最新のデータ圧縮理論による

ファイル圧縮

はじめに

最近のハードおよびソフトの発達に伴い，パーソナルコンピュータは，科学計算や事務処理ばかりでなく，ワープロやデータベース，CAD，CAM，ゲームなどいろいろな分野で用いられている。このようにパーソナルコンピュータの汎用性が強まるにつれて，各種言語のソフトや，作成したワープロ文書，データ，さらにはバックアップコピーなどに多数のフロッピーディスクが必要となる。技術の進歩により3.5インチフロッピーで1Mバイト記憶できるなどフロッピーディスクの高密度化が進んでいるが，まだ容量が不足だと感じ，「効率よくプログラムやデータを記憶させる方法はないものだろうか」と思っている人も多いであろう。

本稿では，このような要望に答えるものとして，データ圧縮技術の簡単な紹介を行い，最新の理論に基づくデータ圧縮用プログラムを紹介する。本プログラムはC言語で書かれており，Lattice-CまたはOptimizing-Cでコンパイルすることができる。本プログラムを用いれば，ソースプログラム，オブジェクトプログラム，文書ファイル，各種データなど，ディスク内のあらゆる種類のファイルを効率よく圧縮することができる。また，圧縮されたファイルを復号プログラムで復号することにより，1ビットの誤りもなく元のファイルを復元できる。圧縮効率は，ファイルの種類により異なるが30～70%程度に圧縮でき，多くの場合は60%以下となる。

図1　モールス符号

文字	符号語	文字	符号語
E	・	M	― ―
T	―	U	・・―
A	・―	G	― ― ・
O	― ― ―	Y	― ・ ― ―
N	― ・	P	・ ― ― ・
R	・ ― ・	W	・ ― ―
I	・・	B	― ・・・
S	・・・	V	・・・―
H	・・・・	K	― ・ ―
D	― ・・	X	― ・・―
L	・ ― ・・	J	・ ― ― ―
F	・・ ― ・	Q	― ― ・ ―
C	― ・ ― ・	Z	― ― ・・

データ圧縮とは

データ圧縮は，「情報源符号化」とも呼ばれ，文章や数値データなどの情報に含まれている冗長な部分をとり除き，より短い記号系列(符号語)に変換(符号化)することである。多くの場合，符号語から逆変換（復号）により，元のデータが1ビットの誤りもなく完全に復元できることが望まれる。このような場合を無雑音データ圧縮という。しかし，場合によっては，ある程度の復号誤りが生じてもかまわないから，圧縮効率

図2　2文字単位の符号化(ハフマン符号)

データ	生起確率	符号語	符号語長	符号語長×生起確率
a a	0.64	0	1	0.64
a b	0.16	11	2	0.32
b a	0.16	100	3	0.48
b b	0.04	101	3	0.12
		平均符号語長		1.56/2=0.78

プログラム

山本博資
中田和宏

をよくしたい場合がある。このような場合をゆがみを伴うデータ圧縮という。例えばアナログ値の1/3を有限な桁数の0.333などで近似するのも一種のゆがみを伴ったデータ圧縮となる。ゆがみを伴ったデータ圧縮は「レイト・ディストーション(効率とゆがみ)理論」により詳しく解析がなされているが，ここでは省略し，無雑音データ圧縮について解説する。

さて，データの冗長部分を取り除く方法を考える。文章データやプログラムなどの場合には，その文法や規則を利用して冗長部分を省略することができる。例えば，①“I am a boy.”を②“I a boy”や③“Iamaboy”に縮めても，英語の文法を知っている人は②や③の文から①を想像できる。しかし，②からは“I was a boy.”も可能であるし，①がテストの答案のような場合には，“I is a boy.”や“Iam a boy.”のように文法的に誤った文も可能性がある。したがって，このような方法ではあらゆる文章に対する無雑音データ圧縮は困難である。

そこで一般によく用いられる方法は，文字や記号の出現頻度(生起確率)を利用する圧縮法である。つまり生起確率の高い文字には長さの短い符号語を割り振り，生起確率の低いものには長い符号語を割り振る。英語では，E, T, A, O, N, ……の順に生起確率が低くなる。したがってこの順に長さの短い符号語を割り振ればよい。実際，英語のモールス符号は，この考えに基づいて作られている(**図1**参照)。

