Shannon-Fano Coding

- Given a set of symbols and their frequencies (probabilities), construct a good variable-length coding scheme.
- The Shannon-Fano approach uses the principle of the yes-no question game.
- **Algorithm**: Divide the set of symbols into two subsets such that the sum of the probabilities in the first set approximately equals the sum of probabilities for the second set.
  Use 0 to select the first set, 1 to select the second set.

  Repeat recursively until all sets are singletons.
- The standard implementation is not clever about dividing the set of symbols. The set is a list of symbols in descending frequency order.
  The list is simply split into two sublists at the best place.

Shannon-Fano Coding, cont'd

Example:

Symbol	Freq	Split 1	Split 2	Split 3	Split 4	Code
e	0.35	0	0			00
t	0.25		1			01
a	0.15	1	0	0		100
o	0.08			1		101
i	0.06		1	0		110
n	0.06			1	0	1110
s	0.05				1	1111

- Perhaps there are better ways to divide the sets, but it is irrelevant because Huffman codes are better (optimal in fact).

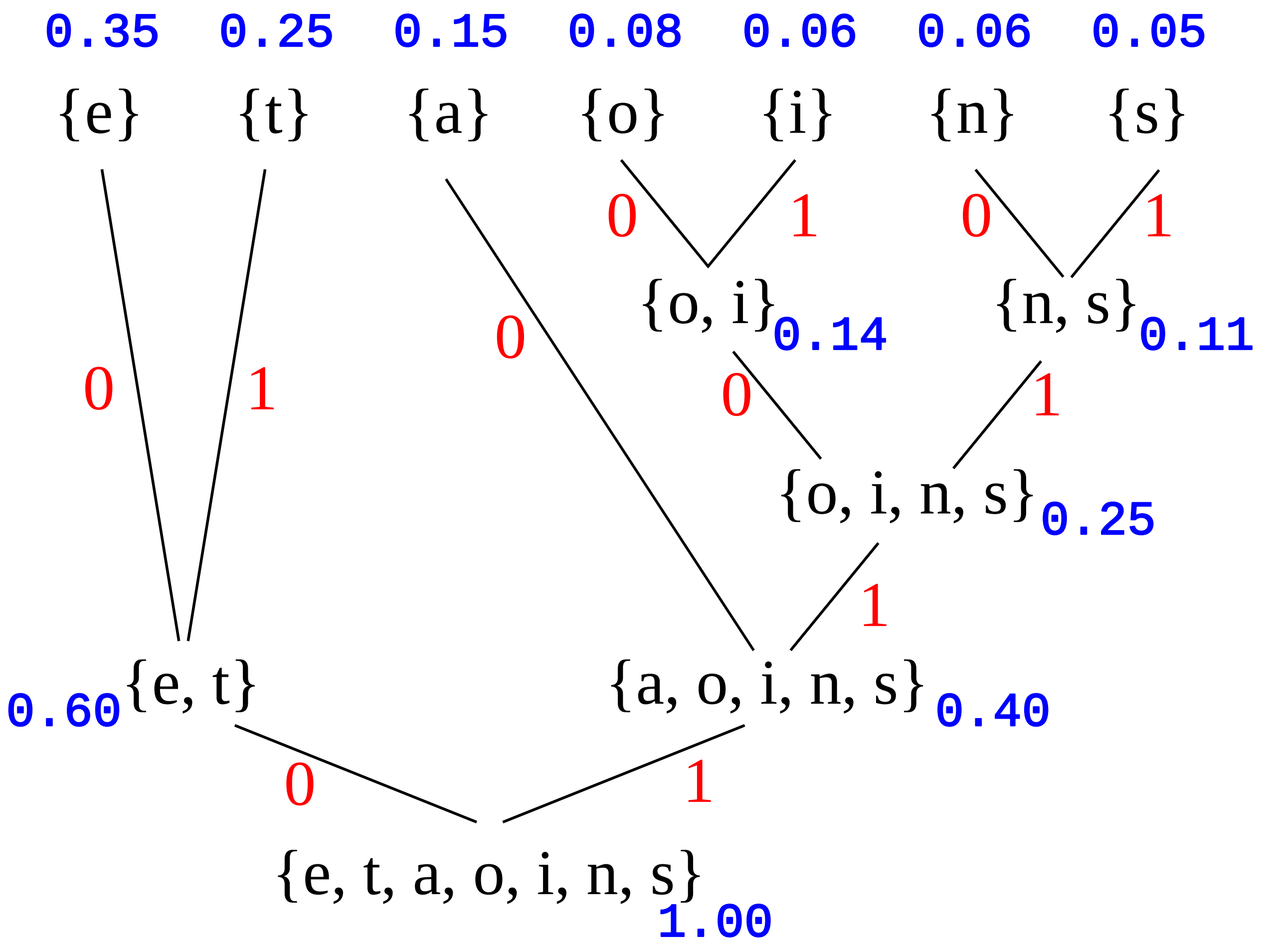
Huffman Codes

- Shannon-Fano starts with one big set of N symbols and keeps splitting it until we have with N sets of one symbol each.
- Huffman starts with N sets of one symbol each and keeps combining until it ends up with one set of N symbols.
- Shannon-Fano is a top-down approach; Huffman is bottom-up.

The Algorithm

1. Form N sets, each containing one of the symbols.
2. Find the two sets with the smallest total probabilities for their symbols;
   associate the symbols in one set with 0 and the other with 1.
   merge the two sets.
3. If more than one set remains, repeat step 2.
4. The code for each symbol is the sequence of associated 0 and 1 bits, in reverse order.

Huffman Coding Example



Turning the picture upside-down gives the code tree.

A Taxonomy of Lossless Coding Schemes

Coding is a mapping from a source alphabet S to a code alphabet X:

$$S^+ \Rightarrow X^+$$

