**Lossless Compression Algorithms (Repetitive Sequence Suppression)**

These methods are fairly straight forward to understand and implement. Their simplicity is their downfall in terms of attaining the best compression ratios. However, the methods have their applications, as mentioned below:

## 1 Simple Repetition Suppresion

If in a sequence a series on *n* successive tokens appears we can replace these with a token and a count number of occurences. We usually need to have a special **flag** to denote when the repated token appears

For Example

89400000000000000000000000000000000

we can replace with

894f32

where f is the flag for zero.

Compression savings depend on the content of the data.

Applications of this simple compression technique include:

* Suppression of zero's in a file (**Zero Length Supression**)
  + Silence in audio data, Pauses in conversation **etc.**
  + Bitmaps
  + Blanks in text or program source files
  + Backgrounds in images
* other regular image or data tokens

## 2 Run-length Encoding

This encoding method is frequently applied to images (or pixels in a scan line). It is a small compression component used in JPEG compression (Section [7.6](http://www.cs.cf.ac.uk/Dave/Multimedia/node234.html#sec:JPEG)).

In this instance, sequences of image elements $X_{1},X_{2},\ldots,X_{n}$ are mapped to pairs $(c_{1},l_{1}),(c_{2},l_{2}),\ldots,(c_{n},l_{n})$ where *ci* represent image intensity or colour and *li* the length of the *i*th run of pixels (Not dissimilar to zero length supression above).

For example:

Original Sequence:

111122233333311112222

can be encoded as:

(1,4),(2,3),(3,6),(1,4),(2,4)

The savings are dependent on the data. In the worst case (Random Noise) encoding is more heavy than original file: 2\*integer rather 1\* integer if data is represented as integers.

**Lossless Compression Algorithms (Pattern Substitution)**

This is a simple form of statistical encoding.

Here we substitue a frequently repeating pattern(s) with a code. The code is shorter than than pattern giving us compression.

A simple Pattern Substitution scheme could employ predefined code (for example replace all occurrences of `The' with the code '&').

More typically tokens are assigned to according to frequency of occurrenc of patterns:

* Count occurrence of tokens
* Sort in Descending order
* Assign some symbols to highest count tokens

A predefined symbol table may used ie assign code *i* to token *i*.

However, it is more usual to dynamically assign codes to tokens. The entropy encoding schemes below basically attempt to decide the optimum assignment of codes to achieve the best compression.

# Lossless Compression Algorithms (Entropy Encoding)

Lossless compression frequently involves some form of **entropy encoding** and are based in information theoretic techniques, Shannon is father of information theory and we briefly summarise information theory below before looking at specific entropy encoding methods.

## Huffman Coding

Huffman coding is based on the frequency of occurance of a data item (pixel in images). The principle is to use a lower number of bits to encode the data that occurs more frequently. Codes are stored in a **Code Book** which may be constructed for each image or a set of images. In all cases the code book plus encoded data must be transmitted to enable decoding.

The Huffman algorithm is now briefly summarised:

* A bottom-up approach

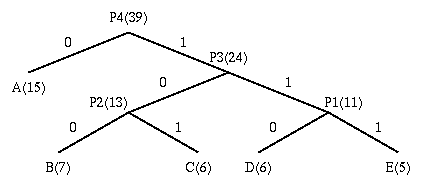
1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).

2. Repeat until the OPEN list has only one node left:

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.

(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.

(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.



Symbol Count log(1/p) Code Subtotal (# of bits)

------ ----- -------- --------- --------------------

A 15 1.38 0 15

B 7 2.48 100 21

C 6 2.70 101 18

D 6 2.70 110 18

E 5 2.96 111 15

TOTAL (# of bits): 87

The following points are worth noting about the above algorithm:

* Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. (There is a bit overhead for sending this, negligible if the data file is big.)
* **Unique Prefix Property**: no code is a prefix to any other code (all symbols are at the leaf nodes) -> great for decoder, unambiguous.
* If prior statistics are available and accurate, then Huffman coding is very good.

In the above example:

Number of bits needed for Huffman Coding is: 87 / 39 = 2.23

## Huffman Coding of Images

In order to encode images:

* Divide image up into 8x8 blocks
* Each block is a symbol to be coded
* compute Huffman codes for set of block
* Encode blocks accordingly

## Adaptive Huffman Coding

The basic Huffman algorithm has been extended, for the following reasons:

(a) The previous algorithms require the statistical knowledge which is often not available (e.g., live audio, video).

