Multimedia Communication (SW-416)

BASICS OF COMPRESSION

Dr. Areej Fatemah

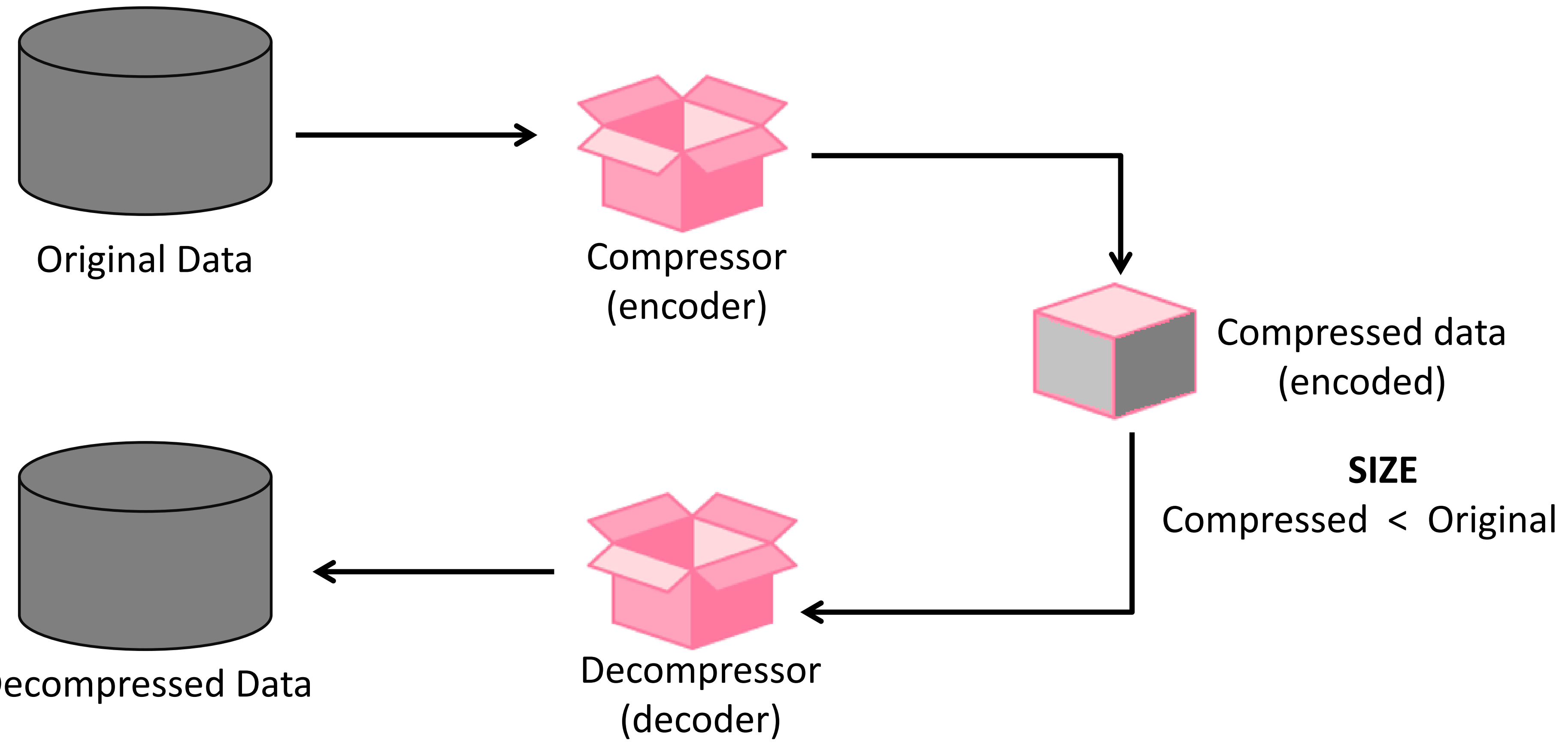
email: areej.fatemah@faculty.muet.edu.pk

Data Compression

- Data compression is the process of encoding, restructuring or otherwise modifying data in order to reduce its size. Fundamentally, it involves re-encoding information using fewer bits than the original representation.
  - Data compression is achieved through use of specific encoding schemes.

- Compression is useful because it helps reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth.
- On the downside, compressed data must be decompressed to be used, and this extra processing may be detrimental to some applications.

Data Compression



Data Compression

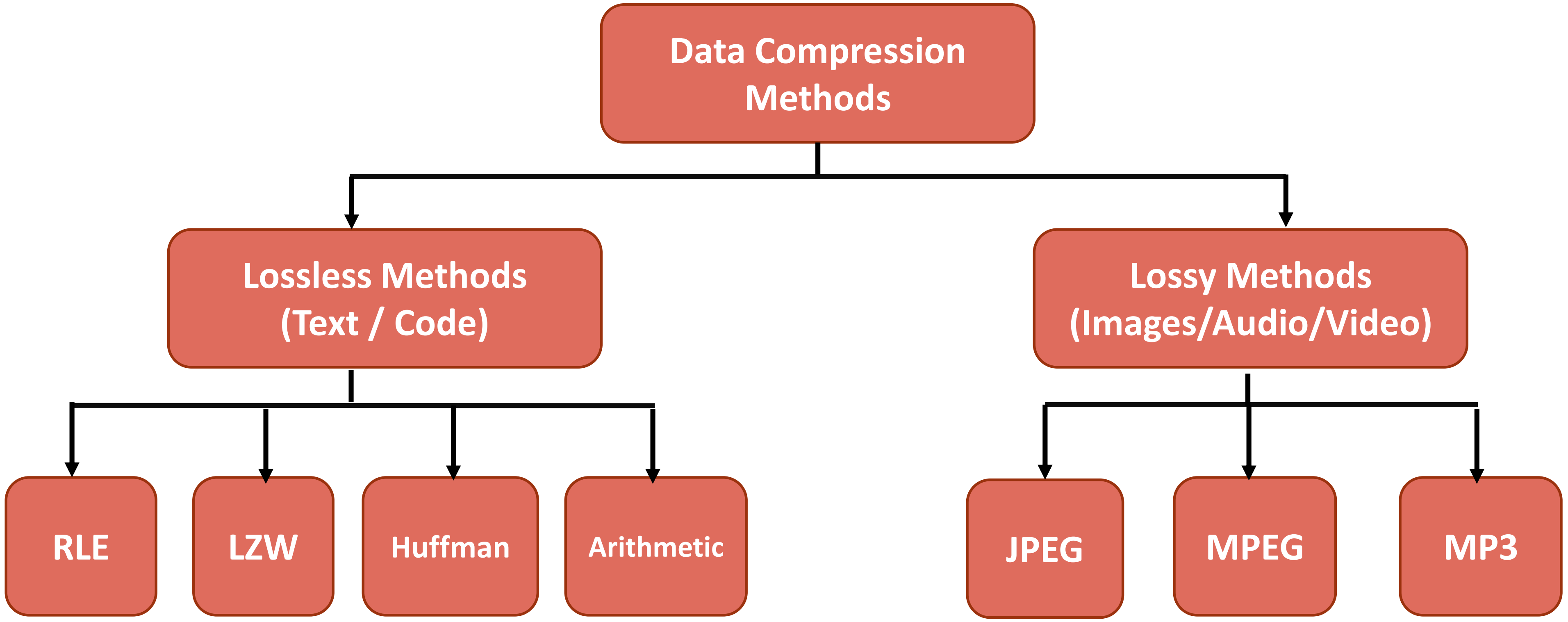
- As the Internet emerged in the 1970s, the relationship between file size and transfer speed became much more apparent.
- Mathematicians around the world addressed the problem for years, but it wasn’t until the *Lempel-Ziv-Welch (LZW)* universal lossless compression algorithms emerged in the mid-1980s that real benefits were realized.
- LZW compression was the first widely used data compression method implemented on computers and it is still used today
  - Using LZW, a large English text file can typically be compressed to about half its original size.
- Common data compression algorithms include:
  - Zip
  - jpeg
  - mpeg

