The Coding Problem

The source alphabet A of n symbols $\{a_1,a_2, \ldots a_n\}$ and a corresponding set of probability estimates $P=\{p_1,p_2,\ldots,p_n\}$ are given, such that $\sum_{1}^{n} p_i = 1$. The coding problem consists of deciding on a *code* giving a representation of each symbol a_i of the alphabet using strings over a channel alphabet B, which is usually $\{0,1\}$.

****

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

(alphabet A) (alphabet B)

alphanumeric symbols Channel alphabet= binary symbols

$|A| = n$ $|B|=2$

****

The symbol a_i may be drawn from a longer *message M* consisting of strings of source alphabet symbols, but at this point we are considering the symbol a_i in isolation. Sometimes we will denote the source alphabet A by the integers $\{1, 2, 3,\ldots,n\}$. Let the *codewords* for a particular coding algorithm be $C=\{c_1,c_2,\ldots c_n\}$ with corresponding lengths of codewords being $L=\{l_1,l_2,..,l_n\}$. Then the **average code length** $\bar{l}$ or the **expected codeword length** $E(C,P)$ is given by

$$E(C,P) = \bar{l} = \sum_{j=1}^{n} p_j l_j$$

Prefix-free Code: A code is said to have prefix property if no code word or bit pattern is a prefix of other code word. Sometimes prefix-free code is also called simply **prefix code**. A code is said to be **uniquely decodable or uniquely decipherable (UD)** if the message for the code string, if it exists, can be recovered unambiguously. The fundamental question is: how short can we make the average code length so that the code is UD. Consider the table below giving different codes for 8 symbols $a_1, a_2, ..., a_8$:

Example Codes: ,

	probabilities	codes					
ai	*p(ai)*	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 but 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), not prefix
Code F: UD, ID, and Prefix and obeys Morse's principle

Code D, E and F: are incomplete, as there are prefixes over the channel alphabets that are not used. For D, all four prefixes 00, 01, 10 and 11 do not occur etc. Code F is a *minimum redundancy* code which is also known as *Huffman* code which we will discuss later

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 Section 2.4, p.29, Sayood) Code 5 (a=0, b=01,c=11) is **not prefix, not instantaneously decodable but is uniquely decodable**. Consider the string '01 11 11 11 11 11 11 11 11'. There is only one way to decode this string which will not have leftover dangling bits. But if we interpret this as '0 11 11 11 11 11 11 11 11 1' , a dangling left over '1' will remain.
3. (See Section 2.4, p.29, Sayood) Code 6 (a=0,b=01,c=10) decodable in two different ways. 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.

Exercise: Find and justify a test for a UD code.

Note there is a whole family of codes that use bit-fractional codes which are not illustrated here in this table. For example, arithmetic codes which we will discuss later.

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

The Kraft-McMillan Inequality

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. If each symbol

a_i has a probability which is a negative power of 2, that is, $p_i = 2^{-k_i}$, then the self-information is $I(a_i) = -\log p_i = k_i$, a whole number. So, if we set $l_i = k_i$, this results in average code length or the expected code length equal to Shannon's entropy bound and hence cannot be further improved. We also have $\sum_{i=1}^{n} 2^{-l_i} = 1$. This led Kraft [1949] to formulate the famous Kraft inequality.

Theorem 1: *(**A necessary condition for UD code**) Let C be a code with n codewords with lengths* $l_1, l_2, ..., l_n$. *If C is uniquely decodable, then*

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

McMillan [1956] extended this result and showed that if the Kraft inequality is satisfied for some code *C*', then it is possible to find a UD prefix code *C* that will have exactly the same lengths of code words as those of *C*'.

Theorem 2: *(A sufficient condition for prefix code) Given a set of integers* $l_1, l_2, ..., l_n$, *that satisfy the inequality* $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, *we can always find a prefix code with codeword lengths* $l_1, l_2, ..., l_n$.

This code is also uniquely decipherable. Further more, this relationship is invertible, that is, if

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

is greater than 1, the code cannot be a prefix-free. As a simple but obvious example, if each code word has length 1, then *K*(*C*) = *n/2*, and a prefix-free code is possible only if *n*=2. The corresponding codes are '0' and '1'.

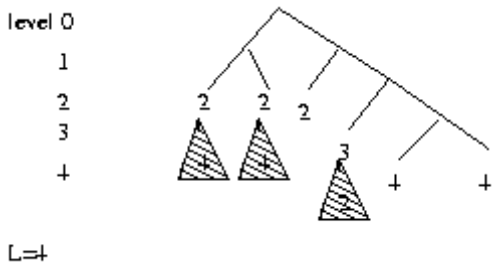
(Formal proofs for Theorem 1 and 2 are given in pp.32-34, Sayood)

