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.

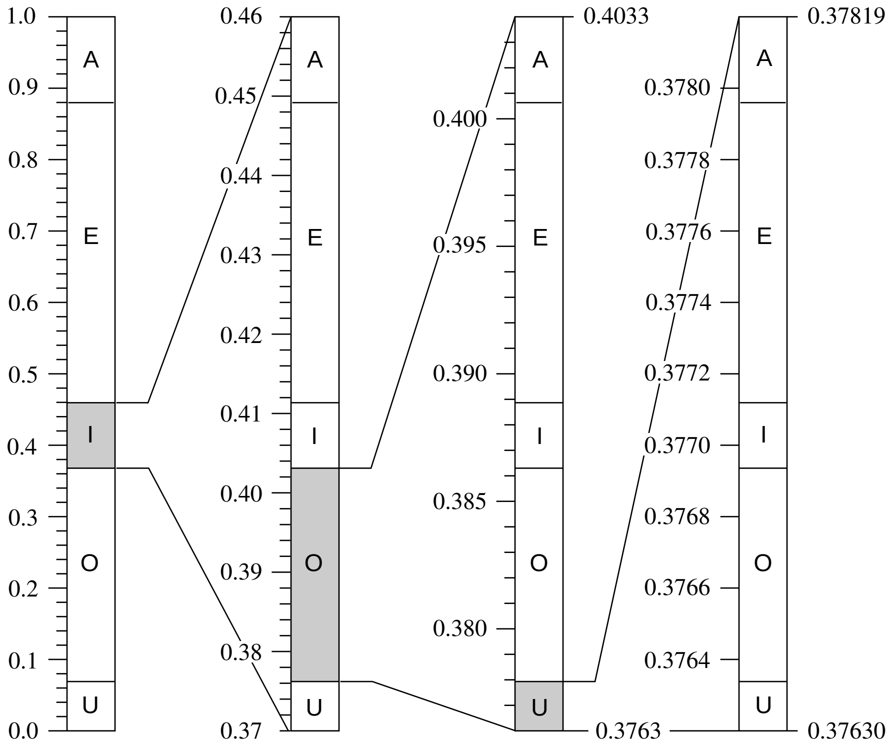

Figure 20.5.1. Arithmetic coding of the message "IOU..." in the fictitious language Vowellish. Successive characters give successively finer subdivisions of the initial interval between 0 and 1. The final value can be output as the digits of a fraction in any desired radix. Note how the subinterval allocated to a character is proportional to its probability of occurrence.

The routine `arcmak` constructs the cumulative frequency distribution table used to partition the interval at each stage. In the principal routine `arcode`, when an interval of size `jdif` is to be partitioned in the proportions of some `n` to some `ntot`, say, then we must compute `(n*jdif)/ntot`. With integer arithmetic, the numerator is likely to overflow; and, unfortunately, an expression like `jdif/(ntot/n)` is not equivalent. In the implementation below, we resort to double precision floating arithmetic for this calculation. Not only is this inefficient, but different roundoff errors can (albeit very rarely) make different machines encode differently, though any one type of machine will decode exactly what it encoded, since identical roundoff errors occur in the two processes. For serious use, one needs to replace this floating calculation with an integer computation in a double register (not available to the `C` programmer).

The internally set variable `minint`, which is the minimum allowed number of discrete steps between the upper and lower bounds, determines when new low-significance digits are added. `minint` must be large enough to provide resolution of all the input characters. That is, we must have $p_i \times \texttt{minint} > 1$ for all i. A value of $100N_{ch}$, or $1.1/\min p_i$, whichever is larger, is generally adequate. However, for safety, the routine below takes `minint` to be as large as possible, with the product `minint*nradd` just smaller than overflow. This results in some time inefficiency, and in a few unnecessary characters being output at the end of a message. You can decrease `minint` if you want to live closer to the edge.

A final safety feature in `arcmak` is its refusal to believe zero values in the table `nfreq`; a 0 is treated as if it were a 1. If this were not done, the occurrence in a message of a single character whose `nfreq` entry is zero would result in scrambling the entire rest of the message. If you want to live dangerously, with a very slightly more efficient coding, you can delete the `IMAX( ,1)` operation.

```
#include "nrutil.h"
#include <limits.h>                    ANSI header file containing integer ranges.
#define MC 512
#ifdef ULONG_MAX                       Maximum value of unsigned long.
#define MAXINT (ULONG_MAX >> 1)
#else
#define MAXINT 2147483647
#endif
```
Here MC is the largest anticipated value of nchh; MAXINT is a large positive integer that does not overflow.

```
typedef struct {
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;

void arcmak(unsigned long nfreq[], unsigned long nchh, unsigned long nradd,
    arithcode *acode)
```
Given a table nfreq[1..nchh] of the frequency of occurrence of nchh symbols, and given a desired output radix nradd, initialize the cumulative frequency table and other variables for arithmetic compression in the structure acode.
```
{
    unsigned long j;

    if (nchh > MC) nrerror("input radix may not exceed MC in arcmak.");
    if (nradd > 256) nrerror("output radix may not exceed 256 in arcmak.");

    acode->minint=MAXINT/nradd;
    acode->nch=nchh;
    acode->nrad=nradd;
    acode->ncumfq[1]=0;
    for (j=2;j<=acode->nch+1;j++)
        acode->ncumfq[j]=acode->ncumfq[j-1]+IMAX(nfreq[j-1],1);
    acode->ncum=acode->ncumfq[acode->nch+2]=acode->ncumfq[acode->nch+1]+1;
}
```