Data Compression Standards

- Standardization has been an essential requirement for any technology that is supported by a large number of manufacturers. Few of the major compression issuing bodies are:

- **CCITT** (International Consulting Committee In Telephony & Telegraphy)
- **CCIR** (International Consulting Committee For Radio)
- **ISO** (International Standards Organisation)

Data Compression
(Types/Methods)


Data Compression Methods
Lossless Methods (Text / Code)
Lossy Methods (Images/Audio/Video)
RLE
LZW
Huffman
Arithmetic
JPEG
MPEG
MP3

Data Compression
(Types/Methods)

- Lossless and Lossy compression are terms that describe whether or not, in the compression of a file, all original data can be recovered when the file is uncompressed.

- With lossless compression, every single bit of data that was originally in the file remains after the file is uncompressed. All of the information is completely restored.
  - This is generally the technique of choice for text or spreadsheet files, where losing words or financial data could pose a problem.
  - The Graphics Interchange File (GIF) is an image format used on the Web that provides lossless compression.

Data Compression
(Types/Methods)

- There are times when a picture may have more detail than the eye can distinguish or an audio file has immensely high quality that might not be noticed by the audience it is intended for; this is where Lossy Compression comes into play.

- This technique results in some loss of information and is used where data accuracy is not essential. Usually for audio, image and video.
  - JPEG
  - MP3
  - MPEG

Data Compression
(Types/Methods)

1. **Lossless**
   - No loss of information or quality
   - Can recreate the original data from the compressed data
   - e.g. GIF, TIFF, BMP, PNG, H.246

2. **Lossy**
   - Some loss of information or quality
   - Cannot recreate the original data from the compressed data
   - e.g. JPEG, MP3, MP4

Data Compression
Compression Groups

- **Group 2**

  Very early (1980s) compression scheme developed for facsimile machines featuring resolutions as high as 100 dpi. It did not provide a very high level of compression and is generally not in use anymore.

- **Group 3 1D**

  Also known as run-length encoding (RLE), its based on the assumption that a typical scan line has long *runs* of pixels of the same color (black or white). Primary applications has been facsimile and very early document imaging systems.

Data Compression
Compression Groups

- **Group 3 2D**

  Also known as modified run-length encoding. It is more commonly used for software-based document imaging systems. While it provides fairly good compression, it is easier to decompress as well. The compression ratio for this scheme averages between 10 and 25. It utilizes a modified READ (Relative Element Address Designated) algorithm.

- **Group 4**

  Two dimensional coding scheme. In this method, the first reference line is an imaginary all-white line above the top of the image. It was designed to address high-resolution images in black-n-white.

Data Compression
Compression Groups

- **Group 5**

  Designed to address the need for an efficient content-based encoding methodology that also addresses the color and shade information.

Text Compression

Run Length Encoding

- Suited for compressing any type of data regardless of its information, but content of data will affect the compression ratio.
  - Low compression ratios
  - Easy to implement
  - Quick to execute
- It works by reducing the size of repeating string of characters.
- A repeating string called *Run* is typically encoded into two bytes.
  - The first byte represents the number of characters in the run and is called *Run Count*. In practice, an encoded run may contain 1 to 128 or 256 characters.
  - The second byte is the value of the character and is called Run Value (range of 0-255).
- Works best when data contains long runs of the same value.

Types of Compression

- **Run Length Encoding**

- **Uncompressed:**
- aaaaaaaaaaaaaaa

Bytes?

- a ASCII = 097 $\longrightarrow$ Binary: $\overbrace{01100001}^{\text{8 bits = 1 byte}}$
- **Uncompressed - 15 bytes**

- **Compressed:**
- 15a

- Bytes:
- **Compressed – 2 bytes**

Types of Compression

- **Run Length Encoding**

- **Uncompressed:**
- aaaaabbbbbbccccddddd

- **Bytes:**
- **20 bytes**

- **Compressed:**
- 5a6b4c5d

- Bytes:
- **8 bytes**

Types of Compression

- **Run Length Encoding**

Negative Compression

- **Uncompressed:**
- aabccdeefghijjklmnooopqrstttttuuvwxyz

37 Bytes

- **Compressed:**
- 2a1b2c1d2e1f1g1h1i2j1k1l1m1n3o1p1q1r1s5t2u1v1w1x1y1z

52 Bytes

Run Length Encoding

- Run Length Encoding (RLE) is most useful on data that contains many runs:
    - for example, relatively repeated characters, simple graphic images such as icons, line drawings, and animations.
- It is not useful with files that don't have many runs as it could potentially double the file size.

