Huffman Coding

Coding Preliminaries

Code: Source message --- -f-----> code words

(alphabet A) (alphabet B)

alphanumeric symbols binary symbols

$|A| = N$ $|B|=2$

A code is

Distinct: mapping f is one-to-one.

Block-to-Block (ASCII – EBCDIC)

Block-to-Variable or VLC (variable length code)(Huffman)

Variable-to-Block (Arithmetic)

Variable-to-Variable (LZ family)

Average code length

Let *lj* denote the length of the binary code assigned to some symbol a_j with a probability p_j, then the average code length is given by $\bar{l}$

$$\bar{l} = \sum_{j=1}^{n} p_j l_j$$

Prefix Code:A code is said to have prefix property if no code word or bit pattern is a prefix of other code word.

UD – Uniquely decodable

Let $S_1 = (a_1, a_2, ..., a_n)$ and $S_2 = (b_1, b_2, ..., b_m)$ be two sequences of some letters from alphabeti α. Let $f : \alpha^p \to \beta^q$ be a variable length code. We say f is UD if and only if

$$f(a_1) \bullet f(a_2) \bullet\bullet\bullet f(a_n) = f(b_1) \bullet f(b_2) \bullet\bullet\bullet f(b_m)$$

implies that $a_1, a_2, ..., a_n$ is identically equal to $b_1, b_2, ..., b_m$. That is, $a_1 = b_1$, $a_2 = b_2$, etc and n=m.

Example Codes: 8 symbols $a_1, a_2, ..., a_8$,

probabilities codes

a_i	$p(a_i)$	Code A	Code B	Code C	Code D	Code E	Code F
a1	0.40	000	0	010	0	0	1
a2	0.15	001	1	011	011	01	001
a3	0.15	010	00	00	1010	011	011
a4	0.10	011	01	100	1011	0111	010
a5	0.10	100	10	101	10000	01111	0001
a6	0.05	101	11	110	10001	011111	00001
a7	0.04	110	000	1110	10010	0111111	000001
a8	0.01	111	001	1111	10011	01111111	000000
Avg.length		3	1.5	2.9	2.85	2.71	2.55

Code A, violates Morse's principle, not efficient (instantaneously decodable)

Code B, not uniquely decodable

Code C, Prefix code that violates Morse's principle

Code D, UD but not prefix

Code E, not instantaneously decodable (need look-ahead to decode)

Code F, UD, ID, Prefix and obeys Morse's principle

Note

1. Code A is optimal if all probabilities are the same, each taking $\lceil \log_2 N \rceil$ bits, where N is the number of symbols.
2. See code 5 and code 6 in K. Sayood, p29. Code 5 (a=0, b=01,c=11) is not prefix, not instantaneously decodable but is uniquely decodable, because there is only one way to decode a string '01 11 11 11 11 11 11 11 11 ' which will not have left over dangling bits. But if we interpret as '0 11 11 11 11 11 11 11 11 1' , a dangling left over '1' will remain.
3. Code 6 (a=0,b=01,c=10) decodable in two different ways without any decoding bit. The sequence ' 0 10 10 10 10 10 10 10 10'= acccccccc but can also be parsed

as '01 01 01 01 01 01 01 01 0'= bbbbbbbba. Both are valid interpretation. So, it is not UD, not prefix

Obviously, every prefix code is UD, but the converse is not true as we have seen.

Sufficient condition for a prefix code

If the code words are the leaf nodes of a binary tree , the code satisfies the prefix condition. In general this is true for any *d*-ary tree with *d* symbols in the alphabet. Why restrict to prefix code? Is it possible to find shorter code if we do not impose prefix property? Fortunately, the answer to this is NO. For any non-prefix uniquely decodable code, we can always find a prefix code with the same codeword lengths.

The lengths of the code words of uniquely decodable codes (by implication, the lengths of any prefix code) can be characterized by what is called the Kraft-McMillan inequality which is presented next.

The Kraft-McMillan Inequality:

Theorem 1: Let C be a code with N symbols or *codewords* with length $l_1, l_2, ..., l_N$. If C is uniquely decodable, then

$$K(C) = \sum_{i=1}^{N} 2^{-l_i} \leq 1$$

Proof: p.32 K.Sayood

The proof is based on computing nth power of $K(C)$, where n is an arbitrary positive integer. If $K(C)$ is greater than 1, this quantity will increase exponentially; if not the inequality is justified.

$$[\sum_{i=1}^{N} 2^{-l_i}]^n = [\sum_{i_1=1}^{N} 2^{-l_{i_1}}]^n [\sum_{i_2=1}^{N} 2^{-l_{i_2}}]^n \ldots\ldots [\sum_{i_n=1}^{N} 2^{-l_{i_n}}]^n$$

$$= \sum_{i_1=1}^{N} \sum_{i_2=1}^{N} \cdots \sum_{i_n=1}^{N} 2^{-(l_{i_1} + l_{i_2} + \ldots\ldots + l_{i_n})}$$

The quantity $l_{i_1} + l_{i_2} + ... + l_{i_n}$ is simply the sum of lengths of code words whose minimum value is *n* (if all the code words were of lengths 1). The maximum value of the exponent is *nl* where *l*= max($l_{i_1}, l_{i_2}, ..., l_{i_n}$). Therefore, we can write the summation as

$$K(C)^n = \sum_{k=n}^{nl} A_k 2^{-k}$$

where A_k is the combinations of *n* codewords that have a combined length of *k*.

