```
        k=ij[k][ip[(c+2) % 10][7 & m++]];
    }
    for (j=0;j<=9;j++)                    Find which appended digit will check properly.
        if (!ij[k][ip[j][m & 7]]) break;
    *ch=j+48;                             Convert to ASCII.
    return k==0;
}
```

CITED REFERENCES AND FURTHER READING:

McNamara, J.E. 1982, *Technical Aspects of Data Communication*, 2nd ed. (Bedford, MA: Digital Press). [1]

da Cruz, F. 1987, *Kermit, A File Transfer Protocol* (Bedford, MA: Digital Press). [2]

Morse, G. 1986, *Byte*, vol. 11, pp. 115–124 (September). [3]

LeVan, J. 1987, *Byte*, vol. 12, pp. 339–341 (November). [4]

Sarwate, D.V. 1988, *Communications of the ACM*, vol. 31, pp. 1008–1013. [5]

Griffiths, G., and Stones, G.C. 1987, *Communications of the ACM*, vol. 30, pp. 617–620. [6]

Wagner, N.R., and Putter, P.S. 1989, *Communications of the ACM*, vol. 32, pp. 106–110. [7]

20.4 Huffman Coding and Compression of Data

A lossless data compression algorithm takes a string of symbols (typically ASCII characters or bytes) and translates it *reversibly* into another string, one that is *on the average* of shorter length. The words "on the average" are crucial; it is obvious that no reversible algorithm can make all strings shorter — there just aren't enough short strings to be in one-to-one correspondence with longer strings. Compression algorithms are possible only when, on the input side, some strings, or some input symbols, are more common than others. These can then be encoded in fewer bits than rarer input strings or symbols, giving a net average gain.

There exist many, quite different, compression techniques, corresponding to different ways of detecting and using departures from equiprobability in input strings. In this section and the next we shall consider only *variable length codes* with *defined word* inputs. In these, the input is sliced into fixed units, for example ASCII characters, while the corresponding output comes in chunks of variable size. The simplest such method is Huffman coding [1], discussed in this section. Another example, *arithmetic compression*, is discussed in §20.5.

At the opposite extreme from defined-word, variable length codes are schemes that divide up the *input* into units of variable length (words or phrases of English text, for example) and then transmit these, often with a fixed-length output code. The most widely used code of this type is the Ziv-Lempel code [2]. References [3-6] give the flavor of some other compression techniques, with references to the large literature.

The idea behind Huffman coding is simply to use shorter bit patterns for more common characters. We can make this idea quantitative by considering the concept of *entropy*. Suppose the input alphabet has N_{ch} characters, and that these occur in the input string with respective probabilities p_i, $i = 1, \dots, N_{ch}$, so that $\sum p_i = 1$. Then the fundamental theorem of information theory says that strings consisting of independently random sequences of these characters (a conservative, but not always realistic assumption) require, on the average, at least

$$H = -\sum p_i \log_2 p_i \tag{20.4.1}$$

bits per character. Here H is the entropy of the probability distribution. Moreover, coding schemes exist which approach the bound arbitrarily closely. For the case of equiprobable characters, with all $p_i = 1/N_{ch}$, one easily sees that $H = \log_2 N_{ch}$, which is the case of no compression at all. Any other set of p_i's gives a smaller entropy, allowing some useful compression.

Notice that the bound of (20.4.1) would be achieved if we could encode character i with a code of length $L_i = -\log_2 p_i$ bits: Equation (20.4.1) would then be the average $\sum p_i L_i$. The trouble with such a scheme is that $-\log_2 p_i$ is not generally an integer. How can we encode the letter "Q" in 5.32 bits? Huffman coding makes a stab at this by, in effect, approximating all the probabilities p_i by integer powers of 1/2, so that all the L_i's are integral. If all the p_i's are in fact of this form, then a Huffman code does achieve the entropy bound H.