Simpler Proofs for Theorems 1 and 2 for the Prefix Code

We prove the Theorem 1 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

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

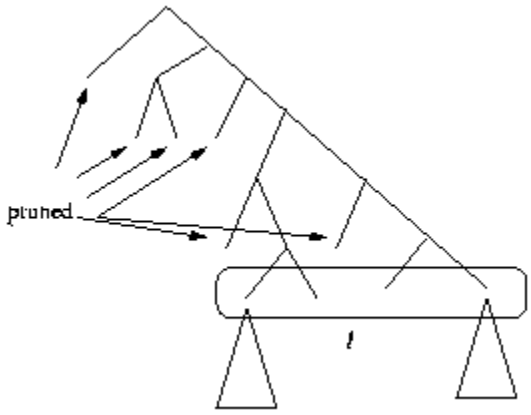


Proof: Given a binary prefix code with word lengths $\{l_j\}$, we may embed it in a binary tree of depth L where $L = \max\{l_j\}$, since each of the prefix code must define a unique path in a binary tree. This embedding assigns to each codeword of length l_j a node on level l_j to serve as the terminal node. Then the entire sub-tree below that node is pruned, 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 $\sum_{i=1}^{n} 2^{L-l_i} \le 2^L$. Diving by 2^L, we get $\sum_{i=1}^{n} 2^{-l_i} \le 1$ which is the Kraft Inequality.

The proof of Theorem 2 is more interesting.

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,

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

The right hand side of Eqn.(1) is simply the number of nodes with $l = l_j$. But

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

Therefore, from Eqn.(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 also have $$\sum_{all\ j} 2^{-l_j} \geq \sum_{j:l_j\leq l} 2^{-l_j}$$

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

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

Some Well Known Prefix Codes

(References: Chapters 1-4 of Moffat and Turpin, pp.47-53 of Salomom)

- Unary code
- Variations
- General Unary Code
- Elias Code
- Golomb Code
- Rice Code
- Fibonacci Code
- Shannon-Fano Code
- Huffman Code

Before we discuss these codes, we need to understand the models that are appropriate for estimation of the probabilities.

Probability Estimation

We learned that the compression system has three components: modeling, probability estimates and coding. We also know that we cannot device a code whose expected average length is less than the entropy of the source for the given model. The human beings are very good at building rather sophisticated model of our languages at a very early age. The human brain could have evolved through millions of years to achieve these characteristics. Consider the following texts:

" If you don't hurray up, you are going to be …. "
"If you don't put on a jacket, you are going to be …"
"If you don't do your assignments in the data compression course,
you are going to get …….".

We can easily fill in the gap. This is equivalent to predicting the next word(s). If we can do this sort of analysis, we may be able to find out a good estimate of the probabilities of the words in English language. Shannon (1951) undertook seminal work in this area and concluded that the entropy of English language is about 2 BPC. Later Cover and King(1978) improved this estimate to 1.3 BPC. Thus any lossless text compression algorithm that comes close to 2.0 BPC is supposed to be very good. Researchers (including our work at UCF) have come close to 2.3 for some specific corpus.

A natural question to ask is: where and how do we get the probability estimates of the source symbols? The answer to this question is extremely difficult. The order(-1) model, assumes equal probability for all symbols even if some symbols may not occur in the text. Thus for ASCII (ISO-646) the alphabet has 128 symbols, each having a probability of 1/128. The extension of ASCII (ISO-8859-1) has 256 symbols, and so the entropy of the source is 8 bits under Order (-1) model. The Unicode (for all languages of planet earth) uses 2 bytes for each symbol has entropy of 16 bits. These are called *static codes.* Both decoder and encoder know the model and there is no overhead to transmit the model from source to destination.

Semi-Static I Model

If a particular text does not have all the symbols of the alphabet, we can determine what symbols it has by a *first pass* of the text. Let's say the text has only 25 symbols. Then under Order(-1) model, the entropy of the source is given by

$$H(P) = -\sum_{i=1}^{25} \frac{1}{25} \log \frac{1}{25} = 4.64$$

Of course, one has to find an algorithm to encode the text that actually obtains this lower bound. In practice it might take 5 or 6 bits. The decoder needs to know the symbols of the alphabet which is sent as a *prelude* taking 8*25=200 bits plus count of the number of symbols, which takes 8 bits (since the maximum count could be 256). If we distribute this overhead on the entire message of length m, the average code length becomes 4.64 + 208/m. If we take m=128, as an example, this becomes 6.27 BPC.

Semi-Static II Model

Perhaps we can improve the situation if we can calculate the *self-probabilities* of the n characters in the message. That is, if the symbols s_i appears v_i times in the message, then

take $p_i=v_i/m$. The quantity $-\sum_{i=1}^{n} v_i \log \frac{v_i}{m}$ is called the *zero-order self-information* of the message and gives the lower bound on the number of bits required to encode the message provided the symbols are *iid*. This gives a figure of 4.22 BPC for the example discussed in Moffat and Turpin, p.23. It looks like an improvement, but it is not. Because, we have to include frequency of symbols information in the prelude, assuming conservatively that we need 4 bits per symbol to send the frequency information, we need 4X25=100 additional bits. This brings the total with m=128 to be 4.22+ (208+100)/128=6.63 which is worst than the previous scheme. Thus, making the model more and more complex does not always buy in compression ratio. We need to strike a balance between the modeling stage and the actual coding stage. If the message is short, the additional cost for complex modeling is not justifiable whereas if the file size is very large and the model is going to be used many times, the overhead could be amortized.