(b) Even when it is available, it could be a heavy overhead especially when many tables had to be sent when a non-order0 model is used, i.e. taking into account the impact of the previous symbol to the probability of the current symbol (e.g., "qu" often come together, ...).

The solution is to use adaptive algorithms. As an example, the Adaptive Huffman Coding is examined below. The idea is however applicable to other adaptive compression algorithms.

ENCODER DECODER

------- -------

Initialize\_model(); Initialize\_model();

while ((c = getc (input)) != eof) while ((c = decode (input)) != eof)

{ {

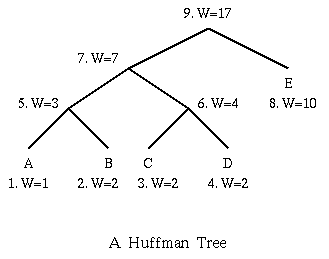
encode (c, output); putc (c, output);

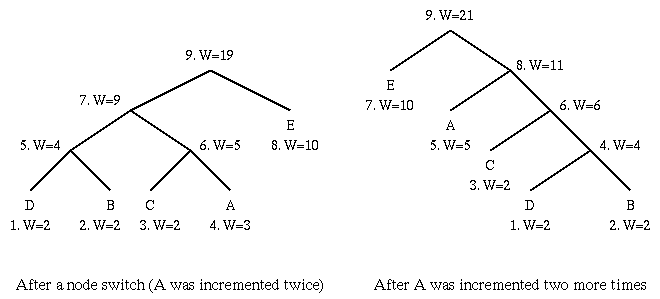
update\_model (c); update\_model (c);

} }

}

* The key is to have both encoder and decoder to use exactly the same *initialization* and *update\_model* routines.
* *update\_model* does two things: (a) increment the count, (b) update the Huffman tree (Fig [7.2](http://www.cs.cf.ac.uk/Dave/Multimedia/node212.html#hufftree)).
  + During the updates, the Huffman tree will be maintained its *sibling property*, i.e. the nodes (internal and leaf) are arranged in order of increasing weights (see figure).
  + When *swapping* is necessary, the farthest node with weight W is swapped with the node whose weight has just been increased to W+1. **Note:** If the node with weight W has a subtree beneath it, then the subtree will go with it.
  + The Huffman tree could look very different after node swapping (Fig [7.2](http://www.cs.cf.ac.uk/Dave/Multimedia/node212.html#hufftree)), e.g., in the third tree, node A is again swapped and becomes the #5 node. It is now encoded using only 2 bits.





**Note:** Code for a particular symbol changes during the adaptive coding process.

Huffman coding and the like use an integer number (k) of bits for each symbol, hence k is never less than 1. Sometimes, e.g., when sending a 1-bit image, compression becomes impossible.

* Idea: Suppose alphabet was

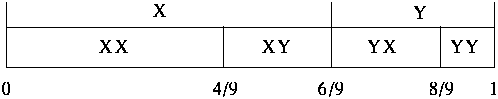
*X*, *Y*

and

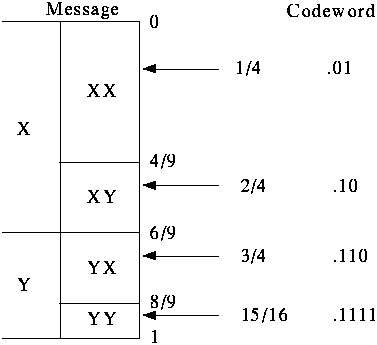
prob(X) = 2/3

prob(Y) = 1/3

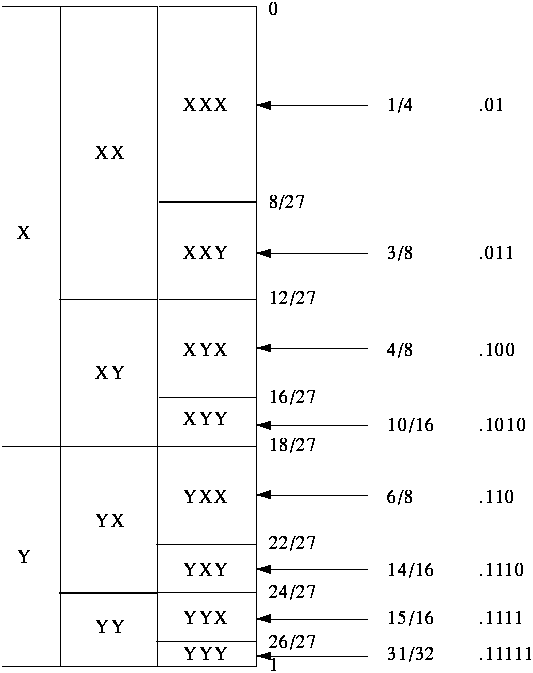
* If we are only concerned with encoding length 2 messages, then we can map all possible messages to intervals in the range [0..1]:



* To encode message, just send enough bits of a binary fraction that uniquely specifies the interval.