The construction of a Huffman code is best illustrated by example. Imagine a language, Vowellish, with the $N_{ch} = 5$ character alphabet A, E, I, O, and U, occurring with the respective probabilities 0.12, 0.42, 0.09, 0.30, and 0.07. Then the construction of a Huffman code for Vowellish is accomplished in the following table:

Node	Stage:	1	2	3	4	5
1	A:	0.12	0.12 ∎			
2	E:	0.42	0.42	0.42	0.42 ∎	
3	I:	0.09 ∎				
4	O:	0.30	0.30	0.30 ∎		
5	U:	0.07 ∎				
6		UI:	0.16 ∎			
7			AUI:	0.28 ∎		
8				AUIO:	0.58 ∎	
9					EAUIO:	1.00

Here is how it works, proceeding in sequence through N_{ch} stages, represented by the columns of the table. The first stage starts with N_{ch} nodes, one for each letter of the alphabet, containing their respective relative frequencies. At each stage, the two smallest probabilities are found, summed to make a new node, and then dropped from the list of active nodes. (A "block" denotes the stage where a node is dropped.) All active nodes (including the new composite) are then carried over to the next stage (column). In the table, the names assigned to new nodes (e.g., AUI) are inconsequential. In the example shown, it happens that (after stage 1) the two

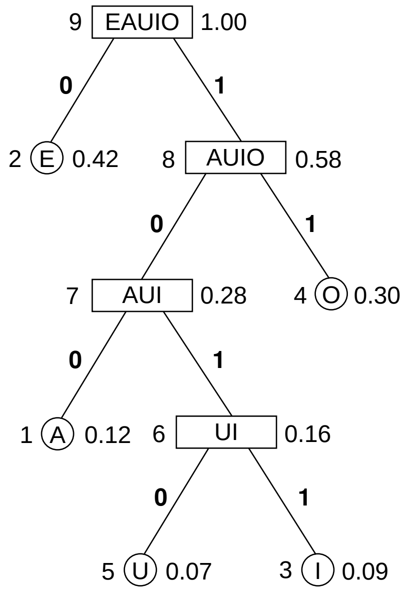

Figure 20.4.1. Huffman code for the fictitious language Vowellish, in tree form. A letter (A, E, I, O, or U) is encoded or decoded by traversing the tree from the top down; the code is the sequence of 0's and 1's on the branches. The value to the right of each node is its probability; to the left, its node number in the accompanying table.

smallest nodes are always an original node and a composite one; this need not be true in general: The two smallest probabilities might be both original nodes, or both composites, or one of each. At the last stage, all nodes will have been collected into one grand composite of total probability 1.

Now, to see the code, you redraw the data in the above table as a tree (Figure 20.4.1). As shown, each node of the tree corresponds to a node (row) in the table, indicated by the integer to its left and probability value to its right. Terminal nodes, so called, are shown as circles; these are single alphabetic characters. The branches of the tree are labeled 0 and 1. The code for a character is the sequence of zeros and ones that lead to it, from the top down. For example, E is simply 0, while U is 1010.

Any string of zeros and ones can now be decoded into an alphabetic sequence. Consider, for example, the string 1011111010. Starting at the top of the tree we descend through 1011 to I, the first character. Since we have reached a terminal node, we reset to the top of the tree, next descending through 11 to O. Finally 1010 gives U. The string thus decodes to IOU.