Please refer class notes for examples of Run Length Encoding.

How Word Editors Use Compression

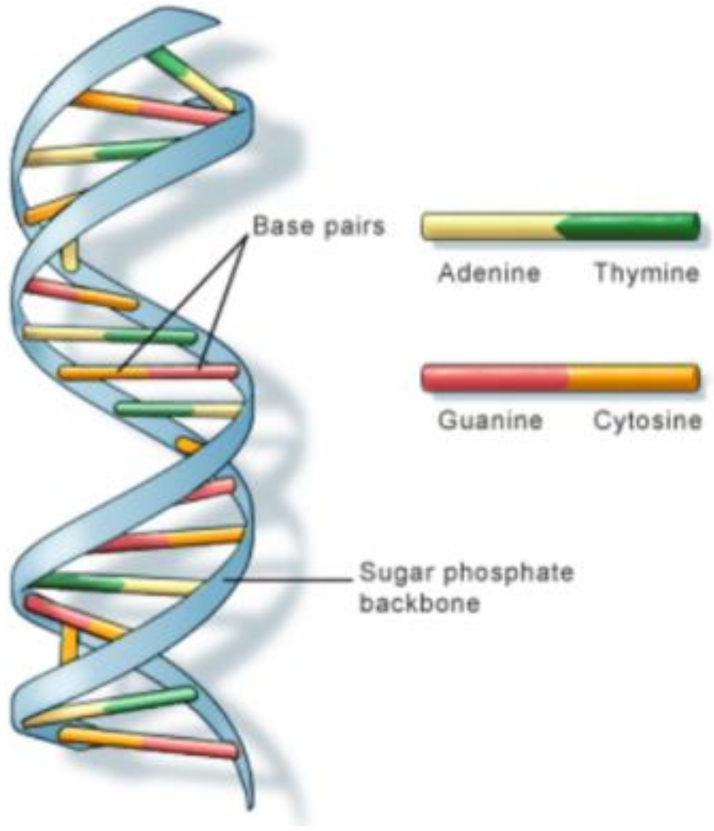
- **DNA Sequence**

	G	C	A	T
	25%	**25%**	**25%**	**25%**
a unique 2-bit combination can be used to represent G, C, A, and T $\longrightarrow$	**00**	**01**	**10**	**11**

Encode the following sequence:

A C T G C T

10 01 11 00 01 11



How Word Editors Use Compression

- **DNA Sequence**

~~GGG~~CA~~GG~~A~~G~~A~~G~~ACC~~G~~T~~G~~CT~~G~~CT~~G~~C~~G~~C~~G~~C~~GG~~CT~~GG~~C~~GG~~A~~G~~T

G	**50%**
C	**25%**
A	**12.5%**
T	**12.5%**

Observing the above sequence, it is obvious that some occurrences are more frequent

How Word Editors Use Compression

- **DNA Sequence**

G	**C**	**A**	**T**
50%	**25%**	**12.5%**	**12.5%**
0	**10**	**110**	**111**

How do you decode?
11010111000111

Instead of a fixed size (2-bit) representation, why not assign variable size bit representations?

The data occurring most frequently can be encoded using fewer bits to save space.

How Word Editors Use Compression

- **Decompression Logic:**

G	C	A	T
0	**10**	**110**	**111**

1 1 0 1 0 1 1 1 0 0 0 1 1 1

How Word Editors Use Compression

- **Decompression Logic:**

~~G~~	C	A	T
0	10	110	111

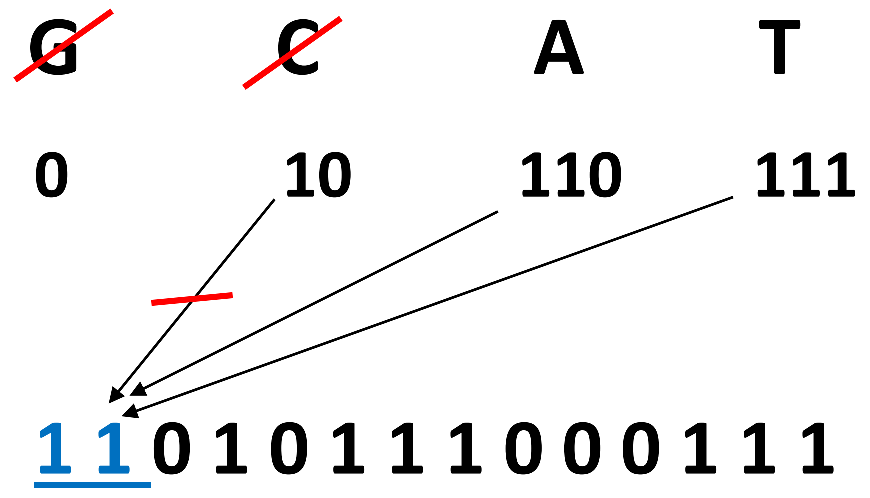
1 1 0 1 0 1 1 1 0 0 0 1 1 1

As the Starting Digit is 1, it cannot be G

It can be either C, A, or T
We need to explore the second digit to be certain

How Word Editors Use Compression

- **Decompression Logic:**



As the second digit is 1, this rules out C which is represented using 10.

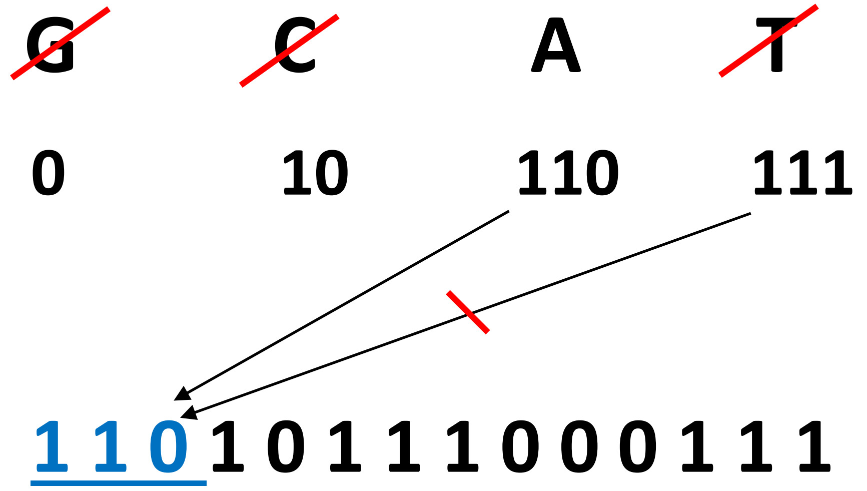
A and T both have the starting sequence of 11.