Block Coding: one source symbol is mapped to one or more code symbols.

Non-Singular Block Coding: each source symbol is represented by a different sequence of code symbols

Uniquely Decodable Block Coding: no two sequences of source symbols map to the same sequence of code symbols

Instantaneous Block Coding: code sequences are decodable from left to right (equivalent to having the prefix property)

(Huffman coding is the optimal block coding scheme)

Non-Block Coding: one or more source symbols map to one or more code symbols. (All current lossless algorithms in common use, e.g. gzip, fall into this category)

Optimality of Huffman Coding

- The Huffman algorithm constructs the optimal set of instantaneous codes – in the sense that no other instantaneous codes would have less redundancy.
- Unless the symbol probabilities are all powers of 0.5, there is redundancy in the coding scheme.
- Redundancy can be reduced by forming an *extension alphabet* and constructing the Huffman codes for that.

  Given an alphabet S, the second extension S^2 is defined as

  $$S^2 = \{(a, b) \mid a, b \in S\}$$

  and the probability $P_{(a,b)} = P_a \times P_b$
- Similarly for S^3, S^4, ...
- Redundancy is progressively reduced as we move to bigger extension alphabets.

Example with Extension Codes

First Extension:

$S = \{a, b, c, d\}$ $P_a = 0.4$, $P_b = 0.3$, $P_c = 0.2$, $P_d = 0.1$

Huffman codes are: a = **`0`**, b = **`10`**, c = **`110`**, d = **`111`**.

Average code length = 1.9 bits per symbol

Second Extension:

$S^2 = \{aa, ab, ac, ad, ba, ... dd\}$ $P_{aa} = 0.16$, $P_{ab} = 0.12$, ... $P_{dd} = 0.01$

Huffman codes are: aa = **`000`**, ab = **`100`**, ba = **`101`**, bb = **`110`**, ac = **`0010`**, ...
bd = **`01110`**, db = **`01111`**, cd = **`11110`**, dc = **`111110`**, dd = **`111111`**.

Average code length = 3.73 bits per symbol in the S^2 alphabet, which equals 1.865 bits per symbol in the S alphabet.

Third extension, Fourth Extension ... ?