These ideas are embodied in the following routines. Input to the first routine `hufmak` is an integer vector of the frequency of occurrence of the `nchin` $\equiv N_{ch}$ alphabetic characters, i.e., a set of integers proportional to the p_i's. `hufmak`, along with `hufapp`, which it calls, performs the construction of the above table, and also the tree of Figure 20.4.1. The routine utilizes a heap structure (see §8.3) for efficiency; for a detailed description, see Sedgewick [7].

```
#include "nrutil.h"

typedef struct {
    unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;

void hufmak(unsigned long nfreq[], unsigned long nchin, unsigned long *ilong,
    unsigned long *nlong, huffcode *hcode)
```
Given the frequency of occurrence table `nfreq[1..nchin]` of `nchin` characters, construct the Huffman code in the structure `hcode`. Returned values `ilong` and `nlong` are the character number that produced the longest code symbol, and the length of that symbol. You should check that `nlong` is not larger than your machine's word length.
```
{
    void hufapp(unsigned long index[], unsigned long nprob[], unsigned long n,
        unsigned long i);
    int ibit;
    long node,*up;
    unsigned long j,k,*index,n,nused,*nprob;
    static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
        0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
        0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
        0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
        0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

    hcode->nch=nchin;                                  Initialization.
    index=lvector(1,(long)(2*hcode->nch-1));
    up=(long *)lvector(1,(long)(2*hcode->nch-1));     Vector that will keep track of
    nprob=lvector(1,(long)(2*hcode->nch-1));              heap.
    for (nused=0,j=1;j<=hcode->nch;j++) {
        nprob[j]=nfreq[j];
        hcode->icod[j]=hcode->ncod[j]=0;
        if (nfreq[j]) index[++nused]=j;
    }
    for (j=nused;j>=1;j--) hufapp(index,nprob,nused,j);
    Sort nprob into a heap structure in index.
    k=hcode->nch;
    while (nused > 1) {                                Combine heap nodes, remaking
        node=index[1];                                     the heap at each stage.
        index[1]=index[nused--];
        hufapp(index,nprob,nused,1);
        nprob[++k]=nprob[index[1]]+nprob[node];
        hcode->left[k]=node;                           Store left and right children of a
        hcode->right[k]=index[1];                          node.
        up[index[1]] = -(long)k;                       Indicate whether a node is a left
        up[node]=index[1]=k;                               or right child of its parent.
        hufapp(index,nprob,nused,1);
    }
    up[hcode->nodemax=k]=0;
    for (j=1;j<=hcode->nch;j++) {                      Make the Huffman code from
        if (nprob[j]) {                                    the tree.
            for (n=0,ibit=0,node=up[j];node;node=up[node],ibit++) {
                if (node < 0) {
                    n |= setbit[ibit];
                    node = -node;
                }
            }
            hcode->icod[j]=n;
            hcode->ncod[j]=ibit;
        }
    }
    *nlong=0;
    for (j=1;j<=hcode->nch;j++) {
        if (hcode->ncod[j] > *nlong) {
            *nlong=hcode->ncod[j];
```

```
            *ilong=j-1;
        }
    }
    free_lvector(nprob,1,(long)(2*hcode->nch-1));
    free_lvector((unsigned long *)up,1,(long)(2*hcode->nch-1));
    free_lvector(index,1,(long)(2*hcode->nch-1));
}
```

```
void hufapp(unsigned long index[], unsigned long nprob[], unsigned long n,
    unsigned long i)
```
Used by `hufmak` to maintain a heap structure in the array `index[1..l]`.
```
{
    unsigned long j,k;

    k=index[i];
    while (i <= (n>>1)) {
        if ((j = i << 1) < n && nprob[index[j]] > nprob[index[j+1]]) j++;
        if (nprob[k] <= nprob[index[j]]) break;
        index[i]=index[j];
        i=j;
    }
    index[i]=k;
}
```

Note that the structure `hcode` must be defined and allocated in your main program with statements like this:

```
#include "nrutil.h"
#define MC 512                     Maximum anticipated value of nchin in hufmak.
#define MQ (2*MC-1)
typedef struct {
    unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;
    ...
    huffcode hcode;
    ...
    hcode.icod=(unsigned long *)lvector(1,MQ);       Allocate space within hcode.
    hcode.ncod=(unsigned long *)lvector(1,MQ);
    hcode.left=(unsigned long *)lvector(1,MQ);
    hcode.right=(unsigned long *)lvector(1,MQ);
    for (j=1;j<=MQ;j++) hcode.icod[j]=hcode.ncod[j]=0;
```

Once the code is constructed, one encodes a string of characters by repeated calls to `hufenc`, which simply does a table lookup of the code and appends it to the output message.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;

void hufenc(unsigned long ich, unsigned char **codep, unsigned long *lcode,
    unsigned long *nb, huffcode *hcode)
```
Huffman encode the single character `ich` (in the range `0..nch-1`) using the code in the structure `hcode`, write the result to the character array `*codep[1..lcode]` starting at bit `nb` (whose smallest valid value is zero), and increment `nb` appropriately. This routine is called

repeatedly to encode consecutive characters in a message, but must be preceded by a single initializing call to hufmak, which constructs hcode.