Example to illustrate the proof

$l_1 = 1, l_2 = 2, l_3 = 2, n = 3$ (Note N is 5, not 3)

$l = \max(1,2,2) = 2$ $\qquad\qquad nl = 3x2 = 6$

$$[\sum_{i=1}^{3} 2^{-l_i}]^3 = (2^{-1} + 2^{-2} + 2^{-2})(2^{-1} + 2^{-2} + 2^{-2})(2^{-1} + 2^{-2} + 2^{-2})$$

$$= \sum_{k=n}^{nl} A_k 2^{-k} = 2^{-3} + 6 \bullet 2^{-4} + 12 \bullet 2^{-5} + 8 \bullet 2^{-2}$$

$$A_3 = 1, A_4 = 6, A_5 = 12, A_6 = 8$$

111	112*	112*
121*	122	122
121*	122	122
211*	212	212
221	222	222
221	222	222
211*	212	212
221	222	222
221	222	222

The example illustrates how the sizes of A_k are determined. The combinations, for example, marked with * contributes to the coefficient of 2^{-4} and there are 6 of them so $A_4 = 6$ and so on. The number of possible binary sequences of length *k* is 2^k. **If the code is uniquely decodable, then each sequence can represent one and only one sequence of code words.** Therefore, the number of possible combination of code words whose combined length is *k* cannot be greater than 2^k. Thus

$$A_k < 2^k$$

$$\sum_{k=n}^{nl} A_k 2^{-k} \le \sum_{k=n}^{nl} 2^k \cdot 2^{-k} \le \sum_{k=n}^{nl} 1 = nl - n + 1 = n(l-1)+1$$

$$\therefore [K(C)]^n = \sum_{k=n}^{nl} A_k 2^{-k} \le n(l-1)+1$$

If $K(C)$ is greater than 1, $[K(C)]^n$ goes exponentially, but $n(l\text{-}1)+1$ goes linearly with n.

Hence $K(C) \le 1$, Or $\sum_{i=1}^{N} 2^{-l_i} \le 1$

The converse of Theorem 1 is also true, as given in Theorem 2..

Theorem 2: Given a set of integers $l_1, l_2, ..., l_N$ such that $\sum_{i=1}^{N} 2^{-l_i} \le 1$, then we can find a prefix code with codeword length $l_1, l_2, ..., l_N$

See proof in Khalid Saywood, p.33. An example illustrating the proof is given below.

Proof: Given the lengths satisfying the stated property, we will construct a prefix code.

Assume $l_1 \le l_2 \le l_3 \le l_4 \le l_5$

$l_1 = 1, \quad l_2 = 2, \quad l_3 = 3, \quad l_4 = 4, \quad l_5 = 4$

$1 \le 1/2 + 1/4 + 1/8 + 1/16 + 1/16$

Define a sequence of numbers $w_1, w_2, ..., w_N$ as follows:

$$w_1 = 0$$

$$w_j = \sum_{i=1}^{j-1} 2^{l_j - l_i}$$

such that $j > 1$.. The binary representation of w_j for $j > 1$ would take $\lfloor \log_2 w_j \rfloor$ bits. We will use these binary representations to construct a prefix code. Note that the binary representation of w_j is less than or equal to l_j. This is obviously true for w_1. For $j > 1$,

$$\log_2 w_j = \log_2[\sum_{i=1}^{j-1} 2^{l_j - l_i}] = \log_2[2^{l_j} \sum_{i=1}^{j-1} 2^{-l_i}] = l_j + \log_2[\sum_{i=1}^{j-1} 2^{l_i}] \le l_j$$

The last inequality is due to the given hypothesis of the theorem. The second item in the right hand side, the logarithm of a number less than 1 is negative, so that the summation of this with l_j has to be less than or equal to l_j.