Now let us observe the third digit to see if the character is A or T.

How Word Editors Use Compression

- **Decompression Logic:**

This third digit is 0 which rules out T



How Word Editors Use Compression

- **Decompression Logic:**

Thus the decoded symbol is A

G	C	A	T
0	**10**	**110**	**111**

$\underline{1\ 1\ 0}$ **1 0 1 1 1 0 0 0 1 1 1**

A

How Word Editors Use Compression

- **Decompression Logic:**

The process will continue till all the bits are examined and the message decoded

~~**G**~~	**C**	**A**	**T**
0	**10**	**110**	**111**

$\underline{\mathbf{1\ 1\ 0\ 1}}\ \mathbf{0\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1}$

A

How Word Editors Use Compression

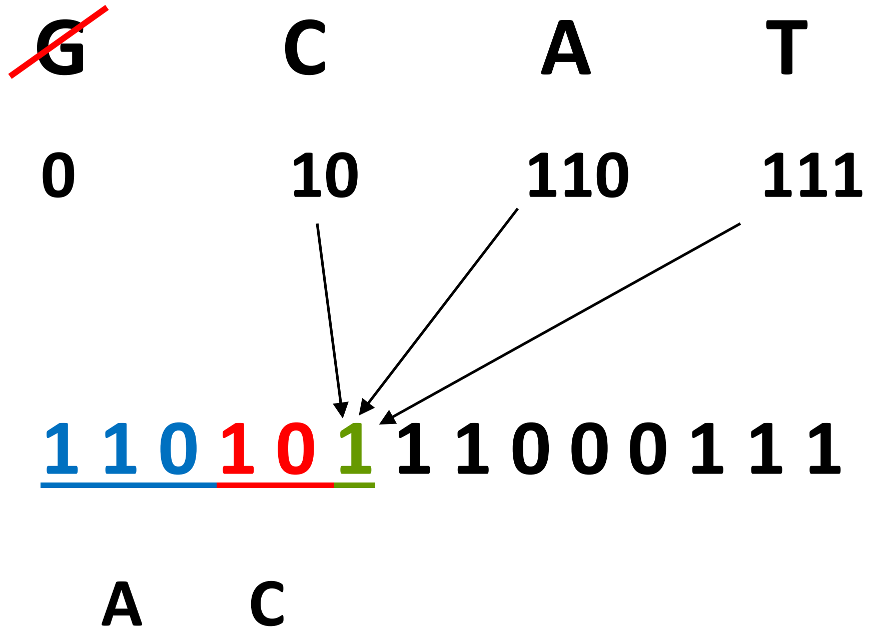
- **DNA Sequence**

$\cancel{\mathbf{G}}$	**C**	$\cancel{\mathbf{A}}$	$\cancel{\mathbf{T}}$
0	**10**	**110**	**111**

$$\underbrace{1\ 1\ 0}_{A}\ \underbrace{1\ 0}_{C}\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1$$

How Word Editors Use Compression

- **DNA Sequence**



How Word Editors Use Compression

- **DNA Sequence**

~~**G**~~	~~**C**~~	**A**	**T**
0	**10**	**110**	**111**

$$\underbrace{1\ 1\ 0}_{A}\ \underbrace{1\ 0}_{C}\ \underline{1\ 1}\ 1\ 0\ 0\ 0\ 1\ 1\ 1$$

How Word Editors Use Compression

- **DNA Sequence**

~~G~~	~~C~~	~~A~~	T
0	10	110	111

$$\underbrace{1\,1\,0}_{A}\,\underbrace{1\,0}_{C}\,\underbrace{1\,1\,1}_{T}\,0\,0\,0\,1\,1\,1$$

How Word Editors Use Compression

- **DNA Sequence**

G	C	A	T
0	10	110	111

$$\underbrace{1\,1\,0}_{A}\,\underbrace{1\,0}_{C}\,\underbrace{1\,1\,1}_{T}\,\underbrace{0}_{G}\,0\,0\,1\,1\,1$$

How Word Editors Use Compression

- **DNA Sequence**

G	C	A	T
0	10	110	111

$$\underbrace{1\ 1\ 0}_{A}\ \underbrace{1\ 0}_{C}\ \underbrace{1\ 1\ 1}_{T}\ \underbrace{0}_{G}\ \underbrace{0}_{G}\ 0\ 1\ 1\ 1$$

How Word Editors Use Compression

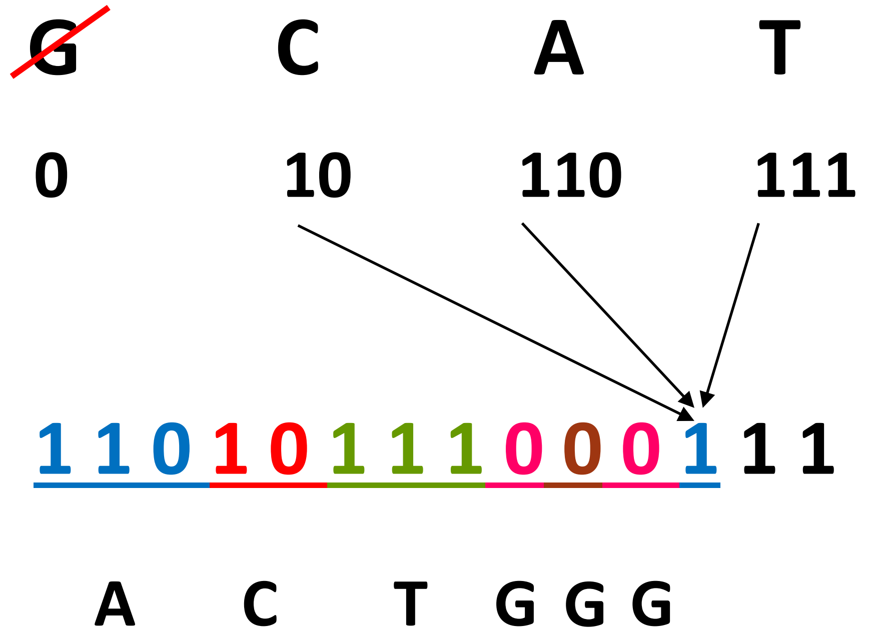
- **DNA Sequence**

G	C	A	T
0	10	110	111

$$\underbrace{1\ 1\ 0}_{A}\ \underbrace{1\ 0}_{C}\ \underbrace{1\ 1\ 1}_{T}\ \underbrace{0}_{G}\ \underbrace{0}_{G}\ \underbrace{0}_{G}\ 1\ 1\ 1$$