```
{
    void nrerror(char error_text[]);
    int l,n;
    unsigned long k,nc;
    static unsigned long setbit[32]={0x1L,0x2L,0x4L,0x8L,0x10L,0x20L,
        0x40L,0x80L,0x100L,0x200L,0x400L,0x800L,0x1000L,0x2000L,
        0x4000L,0x8000L,0x10000L,0x20000L,0x40000L,0x80000L,0x100000L,
        0x200000L,0x400000L,0x800000L,0x1000000L,0x2000000L,0x4000000L,
        0x8000000L,0x10000000L,0x20000000L,0x40000000L,0x80000000L};

    k=ich+1;
    Convert character range 0..nch-1 to array index range 1..nch.
    if (k > hcode->nch || k < 1) nrerror("ich out of range in hufenc.");
    for (n=hcode->ncod[k]-1;n>=0;n--,++(*nb)) {       Loop over the bits in the stored
        nc=(*nb >> 3);                                     Huffman code for ich.
        if (++nc >= *lcode) {
            fprintf(stderr,"Reached the end of the 'code' array.\n");
            fprintf(stderr,"Attempting to expand its size.\n");
            *lcode *= 1.5;
            if ((*codep=(unsigned char *)realloc(*codep,
                (unsigned)(*lcode*sizeof(unsigned char)))) == NULL) {
                nrerror("Size expansion failed.");
            }
        }
        l=(*nb) & 7;
        if (!l) (*codep)[nc]=0;                        Set appropriate bits in code.
        if (hcode->icod[k] & setbit[n]) (*codep)[nc] |= setbit[l];
    }
}
```

Decoding a Huffman-encoded message is slightly more complicated. The coding tree must be traversed from the top down, using up a variable number of bits:

```
typedef struct {
    unsigned long *icod,*ncod,*left,*right,nch,nodemax;
} huffcode;

void hufdec(unsigned long *ich, unsigned char *code, unsigned long lcode,
    unsigned long *nb, huffcode *hcode)
```

Starting at bit number nb in the character array code[1..lcode], use the Huffman code stored in the structure hcode to decode a single character (returned as ich in the range 0..nch-1) and increment nb appropriately. Repeated calls, starting with nb $= 0$ will return successive characters in a compressed message. The returned value ich=nch indicates end-of-message. The structure hcode must already have been defined and allocated in your main program, and also filled by a call to hufmak.

```
{
    long nc,node;
    static unsigned char setbit[8]={0x1,0x2,0x4,0x8,0x10,0x20,0x40,0x80};

    node=hcode->nodemax;
    for (;;) {                      Set node to the top of the decoding tree, and loop
        nc=(*nb >> 3);                  until a valid character is obtained.
        if (++nc > lcode) {         Ran out of input; with ich=nch indicating end of
            *ich=hcode->nch;            message.
            return;
        }
        node=(code[nc] & setbit[7 & (*nb)++] ?
            hcode->right[node] : hcode->left[node]);
            Branch left or right in tree, depending on its value.
        if (node <= hcode->nch) {   If we reach a terminal node, we have a complete
                                        character and can return.
```