$$
\begin{aligned}
&w_1 = 0 \\
&w_2 = 2^{l_2 - l_1} = 2^{2-1} = 2 \\
&w_3 = 2^{l_3 - l_1} + 2^{l_3 - l_2} = 2 + 4 = 6 \\
&w_4 = 2^{l_4 - l_1} + 2^{l_4 - l_2} + 2^{l_4 - l_3} = 8 + 4 + 2 = 14 \\
&w_5 = 2^{l_5 - l_1} + 2^{l_5 - l_2} + 2^{l_5 - l_3} + 2^{l_5 - l_4} = 8 + 4 + 2 + 1 = 15
\end{aligned}
$$

$$
\begin{aligned}
&length = 2 \qquad && w_2 = (10)_2 \qquad && = 2 \\
&length = 3 \qquad && w_3 = 110 \qquad && = 6 \\
&length = 4 \qquad && w_4 = 1110 \qquad && = 14 \\
&length = 5 \qquad && w_5 = 1111 \qquad && = 15
\end{aligned}
$$

Using the binary representation of w_j, we can devise a binary code as follows. If $\left|\log_2 w_j\right| = l_j$, then the *j*th codeword c_j is the binary representation of w_j. If $\left|\log_2 w_j\right| < l_j$, then c_j is the concatenation of binary representation of w_j with $l_j - \left|\log_2 w_j\right|$ 0's appended at the end. The code thus formed $C = (c_1, c_2, ..., c_N)$ is a prefix code. (Formal proof : See Sayood, p.34).

w_j	l_j	$\log_2 w_j$	$l_j -$ $\log_2 w_j$ '0	code
	1			0
2	2	1	0	100
6	3	3		110
14	4	4		1110
15	4	4		1111

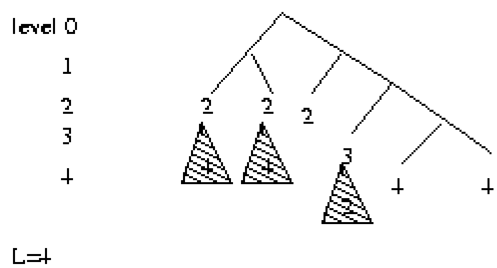
Note the proof assumes only UD property but not the prefix property. But the resulting code has the prefix property. If we know that the code is prefix then a much simpler proof exists.

Theorem 3: Given a prefix code of codeword lengths $l_1, l_2, ..., l_N$, show that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$

We prove the theorem by using a binary tree embedding technique. Every prefix code can be represented in the paths of a binary tree.

Example to illustrate the proof

$$\{l_j\} = \{2,2,2,3,4,4\}$$



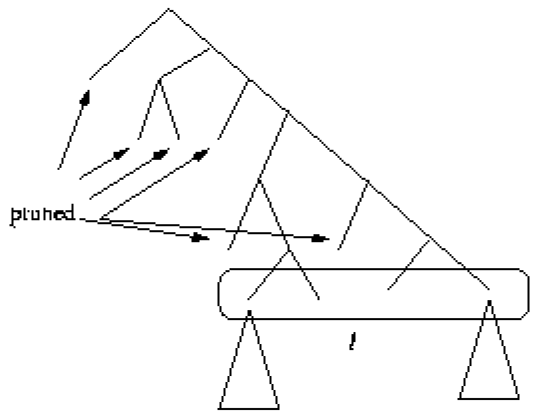
Proof: Given a binary prefix code with word length $\{l_j\}$, we may embed it in a binary tree of depth *L* where $L \geq \max\{l_j\}$, since each of the prefix code must define a unique path in a binary tree. This embedding assigns to each codeword of length l_j a node on level l_j to serve as the terminal node. Then prune the entire sub-tree below that node, wiping out 2^{L-l_j} nodes. Since we cannot prune from a level-*L* tree more than 2^L nodes that were there to start with, we must have $\sum_{i=1}^{N} 2^{L-l_i} \leq 2^L$. Diving by 2^L , we get

$\sum_{i=1}^{N} 2^{-l_i} \leq 1$ which is the Kraft Inequality.

The proof of the converse is more interesting.

Theorem 4: *Given a set of integers $\{l_j\}$ satisfying Kraft inequality, there is a binary prefix code with these lengths.*

Proof: That is, for each level *l* we must show that after we have successfully embedded all words with lengths $l_j < l$, enough nodes at level *l* remain un-pruned so that we can embed a codeword there for each j such that $l_j = l$.