The structure `acode` must be defined and allocated in your main program with statements like this:

```
#include "nrutil.h"
#define MC 512                     Maximum anticipated value of nchh in arcmak.
#define NWK 20                     Keep this value the same as in arcode, below.
typedef struct {
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;
    ...
    arithcode acode;
    ...
    acode.ilob=(unsigned long *)lvector(1,NWK);      Allocate space within acode.
    acode.iupb=(unsigned long *)lvector(1,NWK);
    acode.ncumfq=(unsigned long *)lvector(1,MC+2);
```

Individual characters in a message are coded or decoded by the routine `arcode`, which in turn uses the utility `arcsum`.

```
#include <stdio.h>
#include <stdlib.h>
#define NWK 20
#define JTRY(j,k,m) ((long)((((double)(k))*((double)(j)))/((double)(m))))
```
This macro is used to calculate `(k*j)/m` without overflow. Program efficiency can be improved by substituting an assembly language routine that does integer multiply to a double register.

```
typedef struct {
    unsigned long *ilob,*iupb,*ncumfq,jdif,nc,minint,nch,ncum,nrad;
} arithcode;

void arcode(unsigned long *ich, unsigned char **codep, unsigned long *lcode,
    unsigned long *lcd, int isign, arithcode *acode)
```
Compress (`isign` = 1) or decompress (`isign` = −1) the single character `ich` into or out of the character array `*codep[1..lcode]`, starting with byte `*codep[lcd]` and (if necessary) incrementing `lcd` so that, on return, `lcd` points to the first unused byte in `*codep`. Note that the structure `acode` contains both information on the code, and also state information on the particular output being written into the array `*codep`. An initializing call with `isign=0` is required before beginning any `*codep` array, whether for encoding or decoding. This is in addition to the initializing call to `arcmak` that is required to initialize the code itself. A call with `ich=nch` (as set in `arcmak`) has the reserved meaning "end of message."

```
{
    void arcsum(unsigned long iin[], unsigned long iout[], unsigned long ja,
        int nwk, unsigned long nrad, unsigned long nc);
    void nrerror(char error_text[]);
    int j,k;
    unsigned long ihi,ja,jh,jl,m;

    if (!isign) {                   Initialize enough digits of the upper and lower bounds.
        acode->jdif=acode->nrad-1;
        for (j=NWK;j>=1;j--) {
            acode->iupb[j]=acode->nrad-1;
            acode->ilob[j]=0;
            acode->nc=j;
            if (acode->jdif > acode->minint) return;     Initialization complete.
            acode->jdif=(acode->jdif+1)*acode->nrad-1;
        }
        nrerror("NWK too small in arcode.");
    } else {
        if (isign > 0) {            If encoding, check for valid input character.
            if (*ich > acode->nch) nrerror("bad ich in arcode.");
        }
        else {                      If decoding, locate the character ich by bisection.
            ja=(*codep)[*lcd]-acode->ilob[acode->nc];
            for (j=acode->nc+1;j<=NWK;j++) {
                ja *= acode->nrad;
                ja += ((*codep)[*lcd+j-acode->nc]-acode->ilob[j]);
            }
            ihi=acode->nch+1;
            *ich=0;
            while (ihi-(*ich) > 1) {
                m=(*ich+ihi)>>1;
                if (ja >= JTRY(acode->jdif,acode->ncumfq[m+1],acode->ncum))
                    *ich=m;
                else ihi=m;
            }
            if (*ich == acode->nch) return;        Detected end of message.
        }
        Following code is common for encoding and decoding. Convert character ich to a new
        subrange [ilob,iupb).
```

```
        jh=JTRY(acode->jdif,acode->ncumfq[*ich+2],acode->ncum);
        jl=JTRY(acode->jdif,acode->ncumfq[*ich+1],acode->ncum);
        acode->jdif=jh-jl;
        arcsum(acode->ilob,acode->iupb,jh,NWK,acode->nrad,acode->nc);
        arcsum(acode->ilob,acode->ilob,jl,NWK,acode->nrad,acode->nc);
        How many leading digits to output (if encoding) or skip over?
        for (j=acode->nc;j<=NWK;j++) {
            if (*ich != acode->nch && acode->iupb[j] != acode->ilob[j]) break;
            if (*lcd > *lcode) {
                fprintf(stderr,"Reached the end of the 'code' array.\n");
                fprintf(stderr,"Attempting to expand its size.\n");
                *lcode += *lcode/2;
                if ((*codep=(unsigned char *)realloc(*codep,
                    (unsigned)(*lcode*sizeof(unsigned char)))) == NULL) {
                    nrerror("Size expansion failed");
                }
            }
            if (isign > 0) (*codep)[*lcd]=(unsigned char)acode->ilob[j];
            ++(*lcd);
        }
        if (j > NWK) return;      Ran out of message. Did someone forget to encode a
        acode->nc=j;                  terminating ncd?
        for(j=0;acode->jdif<acode->minint;j++)        How many digits to shift?
            acode->jdif *= acode->nrad;
        if (acode->nc-j < 1) nrerror("NWK too small in arcode.");
        if (j) {                  Shift them.
            for (k=acode->nc;k<=NWK;k++) {
                acode->iupb[k-j]=acode->iupb[k];
                acode->ilob[k-j]=acode->ilob[k];
            }
        }
        acode->nc -= j;
        for (k=NWK-j+1;k<=NWK;k++) acode->iupb[k]=acode->ilob[k]=0;
    }
    return;                       Normal return.
}
```

```
void arcsum(unsigned long iin[], unsigned long iout[], unsigned long ja,
    int nwk, unsigned long nrad, unsigned long nc)
```
Used by `arcode`. Add the integer `ja` to the radix `nrad` multiple-precision integer `iin[nc..nwk]`. Return the result in `iout[nc..nwk]`.
```
{
    int j,karry=0;
    unsigned long jtmp;

    for (j=nwk;j>nc;j--) {
        jtmp=ja;
        ja /= nrad;
        iout[j]=iin[j]+(jtmp-ja*nrad)+karry;
        if (iout[j] >= nrad) {
            iout[j] -= nrad;
            karry=1;
        } else karry=0;
    }
    iout[nc]=iin[nc]+ja+karry;
}
```