```
            *ich=node-1;
            return;
        }
    }
}
```

For simplicity, hufdec quits when it runs out of code bytes; if your coded message is not an integral number of bytes, and if N_{ch} is less than 256, hufdec can return a spurious final character or two, decoded from the spurious trailing bits in your last code byte. If you have independent knowledge of the number of characters sent, you can readily discard these. Otherwise, you can fix this behavior by providing a bit, not byte, count, and modifying the routine accordingly. (When N_{ch} is 256 or larger, hufdec will normally run out of code in the middle of a spurious character, and it will be discarded.)

Run-Length Encoding

For the compression of highly correlated bit-streams (for example the black or white values along a facsimile scan line), Huffman compression is often combined with *run-length encoding*: Instead of sending each bit, the input stream is converted to a series of integers indicating how many consecutive bits have the same value. These integers are then Huffman-compressed. The Group 3 CCITT facsimile standard functions in this manner, with a fixed, immutable, Huffman code, optimized for a set of eight standard documents [8,9].

CITED REFERENCES AND FURTHER READING:

Gallager, R.G. 1968, *Information Theory and Reliable Communication* (New York: Wiley).

Hamming, R.W. 1980, *Coding and Information Theory* (Englewood Cliffs, NJ: Prentice-Hall).

Storer, J.A. 1988, *Data Compression: Methods and Theory* (Rockville, MD: Computer Science Press).

Nelson, M. 1991, *The Data Compression Book* (Redwood City, CA: M&T Books).

Huffman, D.A. 1952, *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101. [1]

Ziv, J., and Lempel, A. 1978, *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530–536. [2]

Cleary, J.G., and Witten, I.H. 1984, *IEEE Transactions on Communications*, vol. COM-32, pp. 396–402. [3]

Welch, T.A. 1984, *Computer*, vol. 17, no. 6, pp. 8–19. [4]

Bentley, J.L., Sleator, D.D., Tarjan, R.E., and Wei, V.K. 1986, *Communications of the ACM*, vol. 29, pp. 320–330. [5]

Jones, D.W. 1988, *Communications of the ACM*, vol. 31, pp. 996–1007. [6]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 22. [7]

Hunter, R., and Robinson, A.H. 1980, *Proceedings of the IEEE*, vol. 68, pp. 854–867. [8]

Marking, M.P. 1990, *The C Users' Journal*, vol. 8, no. 6, pp. 45–54. [9]

20.5 Arithmetic Coding

We saw in the previous section that a perfect (entropy-bounded) coding scheme would use $L_i = -\log_2 p_i$ bits to encode character i (in the range $1 \le i \le N_{ch}$), if p_i is its probability of occurrence. Huffman coding gives a way of rounding the L_i's to close integer values and constructing a code with those lengths. *Arithmetic coding* [1], which we now discuss, actually does manage to encode characters using noninteger numbers of bits! It also provides a convenient way to output the result not as a stream of bits, but as a stream of symbols in any desired radix. This latter property is particularly useful if you want, e.g., to convert data from bytes (radix 256) to printable ASCII characters (radix 94), or to case-independent alphanumeric sequences containing only A-Z and 0-9 (radix 36).

In arithmetic coding, an input message of any length is represented as a real number R in the range $0 \le R < 1$. The longer the message, the more precision required of R. This is best illustrated by an example, so let us return to the fictitious language, Vowellish, of the previous section. Recall that Vowellish has a 5 character alphabet (A, E, I, O, U), with occurrence probabilities 0.12, 0.42, 0.09, 0.30, and 0.07, respectively. Figure 20.5.1 shows how a message beginning "IOU" is encoded: The interval $[0, 1)$ is divided into segments corresponding to the 5 alphabetical characters; the length of a segment is the corresponding p_i. We see that the first message character, "I", narrows the range of R to $0.37 \le R < 0.46$. This interval is now subdivided into five subintervals, again with lengths proportional to the p_i's. The second message character, "O", narrows the range of R to $0.3763 \le R < 0.4033$. The "U" character further narrows the range to $0.37630 \le R < 0.37819$. *Any* value of R in this range can be sent as encoding "IOU". In particular, the binary fraction $.011000001$ is in this range, so "IOU" can be sent in 9 bits. (Huffman coding took 10 bits for this example, see §20.4.)

Of course there is the problem of knowing when to stop decoding. The fraction $.011000001$ represents not simply "IOU," but "IOU...," where the ellipses represent an infinite string of successor characters. To resolve this ambiguity, arithmetic coding generally assumes the existence of a special $N_{ch} + 1$th character, EOM (end of message), which occurs only once at the end of the input. Since EOM has a low probability of occurrence, it gets allocated only a very tiny piece of the number line.

In the above example, we gave R as a binary fraction. We could just as well have output it in any other radix, e.g., base 94 or base 36, whatever is convenient for the anticipated storage or communication channel.

You might wonder how one deals with the seemingly incredible precision required of R for a long message. The answer is that R is never actually represented all at once. At any give stage we have upper and lower bounds for R represented as a finite number of digits in the output radix. As digits of the upper and lower bounds become identical, we can left-shift them away and bring in new digits at the low-significance end. The routines below have a parameter `NWK` for the number of working digits to keep around. This must be large enough to make the chance of an accidental degeneracy vanishingly small. (The routines signal if a degeneracy ever occurs.) Since the process of discarding old digits and bringing in new ones is performed identically on encoding and decoding, everything stays synchronized.