That is,

$$2^l - \sum_{j:l_j<l} 2^{l-l_j} \geq \left|\{j : l_j = l\right| = \text{c} \qquad \ldots(1)$$

The right hand side is simply the number of nodes with $l = l_j$ But

$$\text{c}= \left|\{j : l_j = l\}\right| = \sum_{j:l_j=l} 1 = \sum_{j:l_j=l} 2^0 = \sum_{j:l_j=l} 2^{l-l_j}$$

Therefore, from(1), $$2^l - \sum_{j:l_j<l} 2^{l-l_j} \geq \sum_{j:l_j=l} 2^{l-l_j}$$

Or $$2^l \geq \sum_{j:l_j<l} 2^{l-l_j} + \sum_{j:l_j=l} 2^{l-l_j} \geq \sum_{j:l_j\leq l} 2^{l-l_j}$$

Dividing both sides by 2^l, we have $1 \geq \sum_{j:l_j\leq l} 2^{-l_j}$

Since we have $$\sum_{all \quad j} 2^{-l_j} \geq \sum_{j:l_j\leq l} 2^{-l_j}$$

We must have $$\sum_{j:l_j\leq l} 2^{-l_j} \leq \sum_{all \quad j} 2^{-l_j} \leq 1$$

(All derivations above are valid for d-ary tree. Put d^{-l_j} to replace 2^{-l_j}.)

<u>Examples of Prefix Code</u>:

- Unary code(Salomon pp.47-48)
- Variations (Salomon, p.49)
- General Unary Code(Salomon,p.48)
- Elias Code
- Golomb Code (Salomon, p.53)
- Fibonacci Code
- Shannon-Fano Code
- Huffman Code

<u>Elias Code</u>

Exact value of probabilities are not needed, only the ranking(in terms of its length) x is needed.The rank x is mapped to $\lfloor \log_2 x \rfloor$ number 0's concatenated with the binary representation of x.

Rank											
1	1										
2	0	1	0								
3	0	1	1								
4	0	0	1	0	0						
5	0	0	1	0	1						
6	0	0	1	1	0						
32	0	0	0	0	0	1	0	0	0	0	0

Fibonacci Code

Express rank *x* in terms of a weighted number system where the Fibonacci number are the weights. Then x is encoded as the reverse Fibonacci sequence followed by binary'1'.

N	21	13	8	5	3	2	1
1							1
2						1	0
3					1	0	0
4					1	0	1
5				1	0	0	0
6				1	0	0	1
7				1	0	1	0
8				1	1	0	0
16		1	0	0	1	0	0
32	1	0	1	0	1	0	0

Code							
1	1						
0	1	1					
0	0	1	1				
1	0	1	1				
0	0	0	1	1			
1	0	0	1	1			
0	1	0	1	1			
0	0	1	1	1			
0	0	1	0	0	1	1	
0	0	1	0	1	0	1	1

Shannon-Fano Code:

i	p_i				
1	0.25	1	0		
2	0.2	1	1		
3	0.15	0	0	0	
4	0.15	0	0	1	
5	0.1	0	1	0	
6	0.1	0	1	1	0
7	0.05	0	1	1	1

Code			
1	0		
1	1		
0	0	0	
0	0	1	
0	1	0	
0	1	1	0
0	1	1	1

Average length = 2.7 bit/symbol
Entropy=2.67bit
very good

1	0.25	1	1	
2	0.25	1	0	
3	0.125	0	1	1
4	0.125	0	1	0
5	0.125	0	0	1
6	0.125	0	0	0

Average length = 2.5 bit/symbol
Entropy=2.5bit
perfect code!

The method produces best result if the splits are perfect which happens when the probabilities are 2^{-k} and $\sum 2^{-k} = 1$. This property is also true for Huffman code.

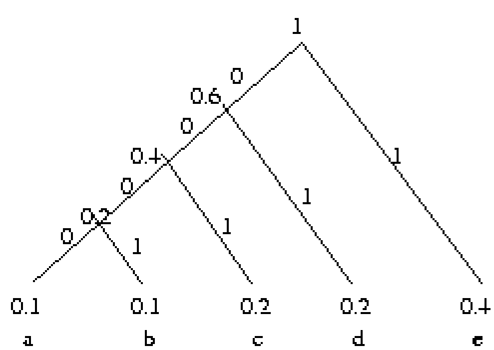
Huffman Code:

- Shannon-Fano is top-down. If you draw a binary tree, the symbols near to the root get codes assigned to them first.
- Huffman is bottom-up. It starts assigning codes from leaf nodes.

Huffman invented this code as an undergraduate at MIT and managed to skip the final exam as a reward!

Same offer: If you come up with an original idea in this course worth publishing in a reputable journal, you may skip the final exam.