Forget it! The rate of convergence is slow and the size of the coding table is growing exponentially. This is not the way to squeeze redundancy out of the encodings. (Arithmetic coding, coming soon, does that better.)

Some Theory

Kraft-MacMillan Inequality (or Kraft Inequality):

Suppose we have a block coding scheme where each symbol $S_1, S_2 \ldots S_n$ in a source alphabet is mapped into codewords $X_1, X_2 \ldots X_n$. We write $L_i = |X_i|$ as the length of codeword X_i.

A necessary and sufficient condition for the existance of an instantaneous coding scheme with lengths $L_1, L_2 \ldots L_n$ is

$$\sum_i r^{-L_i} \leq 1$$

where r is the number of symbols in the code alphabet. For binary, r = 2.

We can write the average length of a code as $\bar{L} = \sum_i P_i L_i$

Assuming a binary code alphabet, it can be shown that the entropy H satisfies $H \leq \bar{L}$ and that $H = \bar{L}$ only if $L_i = -\log P_i$ for all *i*.

The Counting Argument

- Impossible to guarantee that a compression method will compress all inputs.
- Equivalent to the equally ludicrous proposition that a file can be made as small as you please by applying the compression method repeatedly.
- The 'counting argument' is often used to demolish such claims.
- Given a file size of length N bits, there are 2^N such files. Suppose that every one of these files is compressed. Then, after compression, each one must have a size of N-1 bits or fewer. But there are only 2^{N-1} different compressed files, at most.
  Therefore two or more of the original files must be transformed to the same compressed file.
  Implying that the decompression algorithm cannot reconstruct all the original files.
- **The implication**: whatever the lossless compression algorithm, there must be some files for which the size is *increased* by the algorithm.

[There are US patents for compression algorithms which violate the counting argument.]

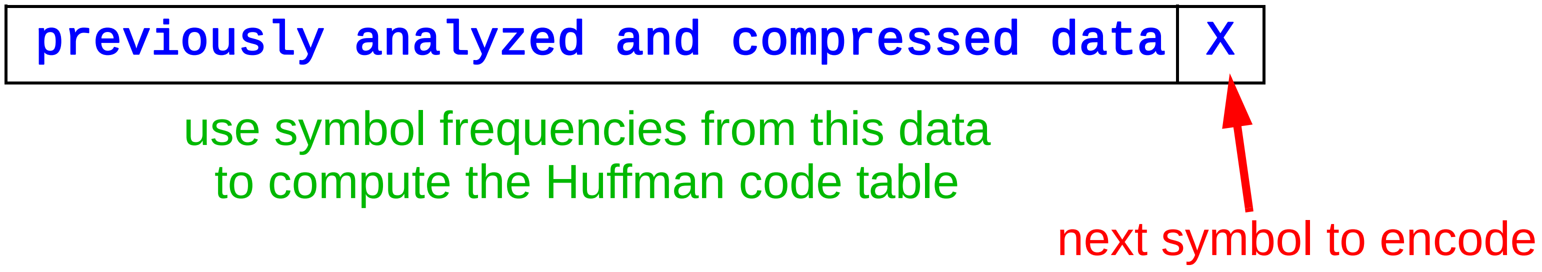
Adaptive Huffman Coding

The normal Huffman Coding approach can be used in two ways.

1. Predefined code tables are built into the compression & decompression programs.
2. The *compression* program makes two passes over the data.
   Pass 1: collects frequency statistics, builds the code table, and outputs a representation of the code table.
   Pass 2: compresses.
   The *decompression* program inputs the code table and then uses it to decompress the following data.

Neither approach is satisfactory.