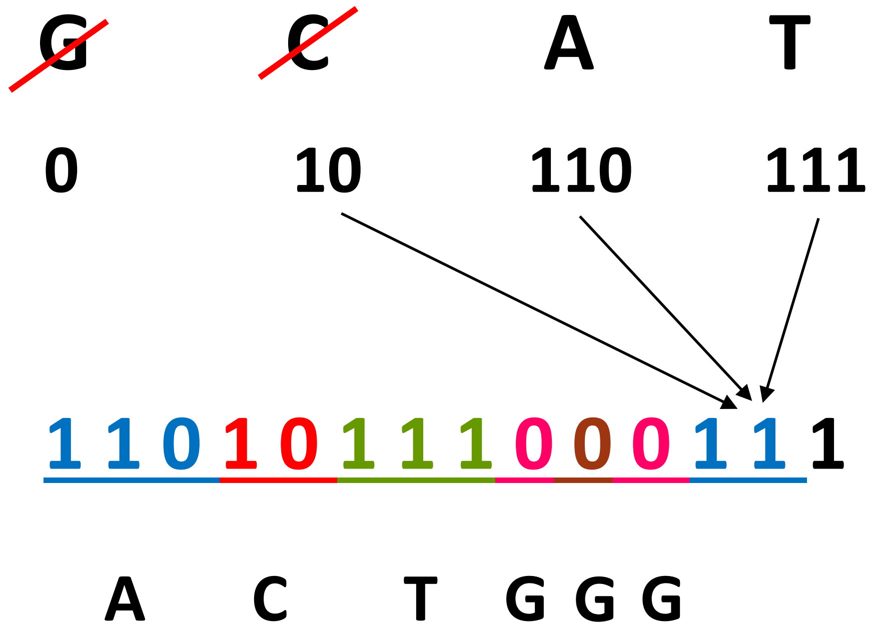
How Word Editors Use Compression

- **DNA Sequence**



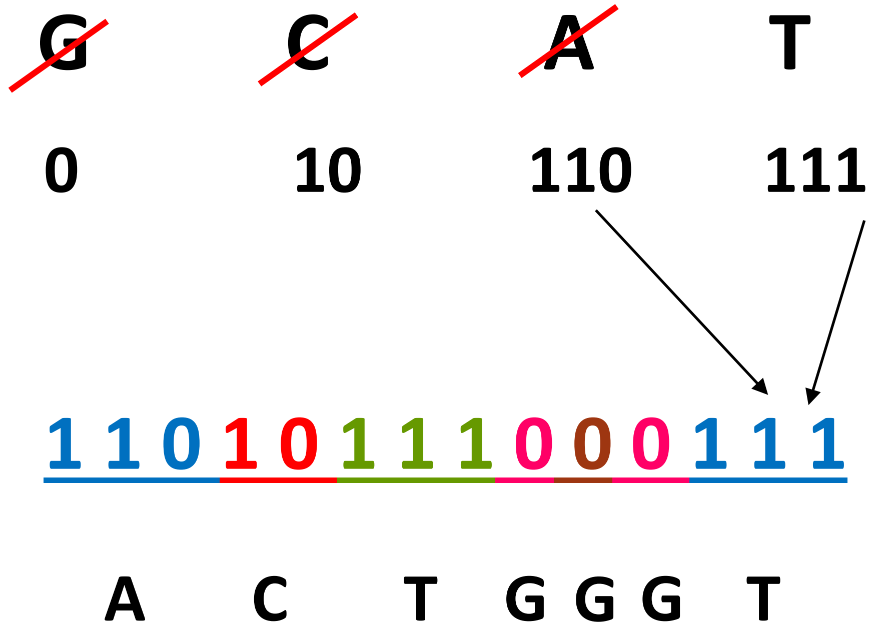
How Word Editors Use Compression

- **DNA Sequence**



How Word Editors Use Compression

- **DNA Sequence**



How Word Editors Use Compression

- **Average Bit Count:**

G	**C**	**A**	**T**
50%	**25%**	**12.5%**	**12.5%**
0	**10**	**110**	**111**

0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3   1.75 bits

- Not all characters are used with the same frequency.

- **Advanced Compression:**
- Uses a dictionary
- Picks out words that commonly occur
  - on, is, if, so, we, to, no, us
  - and, the, but, has, are, for
  - have, this, from, been, some

Huffman Coding

- Huffman coding is a lossless data compression algorithm.
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Huffman Coding

- Huffman encoding algorithm determines the optimal encoding using minimum number of bits.
- Huffman codes have the unique prefix attribute, which means they can be correctly decoded despite being variable length.
- The procedure for building the tree is simple and elegant.
- The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree.
- Each node has a weight, which is simply the frequency or probability of the symbol’s appearance.

Huffman Coding

- The tree is then built with the following steps

1. The two free nodes with the lowest weights are located.
2. A parent node for these two nodes is created.
3. It is assigned a weight equal to the sum of the two child nodes.
4. The parent node is added to the list of free nodes, and the two child nodes are removed from that list.
5. The previous steps are repeated until only one free node is left. This free node is designated the root of the tree