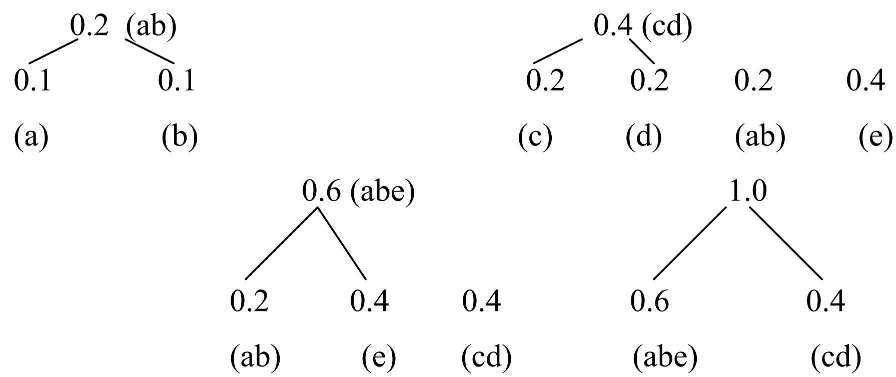
- Huffman code construction. (Encoding).
- Complexity O(nlogn), storage O(n).
- Huffman code decoding
- Average code length $\bar{l} = \sum p_i l_i$. Variance of code $v = \sum_i (l_i - \bar{l})^2 p_i$

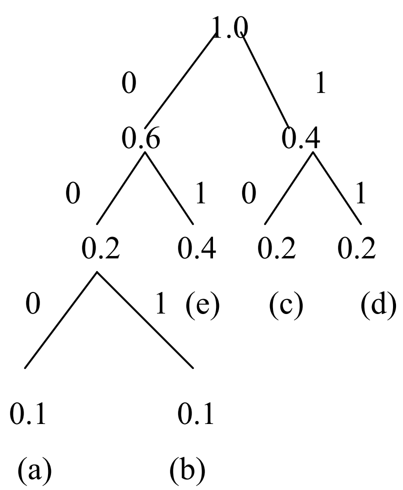


$\bar{l} = 2.2 \qquad v = 1.36$

The codes are a=000,b=001,c=10,d=11 and e=01. (Draw this: combine (a,b), then (c,d). Then combone (ab,) with e and then (a,b,e) with (c,d). You will see the variance is 0.16 although the average length is the same (2.2). The tree looks more bushy.

Rule: *During the iterative steps of ordering the probabilities, move as far right as possible for the composite symbols at higher level in the sort order.*





<u>code</u>

a= 000

b=001

c=10

d=11 and e=01

What is the advantage of having a code with minimum variance?

See discussion on pp.44-45 Sayood.

- Optimality of Huffman code * $H_s \le \vec{l} \le H_s + 1$
- Non-binary Huffman code * Golomb and Rice code
- Adaptive Huffman tree

- Canonical Huffman tree

Optimality of Huffman Code

Theorem 5 *Huffman code is a minimum average length* $(\bar{l})$ *binary prefix code.*

Lemma 1 *If* $p(a_1) \geq p(a_2)$*, then it must be that* $l_1 \leq l_2$ *for the code to have minimum average* $(\bar{l})$ *codelength.*

$$\bar{l} = p(a_1)l_1 + p(a_2)l_2 + \sum_{i=2}^{n} p_i l_i$$

$$= p(a_1)l_1 + p(a_2)l_2 + Q$$

For the sake of contradiction, assume $l_1 > l_2$. Then, we can exchange the codes for a_1 and a_2, giving modified average length:

$$\bar{l}^* = p(a_1)l_2 + p(a_2)l_1 + Q$$

Therefore,

$$\bar{l} - \bar{l}^* = p(a_1)(l_1 - l_2) + p(a_2)(l_2 - l_1)$$

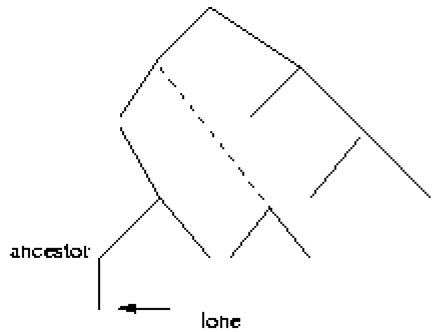
$$= p(a_1)(l_1 - l_2) - p(a_2)(l_1 - l_2)$$

$$= C[p(a_1) - p(a_2)], \qquad C = l_1 - l_2$$

Thus, $\bar{l} > \bar{l}^*$ So $\bar{l}$ is not a minimum, a contradiction.

Lemma 2 *A minimum average length* $\bar{l}$ *binary code has at least two codes of maximum length* l_M*.*

Proof: Let $C = (C_1, C_2, \ldots, C_M)$ be a minimum $\bar{l}$ binary prefix code, such that $p_1 \geq p_2 \geq, \ldots, \geq p_M$. Let l_M be the length of the least likely source symbol whose code is C_M and has length l_M. So, the leaf node sits at the deepest level of the binary tree. It cannot be a lone node at that level, because, if it were, we can replace it by its ancestor on the previous level. Since shuffling the code words to nodes on any fixed level does not affect $\bar{l}$, we may assume that C_{M-1} and C_M stem from the same ancestor, with C_{M-1}, say, encoding in 0 and C_M encoding in 1.That is we put these two leaf nodes on consecutive positions of the Huffman tree. Let's redefine l_{M-1} to be the depth of the node that is the common ancestor of C_M and C_{M-1}, while letting each l_j for $1 \le j \le M-2$ retain the original meaning.