Adaptive Huffman Coding learns the symbol frequencies while processing the data and continuously updates the Huffman code table.


previously analyzed and compressed data
X
use symbol frequencies from this data to compute the Huffman code table
next symbol to encode

Adaptive Huffman, cont'd

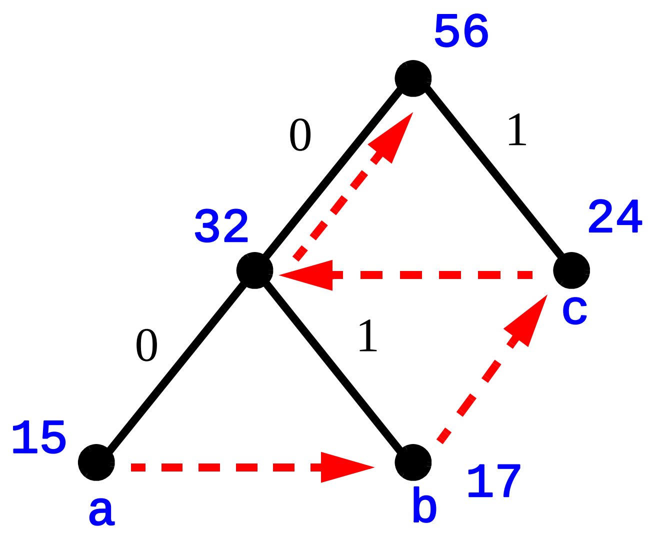
- The compression program uses frequencies from positions 1 through *k*-1 to construct a coding table for encoding the symbol in position *k*.
- The decompression program will have decoded symbols in positions 1 through *k*-1 before it gets to the *k*-th symbol . Therefore it can mirror the compression program and build the same coding table.
- Must solve the *zero-frequency* problem. A symbol that has not appeared yet has a frequency of zero and no Huffman code.
  Usually solved by intializing all 256 symbols to have a frequency of one and incrementing for each occurrence.
- An efficient algorithm to update the Huffman Coding table is required. After much work in the early part of a file, the codes tend to stabilize and change rarely.
  Published algorithms are by Gallager, Cormack & Horspool, Knuth, Vitter.
- The obsolete `compact` program on Unix used the Gallager algorithm. (Since superceded by `compress` and then by `gzip` and now by `bzip2`.)

C & H Adaptive Huffman Algorithm

The algorithm maintains a Huffman coding tree.

- Each leaf node corresponds to a symbol, and has an associated frequency.
- Each non-leaf node has a frequency computed as the sum of the frequencies attached to the leaf nodes in its subtree.

The tree nodes are threaded by the pointers that maintain a linked list. This list implements an ordering Π which sorts the nodes into frequency order.



Cormack & Horspool Algorithm:

```
procedure INCREMENT( N: Node, INCR: Integer ):
   NXT := successor of N in Π;
   if N->parent = Root then
    Root->frequency := Root->frequency + INCR;
    N-> frequency := N->frequency + INCR;
    if N->frequency > NXT->frequency then
      exchange subtrees rooted at N and NXT
      (and exchange N, NXT in the Π ordering)
   else
    if NXT = N->parent then
      NXT := successor of NXT in Π;
    MIN := min(INCR, NXT->frequency-N->frequency);
    REMDR := INCR - MIN;
    if MIN > 0 then
      INCREMENT(N->parent, MIN);
      N->frequency := N->frequency + MIN;
    if REMDR > 0 then
      exchange subtrees rooted at N and NXT
      (and exchange N, NXT in the Π ordering);
      INCREMENT(N, REMDR);
```

Adaptive Huffman, cont'd

- Adaptive Huffman coding gives almost indistinguishable compression performance from two-pass Huffman coding. (Sometimes it's better.)
- Too slow to be competitive (too much bit manipulation when coding/decoding), and gives much worse compression than zip, gzip, etc..
- Most current general-purpose lossless compression methods are adaptive.
- Storage for frequency counters may be a problem. Use of 16 bit integers leads to overflow problems, even 32 bit integers may overflow occasionally. Usually solved by dividing all counters by 2 at regular intervals (an aging process).
- The initial tree can contain all possible symbols.

  Alternatively, we can add each symbol to the tree only when needed.

  Requires the encoder to transmit an escape code and the symbol when it occurs for the first time. (The escape code can be a symbol in the Huffman tree.)