Static Codes

In this section, we will discuss some of simplest static codes. These codes are very suitable for coding a set of *m* integers with smaller values more probable than the larger values. Such integer sequences are often generated as an intermediate output for many data compressor (such as move-to-front method). Since most of the methods do not take into account the probabilities, their compression performance is relatively poor. But, they have very regular structures and can be encoded or decoded very fast. For some very special classes of probability distribution, these codes are also optimal. We assume that the message *M* consists of *m* integers from the source alphabet *S*={1,2,3, …, n} and that their probabilities are $p_1 > p_2 > \ldots > p_n$. We allow *n* to be unbounded in which case the probabilities are $p_1 > p_2 > \ldots > p_i > \ldots > 0$.

Unary Code

In a unary code, an integer *x* is encoded as a sequence of *x*-1 1's (or 0's) followed by one 0 (or 1) as shown below

m	Code	Alternate Code
1	0	1
2	10	01
3	110	001
4	1110	0001
…		…

The unary code is a zero redundancy code if the probability distribution is P= $\{\frac{1}{2},\frac{1}{4},\frac{1}{8},....,2^{-(m-1)},2^{-(m-1)}\}$. For an infinite sequence of numbers it is P= $\{\frac{1}{2},\frac{1}{4},\frac{1}{8},.....,0)$.

General Unary Code

In this scheme, rather than allocating one integer to a unary code, *a group of consecutive integers* are allocated to a unary code followed by a distinct binary code for each member

of the group. The code is best described by a triplet (start, *step, stop*). The *m*th code word has m 1's followed by a single 0 which is followed by all possible combinations of a= *start+m.step* binary digits. If *a=stop*, then the single 0 bit preceding the *a*-bits is dropped. For example, a (3,2,9) code is shown below.

m	a=**3**+m.**2**	mth codeword	# of codewords(2^a)	Range of integers
0	3	0xxx	2^3=8	0-7
1	5	10xxxxx	2^5=32	8-39
2	7	110xxxxxxx	2^7=128	40-167
3	**9**	111xxxxxxxxx	2^9=512	168-679

These codes are optimal for probability distribution (for the groups are: ½,1/4,1/8,1/8) is

$$P = \{\tfrac{1}{8}(\tfrac{1}{2}, \tfrac{1}{2}, ...8times)), \tfrac{1}{32}(\tfrac{1}{4}, \tfrac{1}{4}, ...32times), \tfrac{1}{128}(\tfrac{1}{8}, \tfrac{1}{8}, 128times), \tfrac{1}{256}(\tfrac{1}{8}, \tfrac{1}{8}, ..256times\}$$

Minimal Binary Code

A code is said to be a *minimal binary code* if all prefixes are used in the code. If we use the regular binary numbers to encode the first six integers as (000,001,010,011,100,101), we will miss '11' as a prefix. On the other hand the first six integers can be coded using a code (00,01,100,101,110,111). Note all possible prefixes of one bit (0,1) and all possible prefixes of two bits (00,01,10,11) appear in the code. These codes are also called *complete*. Note, the code of an integer varies depending on maximum value of *n.* Thus if *n*=3, the integer 2 gets encoded as '10' (the codes are 0, 10, 11) but when n=6, 2 is encoded as '01'. It is usually more efficient than the obvious binary numbers, each having $log_2 n$ bits. In general, for an alphabet of *n* symbols, the minimal binary codes have $k = 2^{\lceil \log_2 n \rceil} - n$ codewords that are $\lfloor \log_2 n \rfloor$ bits long and remaining $n\text{-}k=n\text{-}[2^{\lceil \log_2 n \rceil} - n] = 2n - 2^{\lceil \log_2 n \rceil}$ are $\lceil \log_2 n \rceil$ bits long. The shorter codes are allocated to the more probable symbols to minimize the expected code length. The minimal binary codes has the property that if *n* is a power of two and all symbols have equal probability (=1/*n*), then it is a zero-redundancy code. If *n* is not a power of two, it is minimum redundancy for the same equal-probability distribution and in effect becomes a Huffman code, as we will see later. The algorithm to construct the code is as follows: Given *n*, use first $k = 2^{\lceil \log n \rceil} - n$ combinations of $\lfloor \log n \rfloor$ bits to encode *k;* for the remaining combinations append a '0' and then a '1', giving a total of n codes. An example: *n*=11, *k*=16-11=5. So, the code is; 000,001,010, 011,100, 1010,1011,1100,1101,1110,1111.

Elias Code

The Elias code is an elegant compromise between minimal binary code and the unary code. This is also a family of infinite code, that is, it handles an integer *x* of arbitrarily large magnitude and takes *O*(*logx*) number of bits. The most common are the codes C_γ and C_δ, as shown in the following table. The code C_γ can be obtained by writing the binary numbers as 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010 then drop the most significant bit and replace this with a unique unary code prefix. The unary part takes $1+\lfloor \log_2 x \rfloor$ bits and the binary part takes $\lfloor \log_2 x \rfloor$ number of bits making a total of

$1+2\lfloor \log_2 x \rfloor$ bits for the C_γ code. . The code C_γ can be seen as a general unary code that consists of a *selector* part that indicates a range of values (answers the question: is it bigger than 1,3,7,….2^k-1) that collectively form buckets of size 1,2,4,…2^k, and a binary part that indicates a value within the bucket. Since the binary part has all its numbers with most significant bit '1', this bit can be dropped without any ambiguity.
The second Elias code C_δ uses a prefix part that is C_γ rather than a unary code and it takes a total of $1+2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$ bits to code an integer *x*. The algorithm to encode and decode Elias codes are given in Moffat and Turpin "Compression and Coding Algorithms" Chapter 3.