This converts the problem to construct a binary tree with M-1 terminal nodes so as to minimize $\bar{l} = \sum_{j=2}^{M-2} p_j l_j + (p_{M-1} + p_M)[l_{M-1} + 1]$.

Now, define modified probabilities as

$$\{p_j^*, 1 \le j \le M-1\}$$

$$p_{M-1}^* = p_{M-1} + p_M, \qquad p_j^* = p_j \qquad 1 \le j \le M-2$$

Then $\bar{l} = \sum_{j=1}^{M-2} p_j^* l_j + p_{M-1}^*(l_{M-1} + 1) = \sum_{j=1}^{M-1} p_j^* l_j + p_{M-1}^*$

But p^*_{M-1} is a constant of the problem and does not affect how we construct the tree. This has converted our original problem to that of finding a tree with *M*-1 terminal nodes that is optimum for probabilities $\{p_j^*, 1 \le j \le M-1\}$. This, in turn, can be reduced to an (*M*-2) node problem by assigning the code words corresponding to the smallest two of modified probabilities p^*_j to a pair of terminal nodes that share a common immediate ancestor. But, that is, precisely what the next merge operation in Huffman algorithm does! Iterating this argument *M*-1 times establishes that Huffman algorithm produces minimum average length prefix binary codes.

This argument is also valid for d-ary codes!

Theorem 6 *The entropy H of* $\{p_j, 1 \le j \le n\}$ *satisfies* $0 \le H \le \log n$

(The proof is the same as we did in our last lecture on Information Theory)

We need the relations:

$\ln x \le x-1$ ($\ln x = x-1$ if $x=1$). Substituting x by $\frac{1}{x}$, we get $\ln x \ge 1-\frac{1}{x}$

$[\ln\frac{1}{x} \le \frac{1}{x}-1, \ln(1)-\ln(x) \le \frac{1}{x}-1 \therefore \ln(1)-(\frac{1}{x}-1) \le \ln(x) \therefore \ln(x) \ge 1-\frac{1}{x}+\ln(1) \ge 1-\frac{1}{x}]$

Again, equality hold if x=1.

Proof : Left equality (H=0) holds if for some j, $p_j = 1$ and all the p_i's $= 0$. Right equality holds if $p_j = 1/n,\ \forall j$. To obtain the left inequality, note $-p\log p \ge 0$ for $0 \le p \le 1$ with equality iff $p=1$, Hence $H \ge 0$

To obtain the right inequality, we use the fact $\sum_j p_j = 1$ to derive

$$
\begin{aligned}
&\log n - H \\
&\quad = (\sum_j p_j)\log n + \sum_j p_j \log p_j \\
&\quad = \sum_j p_j(\log n + \log p_j) \\
&\quad = \sum_j p_j(\log np_j) \\
&\quad \ge \sum_j p_j(\log_2 e(1-\frac{1}{np_j})) \\
&\quad \ge k(p_j - \frac{1}{np_j}) \qquad\qquad k = \log_2 e
\end{aligned}
$$

and equality holds iff $p_j = 1/n,\ \forall j$. Then, $\log n - H \ge k(\sum_j p_j - \sum_j 1/n) = k(1-1) = 0$

This proves $0 \le H \le \log n$

Lower Bound for average length

$$\bar{l} - H$$

$$= \sum_i p_i l_i - (-\sum_i p_i \log p_i)$$

$$= \sum_i p_i (l_i + \log p_i)$$

$$= \sum_i p_i \log(p_i 2^{l_i})$$

Let $x = p_i 2^{l_i}$. Using the relation $\log x \geq \log_x e(1 - 1/x)$,we then have

$$\log(p_i 2^{l_i}) \geq \log_2 e(1 - 1/p_i 2^{l_i})$$

Thus, $\bar{l} - H$

$$\geq \log_2 e \sum_i p_i (1 - 2^{-l_i}/p_i)$$

$$\geq \log_2 e \sum_i (p_i - 2^{-l_i})$$

$$= \log_2 e[\sum_i p_i - \sum_i 2^{-l_i}]$$

$$= K[1 - C]$$

where $C = \sum_i 2^{-l_i} \leq 1$ (By Kraft inequality). Thus, $\bar{l} - H \geq 0$ .Equality holds when $x = 1$

Thus, $\bar{l} \geq H$ **.** The average code length for any binary prefix code is at least as large as the entropy of the source. [The above derivation is also true for d-ary prefix code. Replace 2^{-l_i} by d^{-l_i} and $\log_2 e$ by $\log_d e$.]