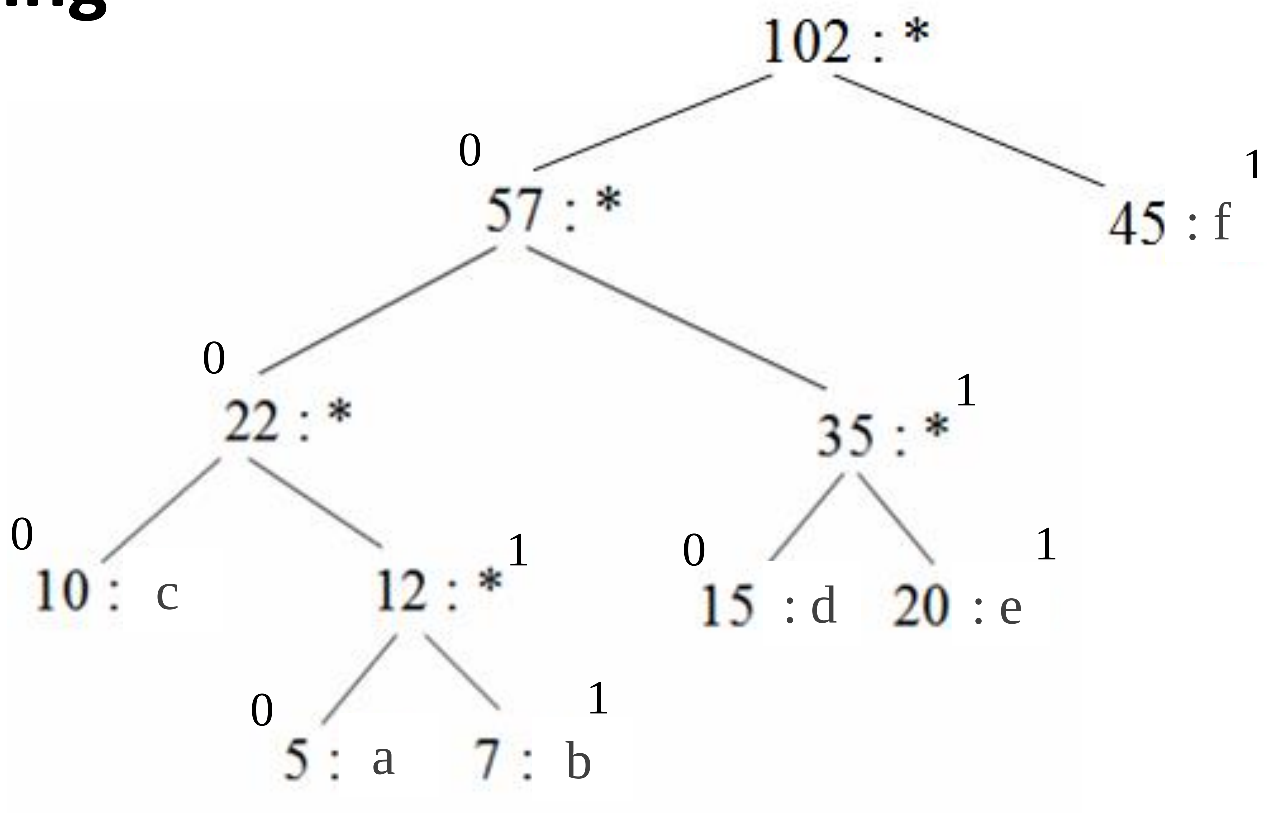
Huffman Coding

- To generate a Huffman code you traverse the tree to the value you want, outputting a **0** every time you take a left-hand branch, and a **1** every time you take a right-hand branch.
- Lets say you have a set of characters and their frequency of use and want to create a Huffman encoding for them:

FREQUENCY	VALUE
5	a
7	b
10	c
15	d
20	e
45	f

Please refer the class notes for the step-by-step solution to this problem and other Huffman coding examples.

Huffman Coding



Thus value for 15 : d becomes - 010

And the value for 5 : a becomes - 0010 and so on

Arithmetic Coding

- It bypasses the idea of replacing an input symbol with a specific code. It works by replacing a stream of input symbols with a single floating-point output number.
- The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0.
- This single number can be uniquely decoded to create the exact stream of symbols that went into its construction.
- To construct the output number, the symbols are assigned a set of probabilities.
- The message “BILL GATES,” for example, would have a probability distribution like this:

Arithmetic Coding

- Once character probabilities are known, individual symbols need to be assigned a range along a “probability line,” nominally 0 to 1.

- The probabilities of the nine-character symbol set used here can be calculated as:

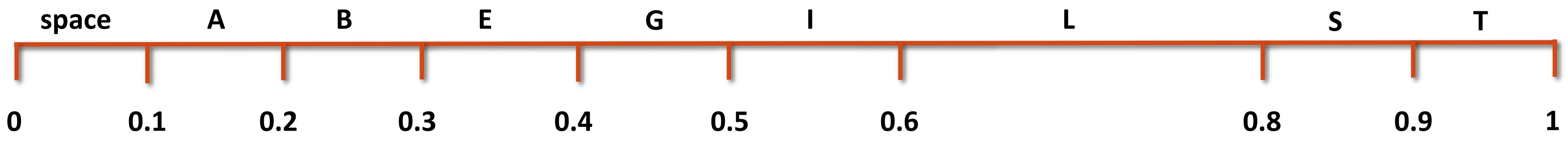
Character	**Probability**
SPACE	1/10
A	1/10
B	1/10
E	1/10
G	1/10
I	1/10
L	2/10
S	1/10
T	1/10

Please refer class notes for a step-by-step solution to this problem and other Arithmetic coding examples.