M	C_γ code	C_δ code
1	0	0
2	10 0	100 0
3	10 1	100 1
4	110 00	101 00
5	110 01	101 01
6	110 10	101 10
7	110 11	101 11
8	1110 000	11000 000
9	1110 001	11000 001
10	1110 010	11000 010

We conclude by citing a direct quote from this book: "The amazing thing about the Elias code is that they are shorter than the equivalent unary codes at all but a small finite number of codewords. The C_γ code is longer than unary code only when *x*=2 or *x*=4 and in each case by only one bit. Similarly, the C_δ code is larger than C_γ only when $x\varepsilon\{2..3,8..15\}$. On the other hand for large values of *x* both Elias codes are not just better than unary, but exponentially better."

The Elias codes are for any arbitrary probability distribution. To see why, assume a probability distribution *P* in which $p_1 > p_2 > \ldots > p_n$. Because of this distribution p_x must be less than $1/x$ for all $1 \le x \le n$. Because if it is not then for some value *x*, we must have

$$\sum_{j=1}^{x} p_j > \sum_{j=1}^{x} 1/x = 1$$

which contradicts the assumption that the sum of probabilities equals to 1. But, as we

know if $p_x \le 1/x$ then the symbol x must take at least $O(\log_2 x)$ bits. Thus, both the Elias codes take number of bits within a multiplicative constant of the entropy bound. The code words are fixed yet they are provably "not bad" and hence general purpose for any probability distribution.

Golomb Code

The Elias code has bucket sizes {1, 2, 4, 8, .. . 2^k,..}. For Golomb code the bucket size is a fixed constant b. Given the n integers to be encoded, first compute two other integers as

$$q = \frac{n-1}{b} \qquad \text{(Integer quotient)}$$

$$r = n - qb \qquad \text{(Integer remainder or residue)}$$

Now, encode q using a unary code and r using a minimal binary code. The concatenation of these two codes gives the Golomb code. The table below illustrates the Golomb code for b=5 and n=9:

n	*q*	*r*	*Golomb Code*
1	0	1	0 00
2	0	2	0 01
3	0	3	0 10
4	0	4	0 110
5	0	5	0 111
6	1	1	10 00
7	1	2	10 01
8	1	3	10 10
9	1	4	10 110

Rice Code

Rice code is a special case of a Golomb code where the bucket size is a power of 2, that is, for some fixed k the bucket size is $b=2^k$. Thus

$$q = \frac{n-1}{2^k} \qquad \text{(Integer quotient)}$$

$$r = n - qb \qquad \text{(integer remainder or residue)}$$

The example below illustrates Rice code for *k=2, b=4,*

n	*q*	*r*	*Golomb Code*
1	0	1	0 00
2	0	2	0 01
3	0	3	0 10
4	0	4	0 11
5	1	1	10 00
6	1	2	10 01
7	1	3	10 10

8	1	4	10 11
9	2	1	110 0

Division by 2^k has a simple shift register implementation:
1)For integer *n,*Take the low-order *k* bits of *n*-1 which gives the minimal binary part of the code; 2) Right shift *n*-1 by *k* bits and t take the least significant *k* bits, which gives the integer *q*, 3)Obtain the unary representation of *q*.
Concatenation of the unary and the binary part is the final Golomb code.

Both Rice and Golomb codes are extensively used in compression system. Golomb codes are particularly useful for Bernoulli distribution – a sequence of Bernoulli trials with probability of success given by *p*. Let p_x be the probability of the next success after *x* trials. Then, $p_1=p$, $p_2=p(1-p)$ etc. and in general, $P=[p(1-p)^{x-1}|1\le x]$. If the distribution *P* has this property and the parameter *b* is chosen as (See Turpin and Moffat, p.38)

$$b=\left\lceil \frac{\log_e 0.5}{\log_e(1-p)} \right\rceil \approx (\log_e 2)\times\frac{1}{p}$$

then Golomb code is a minimum redundancy code. Elias, Golomb and Rice codes have been extensively studied in the literature and several generalizations and mathematical properties of these codes are discussed in text and in the literature. The detail discussion of this material is outside the scope of this course.(Possible Term project)
****

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

Fibonacci Code
Express the integer 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'. See the above tables.

Further reading: (available at the reserved material desk at the library) 1)Moffat and Turpin: Chapter 3, pp.29-41. 2) P. Fenwick, Chapter 3, "Lossless Compression Handbook" (Ed. Sayood)

Minimum -Redundancy Code

A code is a *minimum-redundancy* code for a probability distribution *P* if its average length or expected code length $E(C,P) \le E(C',P)$ for every *n* symbol prefix-free code *C'*; thus, there is no other prefix-free whose average code length is strictly less than that of *C*. The code obviously has to obey the Kraft inequality.