Upper Bound

See Sayood, pp.46-51. (Reading Assignment)

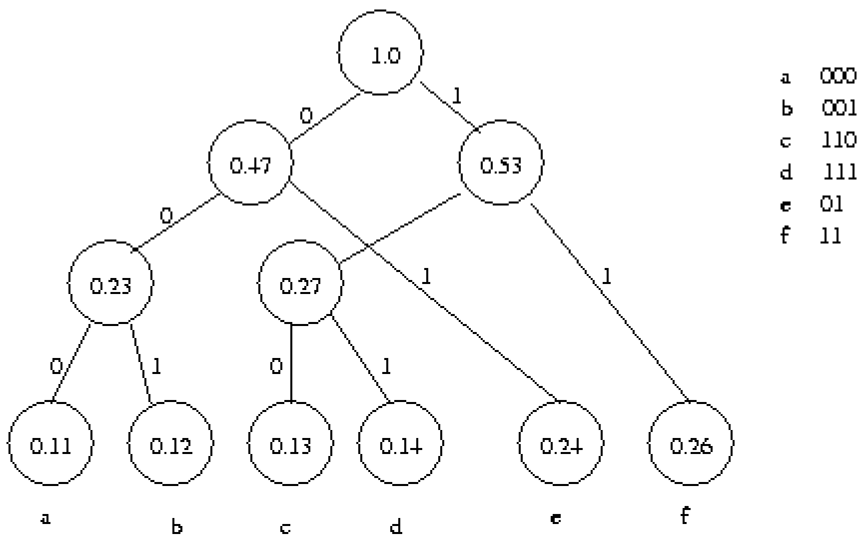
Theorem 7: $H(S) \leq \bar{l} \langle H(S) + 1$

Canonical Huffman Code

Huffman code has some major disadvantages. If the alphabet size is large, viz. word based Huffman need to code each word of a large English dictionary.

1. Space: with *n* symbols leaf nodes, there are *n*-1 internal nodes. Each internal node has two pointers, each leaf stores a pointer to a symbol value and a flag saying it is a leaf node. Thus needs around 4n words.
2. Decoding is slow – it has to traverse the whole tree with a lot of pointer chasing with no locality of storage access. Each bit needs a memory access during decoding.

Consider the following example of probability distribution:



As we know, if there are *n*-1 internal nodes, we can create 2^{n-1} new Huffman codes by re-labeling (at each internal node there are two choices of labeling with 0 and 1). So, we should have $2^5 = 32$ Huffman codes. But, let us create the codes as 00*x*, 10*x*, 01, and 11 where *x* = 0 or 1. let *A*=00, *B*=10, *C*=01, *D*=11. The codes are *Ax, Bx, C, D*. Any permutation of *A, B, C,* D will lead to a valid Huffman code. There are 4! permutation and *x* has two possible values – hence a total of 96 Huffman codes! (Actually 94 if we do the enumeration.) This means that there are Huffman codes that cannot be generated by Huffman tree. Canonic Huffman code is one such Huffman code.

a 000

b 001

c 010

d 011

e 10

f 11

is one such example. Note all the codes of same length are consecutive binary integers of given length. Given the length of the Huffman words, these codes can be generated as follows.

Algorithm to Generate the Canonical Huffman Codes

1. Take the largest length group with length $l_{\max}$. If there are k_1 words of this length, generate the first k_1 binary numbers of length $l_{\max}$.
2. If the next length is l_2, extract l_2 bit prefix of the last code of the previous group. Add 1 k_2 times, where k_2 is the number of words of length l_2, to get the code for the group.
3. Iterate the process for all groups l_i.

Example : The lengths are (5,5,5,5,3,2,2,2)

0 0 0 0 0

0 0 0 0 1

0 0 0 1 0

0 0 0 1 1

0 0 1

01

10

11

Note, not all length sequences are valid. For example, there cannot be a Huffman code for (5,5,5,5,3,2,2,2,2). Problem: why?

.

The algorithm to generate the codes seems very straight forward as described above in the code generation steps. If the first number using l_i bits is somehow figured out for the code group of length l_i, then we know the remaining codes in this group are consecutive numbers. Let us denote by *first(l)* be the first number in the code group of length *l*. For encoding purpose we only need *first(l)* for values of *l* equal to $l_1, l_2, \ldots, l_{\max}$ which are the lengths of the codes. But, we will compute *first(l)* for all values of *l* in the range $l_1 \le l \le l_{\max}$ since, as we will see later, we will need this for the purpose of decoding.Let *numl*(*l*) denote the number of codes of length *l*, $l_1 \le l \le l_{\max}$. The computation of *first(l)* is given by the two line code:

---

first($l_{\max}$):=0;

for l := $l_{\max}$ -1 *down to* 1 *do*

$$first(l) := \lceil (first(l+1) + numl(l+1))/2 \rceil;$$

Given the lengths as (6,6,6,6,6,6,6,5,5,5,5,5,3,3,3,3)

We have

l	1	2	3	4	5	6
numl(*l*)	0	0	4	0	5	7
first(l)	2	4	3	5	4	0

$first(6) = 0$

$first(5) = (0+7)/2 = 4$

$first(4) = (4+5)/2 = 5$

$first(3) = (5+0)/2 = 3$

$first(2) = (3+4)/2 = 4$

$first(1) = \lceil (4+0)/2 \rceil = 2$

Therefore,

l	1	2	3	4	5	6
$numl(l)$	0	0	4	0	5	7
$first(l)$	2	4	3	5	4	0

need the values for encoding

Given the array *first(l),* the algorithms steps can now be followed to obtain the canonical codes.

The expression $\lceil (first(l+1)+numl(l+1))/2 \rceil$ guarantees that the resulting code is a prefix. If $first(l+1)=n_1$ and $numl(l+1)=N$. $(n_1+N)/2$ is right shifted by one bit and ceiling operation add a 1 to it if it is an odd number. Convince yourself that implies prefix property.

Decoding

Canonical Huffman is very useful when the alphabet is large but fast decoding is necessary. The code is stored in consecutive memory addresses, along with symbol.

Let's do the example (5,5,5,5,3,2,2,2) again

l	1	2	3	4	5
numl	0	3	1	0	4
first(l)	2	1	1	2	0

Address	Symbol	Code				
0	a	0	0	0	0	0
1	b	0	0	0	0	1
2	c	0	0	0	1	0
3	d	0	0	0	1	1
4	e	0	0	1		
5	f	0	1			
6	g	1	0			
7	h	1	1			

$first(5) = 0$ [redacted]

$first(4) = \ (0+4)/2$ [redacted]

$first(3) = \ (2+0)/2$ [redacted]

$first(2) = \ (1+1)/2$ [redacted]

$first(1) = \ (1+3)/2$ [redacted]

Compute an array called *first_address(l)*

<u>Begin</u>

$first\_address(l_{\max}) \leftarrow 0;;$

$next\_available\_address \leftarrow 0 + numl(l_{\max});$

for $l = l_{\max}$ -1 down to 1 do {

if $num(l) \neq 0$ then {

$first\_address(l) \leftarrow next\_available\_address;$

$next\_available\_address \leftarrow first\_address(l) + numl(l);$

} else

$first\_address(l) \leftarrow 0;$

}
End

So, the result is: (you may verify the addresses for lengths are 5,3,2 are 0,4,5 respectively, in the above table).

l	1	2	3	4	5
first_address(l)	0	5	4	0	0

Now, we are ready to perform the decoding operation given an input bit string. We define a bit string variable v which stuffs bits into it as long as the binary number is less than *first(l)* . Note here we need the values of *first(l)* even if there is no code with length l. As soon as v becomes greater than or equal to first(l), we know we are in the middle of some group, so we need to have the off-set address in this group to access the symbol stored in an array in a RAM.

Decode

```
    While input is not exhausted do {
        l = 1;
        input bit v;
        while v < first(l)  {
            append next input bit to v;
            l = l+1;
        }
        difference = v – first(l);
        output symbol at first_address(l) + difference;
    }
```

Note for each symbol that is only one memory access, while for Huffman tree the memory access will be for each bit. For the example shown below, Huffman decoder will need 10 memory access as opposed to only 3 for canonic Huffman code.

Address	Symbol	Code				
0	a	0	0	0	0	0
1	b	0	0	0	0	1
2	c	0	0	0	1	0
3	d	0	0	0	1	1
4	e	0	0	1		

5	f	0	1
6	g	1	0
7	h	1	1

Input: 0 0 1 1 0 0 0 0 1 0

e g c

Now that we have a Huffman code that has a very fast decoding algorithm, the question is: given the probabilities, how do you obtain the lengths of the codes? One way will be to develop the regular Huffman tree, extract the length information and then don't use the code of the tree. Instead design a canonic codes using the length information. But, this actually defeats the original purpose where we were confronted with a large alphabet like the words in the English dictionary and we need good amount of storage and computation overhead to generate the length information. It is possible to obtain the lengths directly from probabilities by using a fairly complex data structure and algorithm (heap and a linear array for full binary tree , which you must have studied if you took a course on advanced data structure or design and analysis of algorithms) which will not be presented now. I would like to assign this as optional reading: from Witten, Moffat and Bell,pp.41-51 and David Salomon, pp.73-76.

Non-Binary Huffman Code (See Section 3.3, Sayood)

Adaptive Huffman Code (See Section 3.4, Sayood)