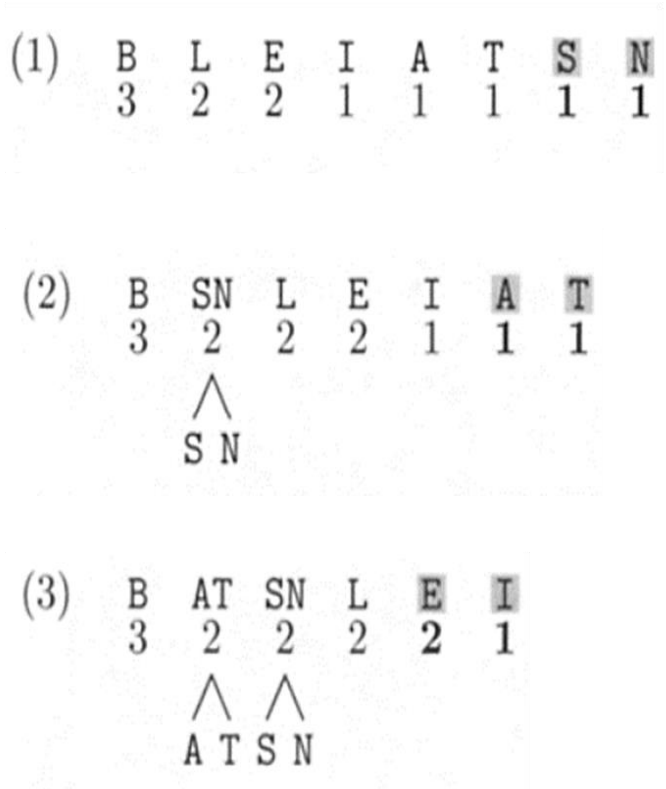
Huffman! Always produces best binary tree for given probabilities.

A bottom-up approach:

1. Initialization: put all nodes in a list L, keep it sorted at all times (e.g., ABCDE).
2. Repeat until the list L has more than one node left:
   - From L pick two nodes having the lowest frequencies/probabilities, create a parent node of them.
   - Assign the sum of the children's frequencies/probabilities to the parent node and insert it into L.
   - Assign code 0/1 to the two branches of the tree, and delete the children from L.
3. Coding of each node is a top-down label of branch labels.

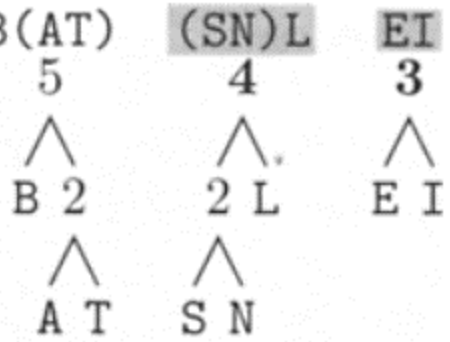
Huffman Algorithm Example

Example: consider the alphabet of (B, I, L, E, A, T, S, N) with associated frequencies of (3,1,2,2,1,1,1,1), the binary tree of the prefix code is constructed as follow

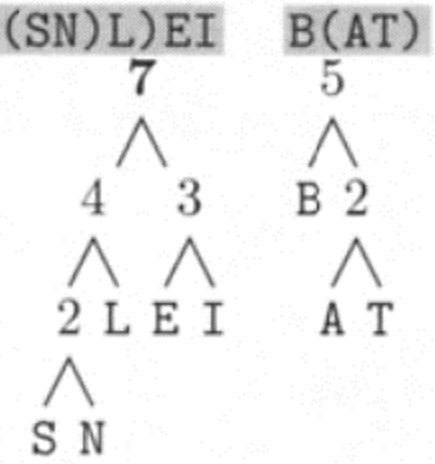


Huffman Algorithm Example

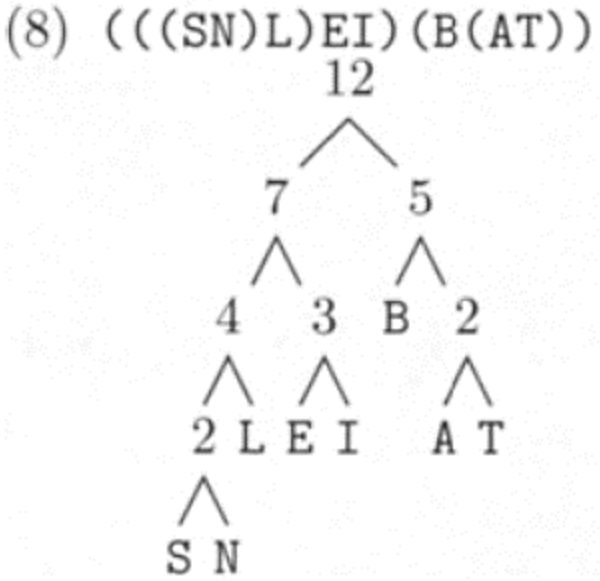
(6)



(7)

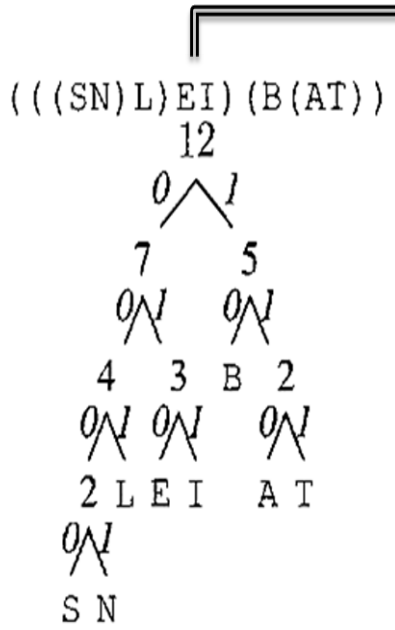


(8)



Huffman Algorithm Example

Constructing the prefix code: after constructing the binary tree, we assign 1 to the right branches in the tree and zero to the left branches. Then we trace the code from the root to the leaves. Like



The prefix code **(10 001 010 011 110 111 0000 0001)** for the whole alphabet **(B, L, E, I, A, T, S, N)** respectively.

Huffman Algorithm Discussion

The following points are worth noting about the above algorithm:

- Decoding for the above two algorithms is trivial as long as the coding table/book is sent before the data.
  - There is a bit of an overhead for sending this.
  - But negligible if the data file is big.


Enter
Thank You!