Shannon-Fano Code

The Shannon-Fano code was the first attempt to find a minimum -redundancy code. The motivation for this algorithm is this: if the 0's and 1's are equally useful in the code, then each bit position in the code word should correspond to a choice between groups of symbols whose probabilities add roughly to the same amount. The algorithm is a top-down approach and the most significant bit positions for all symbols are determined first. The idea is to partition the symbols in two groups such that their probability sums are approximately as much equal as is possible. The process is then repeated for each sub-partition. Each partition can be imagined as an abstract symbol representing the sum of probabilities of symbols in the partition. The process is iterated until each of the final partitions contains only one original symbol to be encoded. The algorithm is illustrated below for the probability distributions *P*=(0.25, 0.2, 0.15, 0.15.0.1, 0.1, 0.05) and *P*=(0.25, 0.25, 0.125, 0.125, 0.125). 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 codes as we will see later.

i	*pi*					Code				
1	0.25	1	0			1	0			Average length = 2.7 bit/symbol
2	0.2	1	1			1	1			Entropy=2.67bit
3	0.15	0	0	0		0	0	0		very good
4	0.15	0	0	1		0	0	1		
5	0.1	0	1	0		0	1	0		
6	0.1	0	1	1	0	0	1	1	0	
7	0.05	0	1	1	1	0	1	1	1	

1	0.25	1	1		Average length = 2.5 bit/symbol
2	0.25	1	0		Entropy=2.5bit
3	0.125	0	1	1	perfect code!
4	0.125	0	1	0	
5	0.125	0	0	1	
6	0.125	0	0	0	

Shannon-Fano code does not always produce the best expected length codes. For example, take *P*=(0.4, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1). The first partition will group the symbols with probabilities 0.4 and 0.1 in one group and the rest in the second group and so on resulting in a *code length sequence L*=(2,2,3,3,3,4,4). This gives an average code length of 2.70 bits. A Huffman code, as we will see soon, will yield a code length sequence *L*=(1,3,3,4,4,4,4) yielding an average code length of 2.60 bits. There is even a "non-Huffman" prefix code given below with length sequence *L*=(2,3,3,3,3,3,3) whose average code length is 2.60 bits.

1 : 00
2 : 010
3 : 011
4 : 100
5 : 101
6 : 110
7 : 111

The top-down construction of the Shannon-Fano code forces the second symbol to have two bits and some others to have 4 bits, although all symbols from second to the last have the same equal probability 0.1. The last code gives them all the same length. The Huffman code give the first symbol a code length of only one bit which more than compensates for some symbols having code length 4.

Huffman Code:

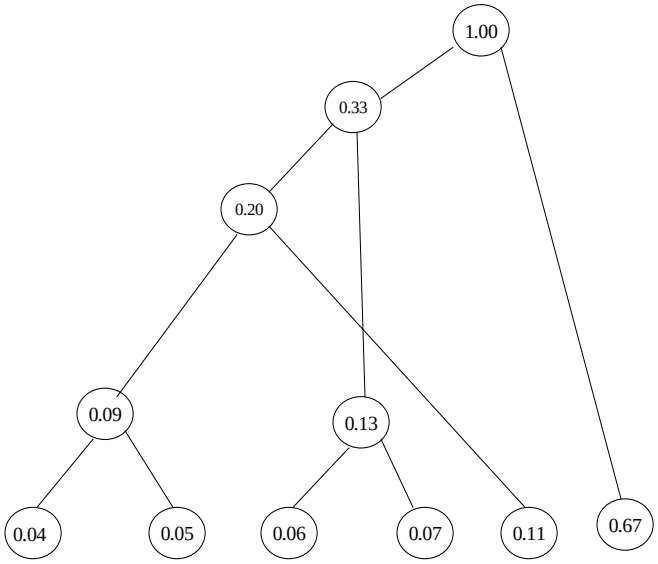
- Shannon-Fano is top-down. If you draw a binary tree, the symbols near to the root get codes assigned to them first.
- Huffman is bottom-up. It starts assigning codes from leaf nodes.

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

<u>Same offer</u>: 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)

Huffman code uses a bottom-up approach. At the beginning, each symbol has code word of length 0. Unless n=1, this violates Kraft inequality and it is not prefix. At each stage of the algorithm, two symbols having lowest probabilities are combined to form a composite symbol whose probability is the sum of probabilities of its constituent symbols, which are recognized as child nodes in the binary tree depicting the construction. The construction is as shown below. A code bit of '0' is prefixed to the already generated code for the symbol to its left child and a code bit '1' is prefixed to the right child. This reduces the Kraft inequality value and also reduces the total number of symbols yet to be coded by one. The probabilities are sorted again and the process is repeated.



The steps are given below with corresponding value of the Kraft inequality values K(C).

P=(0.67, 0.11, 0.07, 0.06, 0.05, 0.04) ; K(C)=6
P=(0.67, 0.11, 0.09, 0.07, 0.06) ; K(C)=5
P=(0.67, 0.13, 0.11, 0.09) ; K(C)=4
P=(0.67, 0.20, 0.13) ; K(C)=3
P=(0.67, 0.33) ; K(C)=2
P=(1.00) ; K(C)=1

When the number of composite symbols becomes exactly one (the root node of the binary tree), the process terminates with K(C) =1.

What is the complexity of the Huffman algorithm? What is its storage complexity ?