Arithmetic Coding

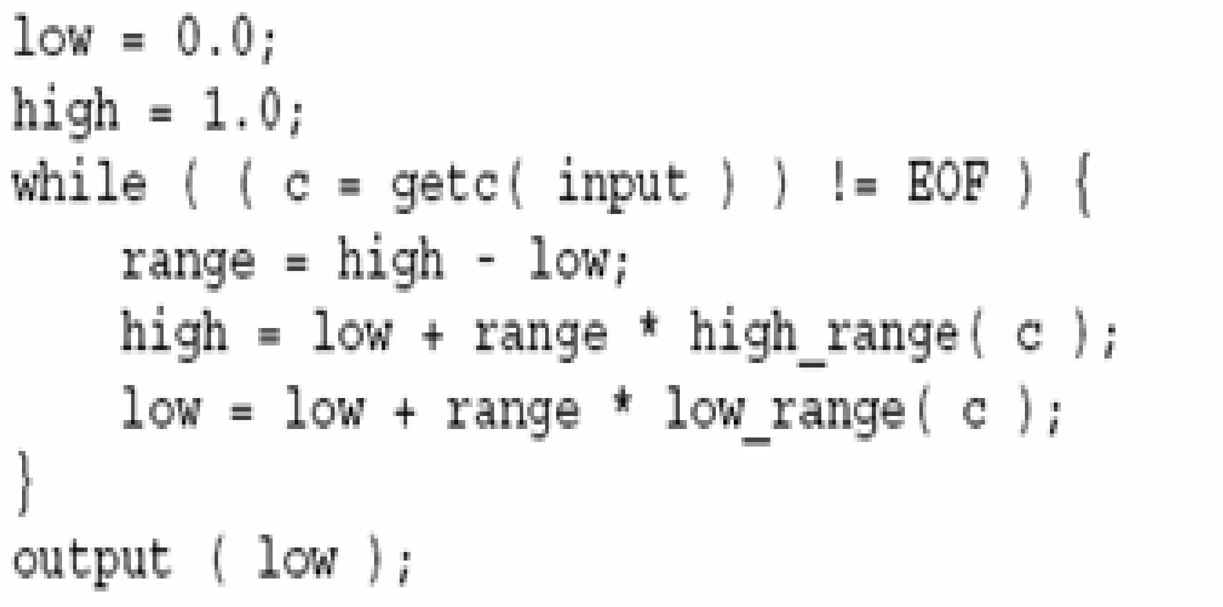
- Once the probabilities are known, each character is assigned the portion of the 0 to 1 range that corresponds to its probability of appearance.

SPACE	0.00 <= r < 0.10
A	0.10 <= r < 0.20
B	0.20 <= r < 0.30
E	0.30 <= r < 0.40
G	0.40 <= r < 0.50
I	0.50 <= r < 0.60
L	0.60 <= r < 0.80
S	0.80 <= r < 0.90
T	0.90 <= r < 1.00



Arithmetic Coding

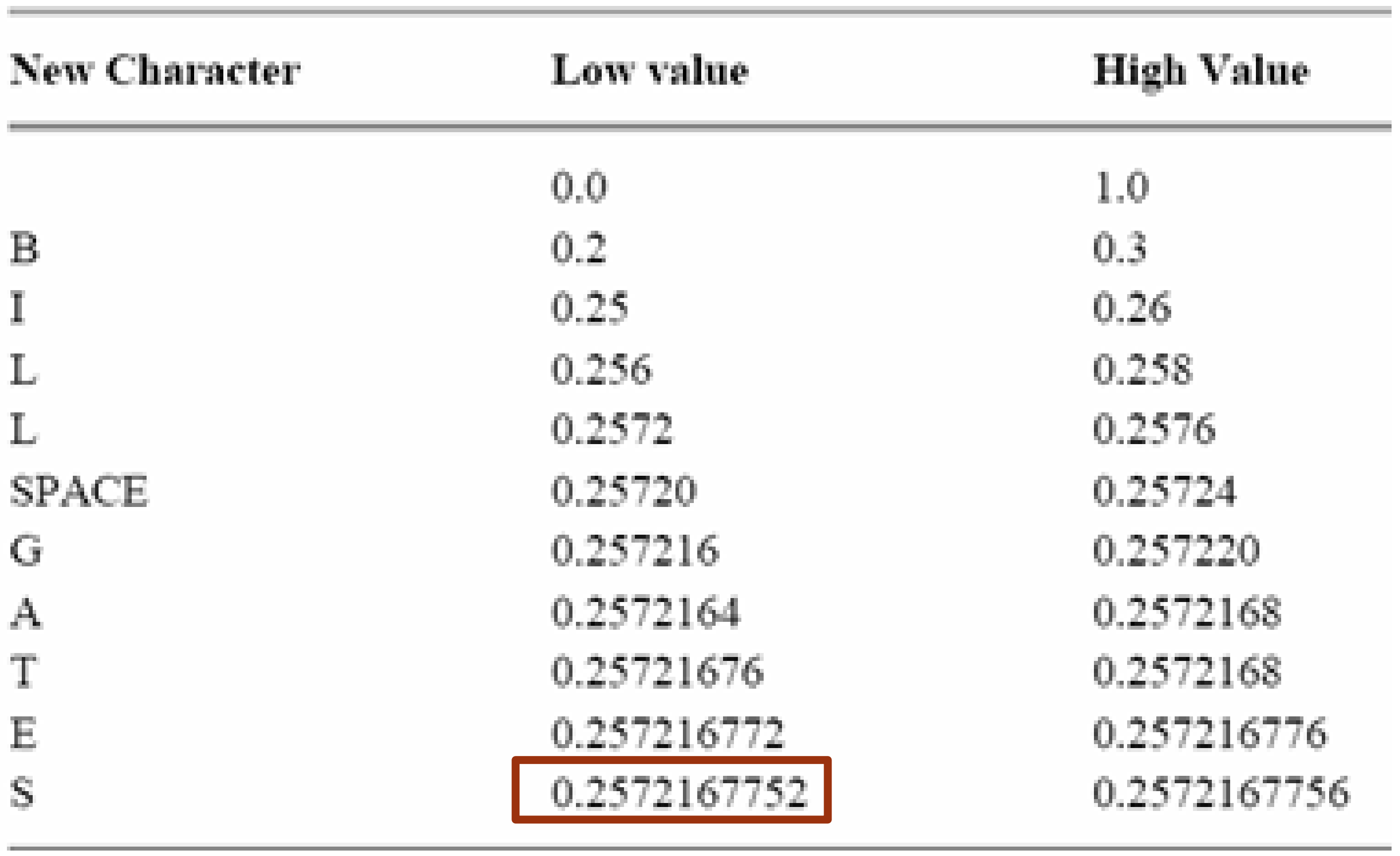
- To Encode:



```
low = 0.0;
high = 1.0;
while ( ( c = getc( input ) ) != EOF ) {
    range = high - low;
    high = low + range * high_range( c );
    low = low + range * low_range( c );
}
output ( low );
```

Arithmetic Coding

- To encode this number, track the range it could fall in.
- After the first character is encoded, the low end for this range is .20 and the high end is .30.
- During the rest of the encoding process, each new symbol will further restrict the possible range of the output number.
- The next character to be encoded, the letter I, owns the range .50 to .60 in the new sub range of .2 to .3