If radix-changing, rather than compression, is your primary aim (for example to convert an arbitrary file into printable characters) then you are of course free to set all the components of `nfreq` equal, say, to 1.

CITED REFERENCES AND FURTHER READING:

Bell, T.C., Cleary, J.G., and Witten, I.H. 1990, *Text Compression* (Englewood Cliffs, NJ: Prentice-Hall).

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

Witten, I.H., Neal, R.M., and Cleary, J.G. 1987, *Communications of the ACM*, vol. 30, pp. 520–540. [1]

20.6 Arithmetic at Arbitrary Precision

Let's compute the number π to a couple of thousand decimal places. In doing so, we'll learn some things about multiple precision arithmetic on computers and meet quite an unusual application of the fast Fourier transform (FFT). We'll also develop a set of routines that you can use for other calculations at any desired level of arithmetic precision.

To start with, we need an analytic algorithm for π. Useful algorithms are quadratically convergent, i.e., they double the number of significant digits at each iteration. Quadratically convergent algorithms for π are based on the *AGM (arithmetic geometric mean)* method, which also finds application to the calculation of elliptic integrals (cf. §6.11) and in advanced implementations of the ADI method for elliptic partial differential equations (§19.5). Borwein and Borwein [1] treat this subject, which is beyond our scope here. One of their algorithms for π starts with the initializations

$$\begin{aligned} X_0 &= \sqrt{2} \\ \pi_0 &= 2 + \sqrt{2} \\ Y_0 &= \sqrt[4]{2} \end{aligned} \tag{20.6.1}$$

and then, for $i = 0, 1, \dots$, repeats the iteration

$$\begin{aligned} X_{i+1} &= \frac{1}{2}\left(\sqrt{X_i} + \frac{1}{\sqrt{X_i}}\right) \\ \pi_{i+1} &= \pi_i \left(\frac{X_{i+1} + 1}{Y_i + 1}\right) \\ Y_{i+1} &= \frac{Y_i\sqrt{X_{i+1}} + \dfrac{1}{\sqrt{X_{i+1}}}}{Y_i + 1} \end{aligned} \tag{20.6.2}$$

The value π emerges as the limit π_∞.

Now, to the question of how to do arithmetic to arbitrary precision: In a high-level language like C, a natural choice is to work in radix (base) 256, so that character arrays can be directly interpreted as strings of digits. At the very end of our calculation, we will want to convert our answer to radix 10, but that is essentially a frill for the benefit of human ears, accustomed to the familiar chant, "three point