Huffman code decoding

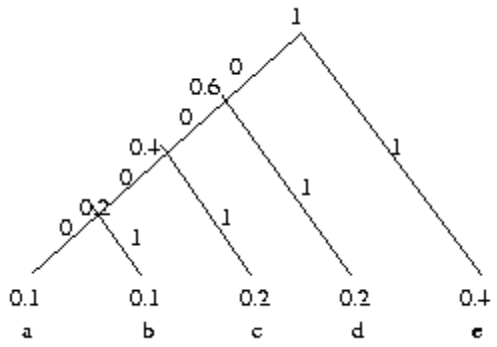
Note there is a unique path from the root to each leaf node each of which represents a source symbol. The internal and the root nodes do not represent the source symbols, they represent abstract composite symbols.

Based on the tree representation of the Huffman code, can you formulate an obvious decoding algorithm? What is the most efficient way to store the tree?

Note the decoder must have an exact same copy of the Huffman tree. This constitutes an overhead which becomes insignificant if the tree (sometimes also refereed to as a table) is used many times over large number of files.

Minimum Variance Code

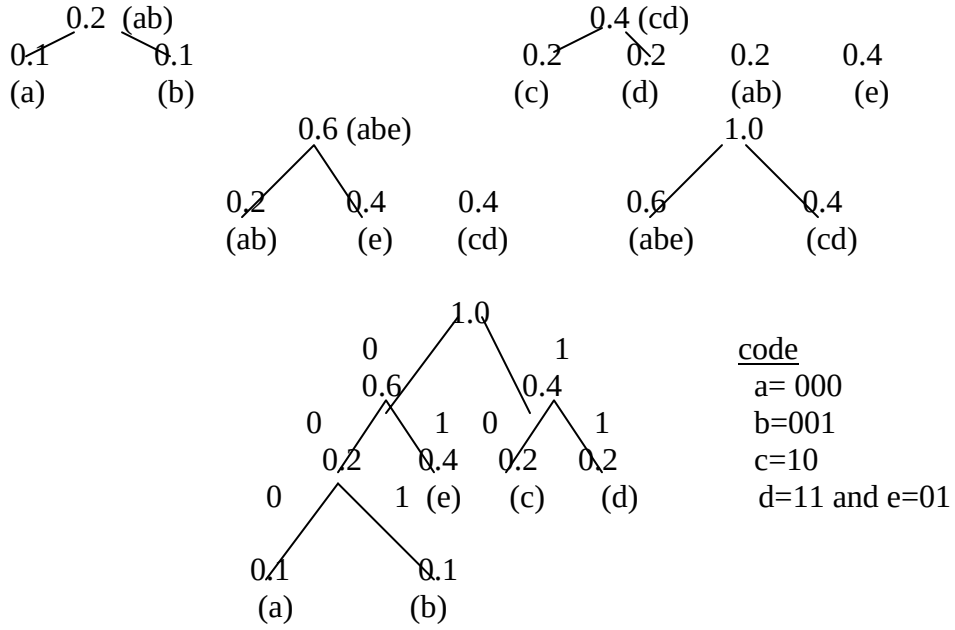
- Average code length $\bar{l} = \sum p_i l_i$. Variance of code $v = \sum_i (l_i - \bar{l})^2 p_i$



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

The codes for the above tree are *a*=0000, *b*=0001, *c*=001, *d*=01 and *e*=1. If we draw the tree as: combine (a,b), then (c,d). Then combone (ab,) with e and then (a,b,e) with (c,d). Then the codes are *a*=000, *b*=001, *c*=10, *d*=11 and *e*=01. You will see the variance is 0.16 although the average length is the same (2.2). The tree looks more **bushy**. Another bushy construction with same variance is shown next page. The more bushy the tree is the less will be its variance value because ,by definition, the mutual difference in length of the codes are smaller for bushy trees.

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.*



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

Optimality of Huffman Code

Theorem 2 *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.*

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

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:

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

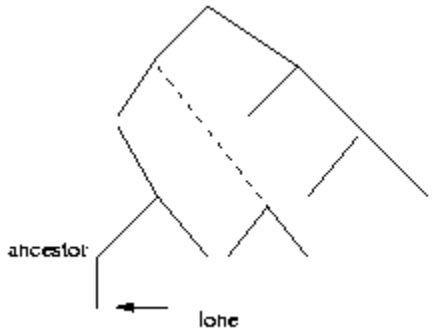
Therefore,

$$\begin{aligned} \bar{l} - \bar{l}^* &= p(a_1)(l_1 - l_2) + p(a_2)(l_2 - l_1) \\ &= p(a_1)(l_1 - l_2) - p(a_2)(l_1 - l_2) \\ &= C[p(a_1) - p(a_2)], \qquad C = l_1 - l_2 \end{aligned}$$

Thus, $\bar{l} > \bar{l}^*$. This means that $\bar{l}$ is not 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 $1 \leq j \leq M-2$ 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 $\{p_j^*, 1 \le j \le M-1\}$ as

$$p_{M-1}^* = p_{M-1} + p_M,$$

$$p_j^* = p_j \qquad 1 \le j \le M-2$$

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 these argument M-1 times establishes that Huffman algorithm produces minimum average length prefix binary codes, which proves Theorem 2.