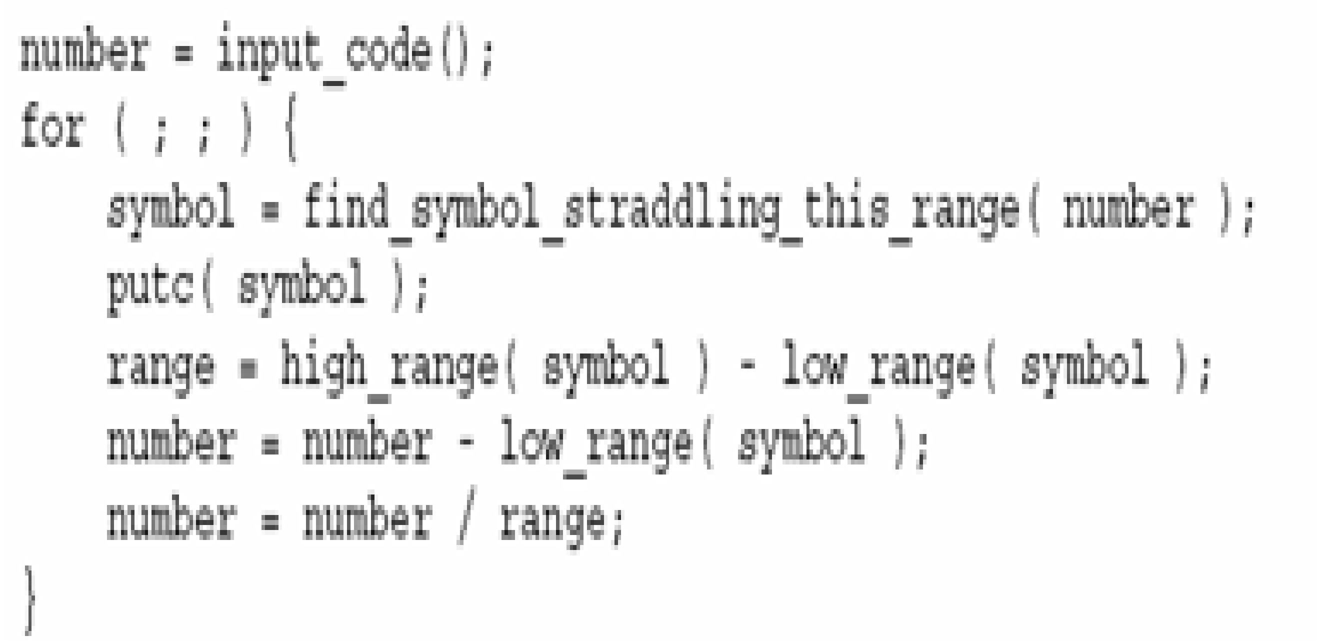
New Character	Low value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	$\boxed{0.2572167752}$	0.2572167756

So the final low value, **0.2572167752**, will uniquely encode the message “BILL GATES” using our present coding scheme.

Arithmetic Coding

- To decode the first character properly, the final coded message has to be a number greater than or equal to .20 and less than .30.

- To decode:



```
number = input_code();
for ( ; ; ) {
    symbol = find_symbol_straddling_this_range( number );
    putc( symbol );
    range = high_range( symbol ) - low_range( symbol );
    number = number - low_range( symbol );
    number = number / range;
}
```

LZW Encoding
(Lempel Ziv Welch)

- The LZW algorithm is a greedy algorithm in that it tries to recognize increasingly longer and longer phrases that are repetitive, and encode them.
  - Each phrase is defined to have a prefix that is equal to a previously encoded phrase plus one additional character in the alphabet. Note “alphabet” means the set of legal characters in the file. For a normal text file, this is the ascii character set.
- In many texts, certain sequences of characters occur with high frequency.
  - In English, for example, the word ***the*** occurs more often than any other sequence of three letters, with ***and***, ***ion***, and ***ing*** close behind.
- Although it is impossible to improve on Huffman encoding with any method that assigns a fixed encoding to each character, we can do better by encoding entire sequences of characters with just a few bits. LZW takes advantage of frequently occurring character sequences of any length. It typically produces an even smaller representation than is possible with Huffman trees.

LZW Encoding

- The algorithm works on the concept that integer codes (numbers) occupy less space in memory compared to the strings literals thus leading to compression of data.
- The **LZW** algorithm reads the sequence of characters and then starts grouping them into strings of repetitive patterns and then converts them to 12-bit integer codes which in turn compresses the data without any loss.

LZW Encoding

Advantage:

i. The LZW algorithm is faster compared to the other algorithms.

ii. The algorithm is simple, easy, and efficient.

iii. The LZW algorithm compresses data in a single pass.

iv. The LZW algorithm works more efficiently for files containing lots of repetitive data.

v. Another important characteristic of the LZW compression technique is that it builds the encoding and decoding table on the go and does not require any prior information about the input.

vi. For some text files, the LZW algorithm tends to have a compression ratio of 60 - 70%.

LZW Encoding
Mapping Table Formation

- The mapping table or the string table is predefined with all the default single characters. When the input string is sent, it starts with a single character and starts adding characters to it to form newer strings. Each time when it adds a character, it checks if the newly formed string is already in the table else it maps the newly formed string with the next code available in the mapping table, thus creating a new entry in the table.
- After updating the table with the new string pattern it again starts with the last added character and starts adding characters to it and again the same process continues. Hence, while we go through the input stream of data, the mapping table gets updated with newer string patterns that are used in the compression.

LZW Encoding
Algorithm (Encoder)

Please refer class notes for a step by step solution to this problem and other LZW encoding examples.

- Create the Table and enter all letters to the table
- Initialize string s to the first letter from the given input
- While any character still left in input
  - Read character c
  - If (s + c) is already in the table
    - s = s + c
  - Else
    - output codeword (s)
    - enter s + c to the table
    - s = c
- output codeword (s)

LZW Encoding
Algorithm (Decoder)

- Create the Table and enter all source letters to the table;
- Read priorCodeword and output one symbol corresponding to it;
- While codewords are still left
  - Read codeword;
  - priorString = string(priorCodeword);
  - **If** codeword is in dictionary
    - Enter in dictionary priorString + firstSymbol(string(codeword));
    - Output string(codeword);
  - **Else**
    - enter in dictionary priorString + firstSymbol(priorString);
    - Output priorString + firstSymbol(priorString);
- priorCodeword = codeword;