* Similarly, we can map all possible length 3 messages to intervals in the range [0..1]:



* Q: How to encode X Y X X Y X ?

Q: What about an alphabet with 26 symbols, or 256 symbols, ...?

* In general, number of bits is determined by the size of the interval.

Examples:

* + first interval is 8/27, needs 2 bits -> 2/3 bit per symbol (X)
  + last interval is 1/27, need 5 bits
* In general, need $- \log p$ bits to represent interval of size *p*. Approaches optimal encoding as message length got to infinity.
* Problem: how to determine probabilities?
  + Simple idea is to use adaptive model: Start with guess of symbol frequencies. Update frequency with each new symbol.
  + Another idea is to take account of intersymbol probabilities, e.g., Prediction by Partial Matching.

## Lempel-Ziv-Welch (LZW) Algorithm

The LZW algorithm is a very common compression technique.

Suppose we want to encode the Oxford Concise English dictionary which contains about 159,000 entries. Why not just transmit each word as an 18 bit number?

**Problems:**

* Too many bits,
* everyone needs a dictionary,
* only works for English text.
* **Solution**: Find a way to build the dictionary adaptively.
* Original methods due to Ziv and Lempel in 1977 and 1978. Terry Welch improved the scheme in 1984 (called LZW compression).
* It is used in UNIX *compress* -- 1D token stream (similar to below)
* It used in GIF comprerssion -- 2D window tokens (treat image as with Huffman Coding Above).

*Reference:* Terry A. Welch, "A Technique for High Performance Data Compression", IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.

The LZW Compression Algorithm can summarised as follows:

w = NIL;

while ( read a character k )

{

if wk exists in the dictionary

w = wk;

else

add wk to the dictionary;

output the code for w;

w = k;

}

* Original LZW used dictionary with 4K entries, first 256 (0-255) are ASCII codes.

**Example:**

Input string is "^WED^WE^WEE^WEB^WET".

w k output index symbol

-----------------------------------------

NIL ^

^ W ^ 256 ^W

W E W 257 WE

E D E 258 ED

D ^ D 259 D^

^ W

^W E 256 260 ^WE

E ^ E 261 E^

^ W

^W E

^WE E 260 262 ^WEE

E ^

E^ W 261 263 E^W

W E

WE B 257 264 WEB

B ^ B 265 B^

^ W

^W E

^WE T 260 266 ^WET

T EOF T

* A 19-symbol input has been reduced to 7-symbol plus 5-code output. Each code/symbol will need more than 8 bits, say 9 bits.
* Usually, compression doesn't start until a large number of bytes (e.g., > 100) are read in.

The LZW Decompression Algorithm is as follows:

read a character k;

output k;

w = k;

while ( read a character k )

/\* k could be a character or a code. \*/

{

entry = dictionary entry for k;

output entry;

add w + entry[0] to dictionary;

w = entry;

}

**Example (continued):**

Input string is "^WED<256>E<260><261><257>B<260>T".

w k output index symbol

-------------------------------------------

^ ^

^ W W 256 ^W

W E E 257 WE

E D D 258 ED

D <256> ^W 259 D^

<256> E E 260 ^WE

E <260> ^WE 261 E^

<260> <261> E^ 262 ^WEE

<261> <257> WE 263 E^W

<257> B B 264 WEB

B <260> ^WE 265 B^

<260> T T 266 ^WET

* Problem: What if we run out of dictionary space?
  + Solution 1: Keep track of unused entries and use LRU
  + Solution 2: Monitor compression performance and flush dictionary when performance is poor.
* Implementation Note: LZW can be made *really* fast; it grabs a fixed number of bits from input stream, so bit parsing is very easy. Table lookup is automatic.

## Entropy Encoding Summary

* Huffman maps fixed length symbols to variable length codes. Optimal only when symbol probabilities are powers of 2.
* Arithmetic maps entire message to real number range based on statistics. Theoretically optimal for long messages, but optimality depends on data model. Also can be CPU/memory intensive.
* Lempel-Ziv-Welch is a dictionary-based compression method. It maps a variable number of symbols to a fixed length code.
* Adaptive algorithms do not need a priori estimation of probabilities, they are more useful in real applications.