This argument is also valid for d-ary codes!

Theorem 3 *The entropy H of* $\{p_j, 1 \le j \le n\}$ *satisfies* $0 \le H \le \log n$

Theorem 2 says that Huffman code produces a minimum average length code. Now, we want to show that this average length is bounded below by the entropy of the source *S* denoted as *H(S)* and bounded above by *H(S)*+1 bits. That is $H(S) \le \bar{l} \le H(S) + 1$.

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 $\log x \geq \log_2 e(1 - 1/x)$, 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

The upper bound will be proved by showing that there exists a UD code (prefix free code is UD) with average code word length *H*(*S*) +1. Thus the optimal code must have an average length less than *H*(*S*) +1.

For our model, we know $l_i = -\log p_i$ **.** The integral value of the length is $l_i = \left\lceil \log \frac{1}{p} \right\rceil$**.**

Therefore, $-\log p_i \leq l_i \leq -\log p_i + 1$

Thus, from left inequality, we have $p_i \leq 2^{-l_i}$. Therefore

$$\sum_{i=1}^{K} 2^{-l_i} \leq \sum_{i=1}^{K} p_i = 1$$

By Kraft-McMillan theorem, there exists a UD code with code word lengths {l_i}. Then, we can use the right inequality of $-\log p_i \leq l_i \leq -\log p_i + 1$ to write**:**

$$\bar{l} = \sum p_i l_i \le \sum p_i(-\log p_i + 1) \le -\sum p_i \log p_i + \sum p_i \le H(S) + 1$$

Combining the lower and upper bound, we have (See pp. 46-48, Sayood)

Theorem 4: $H(S) \le \bar{l} < H(S) + 1$

Rather than developing Huffman code for each symbol of the alphabet, if we develop coding for *k* symbols together (the so-called *k*-grams), we can show (Sayood, pp49-52)

$$H(S) \le \bar{l}_k \le H(S) + \frac{1}{k}$$

Thus, when $k \to \infty$, the upper and lower bound collapses to the value of the entropy of the source. This is the reason why Huffman codes are called **optimal** binary codes. But be aware, to achieve optimality, we must have probability values for all grams of arbitrary length which is very impractical. But, even at the word level Huffman's performance is even better than LZ family of codes, as we will see later.

Canonical Huffman Code

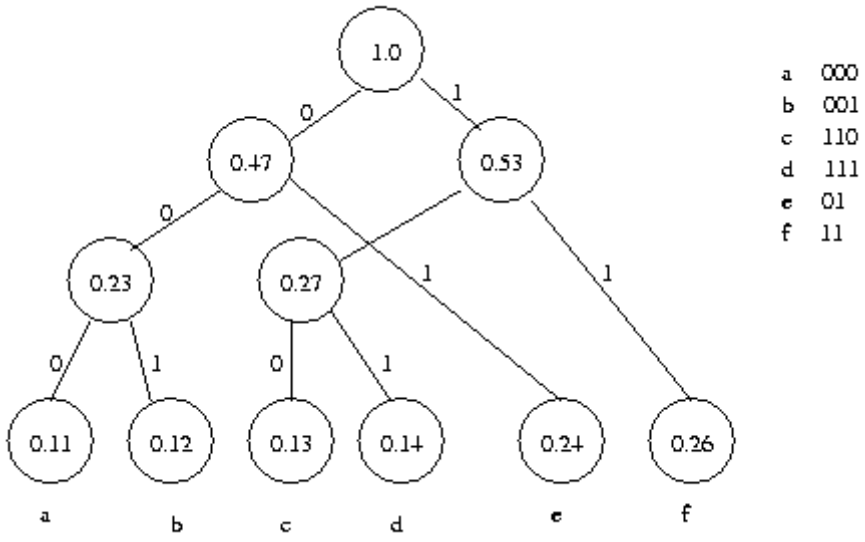
Huffman code has some major disadvantages. If the alphabet size is large, viz. word based Huffman need to code each word of a large English dictionary.

1. Space: *n* symbols leaf nodes; *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, the tree 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.

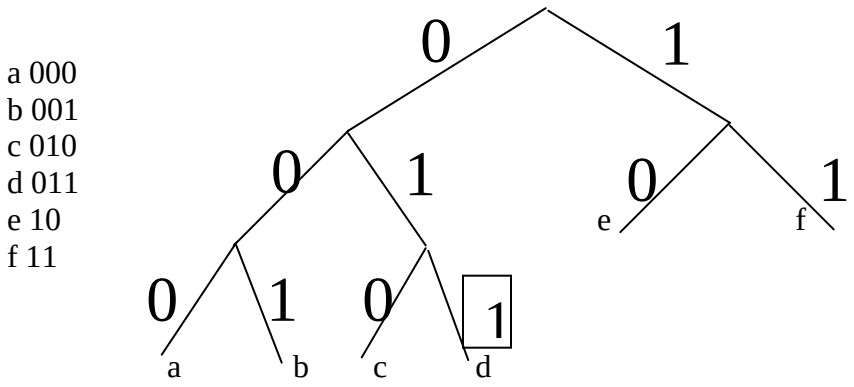
The canonical Huffman code does not need any prelude to be sent receiver. It also needs less storage. Canonical Huffman is very useful when the alphabet is large but fast decoding is necessary. The code is stored in consecutive memory addresses, along with symbols. The encoding and decoding steps are very fast. The design of the code starts with the knowledge of the lengths of the code given as input to the encoder. This step takes additional computation time but can be performed offline.

Non-Huffman Codes Having Same Average Length as That of Huffman Code

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, for this example, **we should have** $2^5 = 32$ **Huffman codes**. But, let us create the codes as 00*x*, 10*y*, 01, and 11 where *x*,*y* = 0 or 1. let *A*=00, *B*=10, *C*=01, *D*=11. The codes are *Ax, By, C, D*. Any permutation of *A, B, C,* D will lead to a valid Huffman code in the sense that code lengths will be the same and all codes will be prefix codes. There are 4! permutations and (*x*,*y*) has 4 possible values – hence a total of 96 codes! This means that there are prefix codes that cannot be generated by Huffman tree but has the same average length as that of the Huffman code. *Canonic Huffman* code is one such "Huffman" code. An example is given below. Note all the codes of same length are consecutive binary integers of given length.



The corresponding binary tree cannot be derived following Huffman's algorithm. But, it is prefix, minimum redundancy and has same average code length as that of

the Huffman code. Given the lengths of the Huffman words, these codes can be generated as follows.

Algorithm to Generate the Canonical Huffman Codes (Encoding)

1. The input to the algorithm is the code length sequence in non-increasing sequence {l_{max}, …., l_k).
2. 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 $l_{\max}$.
3. If the next length is l_{k_2}, extract l_{k_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_{k_2} to get the code for the group.
4. 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 code 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 *first(l)* denote the first code 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 *num*(*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*) := $\lceil (first(l+1) + num(l+1))/2 \rceil$;

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

$$first\,(5) = 0$$
$$first\,(4) = \lceil (0+4)/2 \rceil = 2$$
$$first\,(3) = \lceil (2+0)/2 \rceil = 1$$
$$first\,(2) = \lceil (1+1)/2 \rceil = 1$$
$$first\,(1) = \lceil (1+3)/2 \rceil = 2$$

We have

l	1	2	3	4	5
$num\,(l)$	0	3	1	0	4
$first(l)$	2	**1**	**1**	2	**0**

Only the bold numbers in the array *first(l)* are used. Given the array *first(l)*, the algorithms steps can now be followed to obtain the canonical codes. The expression $\lceil (first(l+1) + num(l+1))/2 \rceil$ guarantees that the resulting code is prefix-free. Convince yourself that the algorithm generates a prefix-free code with specified lengths.

Storing the Code in Memory

We will now give an algorithm to store the resulting code in consecutive locations in main memory, starting from address 0. It is this property that will make the decoding operations very efficient as we will see soon. Since it is a variable length code, provision must be made to detect the end of a code word in each address. The following code gives the address of the first code word in each group.

Compute an array called *first_address(l)*

Begin

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

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

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

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

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

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

} else

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

} *End*

So, the result is: you may verify the addresses for lengths 5, 3, 2 are 0, 4, 5 respectively, in the table (indicated by bold). Note the addresses for lengths 4 and 1 are set to 0 and they represent dummy addresses but are useful in decoding.

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

Decoding Algorithm

Now, we are ready to perform the decoding operation given an input bit string. We define a bit string variable ***v*** which stuffs input bits (to be decoded) into it as long as the binary number represented by ***v*** 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 of codes, so we need to have the off-set address in this group to access the symbol stored in an array in a RAM. Here is the algorithm:

```
while input is not exhausted do {
        l = 1;
        stuff input bit in v; /* preparing the code word, msb first/*
        while v < first(l) do{
                append next input bit to v;
                l = l+1;
                        }
difference = v – first(l); /* computes the offset address within the group./*
output symbol at first_address(l) + difference }
```

Note for each decoded symbol, we only need 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 accesses as opposed to only 6 for canonic Huffman code.

l	1	**2**	**3**	4	**5**
num	0	**3**	**1**	0	**4**
first(l)	2	1	1	2	0
first_address(*l*)	0	**5**	**4**	0	**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			

Try to trace the algorithm and see whether the following bit string gives the correct symbol sequence.
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 tree. Instead, design 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 trees) which will not be presented in these notes.. I would like to assign this as optional reading: from Witten, Moffat and Bell ,pp.41-51 and David Salomon, pp.73-76, Moffat and Turpin, Ch.4.

Non-Binary Huffman Code (See Section 3.3, Sayood)
Adaptive Huffman Code (See Section 3.4, Sayood)

Adaptive Huffman Code

Huffman tree has the following properties

- ***Each node except root node has a sibling.***
- ***If the nodes (excluding root) are listed in order of non-increasing weight,then each node is adjacent to its sibling.***

Procedure: Whenever the count of a node is incremented, the new count is compared with the two counts of the next higher sibling pair (if any) in the ordered list. If the new count becomes larger than any one of these two counts, the two nodes must be interchanged (or the two subtrees must be interchanged) and the new counts computed until no further interchange can take place.

Sibling pairs:(x,f)(y,e)(z,d)(c,j)(a,b)