図1の符号を用いればASCIIコードを用いる場合に比べて英語の文章を効率よく符号化できるが，まだ十分な圧縮効果は得られない。もっと効率のよい圧縮を行うためには，図1のように1文字ごとに符号化するのではなく，数文字単位で符号化する必要がある。例えば，簡単のために，|a, b|の2種類の文字からなるデータ系列を|0, 1|の系列に符号化する場合を考えてみる。a，bの生起確率P_a，P_bを，それぞれ0.8，0.2とする。この時，1文字ごとに符号化したのでは

a→0　　b→1

または

a→1　　b→0

に対応づけるしかなく，元のデータ1文字を表すのに1ビット必要となり圧縮が行えない。しかし，2文字単位で**図2**のように符号化すれば，元のデータ1文字を平均0.78ビットで表現できる。さらに3文字単位では平均0.728ビット(**図3**)で表せる。

以上をまとめると，生起確率の大きいものには短い符号語を，生起確率の小さいものには長い符号語を対応づけ，かつ一度に符号化する文字単位の個数をできるだけ大きく選べば，効率よく圧縮できる。

さて，データはいくらでも短く圧縮できるというものではなく，限界が存在する。その限界は，データ圧縮の研究の創始者で

図3　3文字単位の符号化（ハフマン符号）

データ	生起確率	符号語	符号語長	符号語長×生起確率
aaa	0.512	0	1	0.512
aab	0.128	100	3	0.384
aba	0.128	101	3	0.384
baa	0.128	110	3	0.384
abb	0.032	11100	5	0.16
bab	0.032	11101	5	0.16
bba	0.032	11110	5	0.16
bbb	0.008	11111	5	0.04
		平均符号語長		2.184/3=0.728

あるC. E. Shannonにより明らかにされており，その限界を達成するためにいろいろな符号が提案されている。その代表的なものにシャノン・ファノ(Shannon-Fano)符号やハフマン(Huffmann) 符号などがある。

しかし，これらの符号を用いて圧縮限界を達成するには，一度に符号化する文字単位数を長くしなければならないが，メモリ容量や計算時間の制限で実用上あまり長くできない欠点がある。また，途中で生起確率が大きく変化するようなデータに対して，アダプティブに符号を修正することも困難である。

最近，これらの欠点を解決した符号として「算術符号(Arithmetic code)」が提案され，文書データや画像データなどのあらゆる種類のデータ圧縮用に応用され始めている。この算術符号は，データを何文字かごとに区切って符号化するのではなく，逐次的なアルゴリズムで符号化するため，等価的に全データを一度に符号化するのと等価となり，容易に圧縮限界を達成できる。また，適応性にもすぐれ，統計的な性質が一定でないデータをも効率よく圧縮できる。次節ではこの算術符号を簡単に説明し，第5節でフロッピーディスク内のファイルを圧縮するためのプログラムを紹介する。

なお，データ圧縮用符号には，上記の符号以外に「ランレングス符号」がよく知られている。これは白黒画像データなどのように2つの文字(この場合には白か黒)の一方が極端に生起しやすいデータに適した符号である。例えば，文字の種類が{w, b}でwが非常に生起しやすい場合，**図4**のようなランレングス符号が考えられる。符号語はwが続いて生起する長さ(ランレングス)を2進符号化したものである。この符号で

wwwwb wwwwwwwb wwwb

wwwwwwb

を符号化すると

100 110 011 110

と圧縮できる。しかし，ランレングス符号は，2つの文字の生起確率に極端な差がない場合には効率がわるくなり，場合によっては下の例のように，符号化によりかえって長くなる場合がある。

wb wwb wb b wwb wb

001 010 001 000 010 001

したがって，ランレングス符号は，文書ファイルなどのデータ圧縮には適していない。

データ	符号語
b	000
wb	001
wwb	010
wwwb	011
wwwwb	100
wwwwwb	101
wwwwwwb	110
wwwwwww	111

図4　ランレングス符号

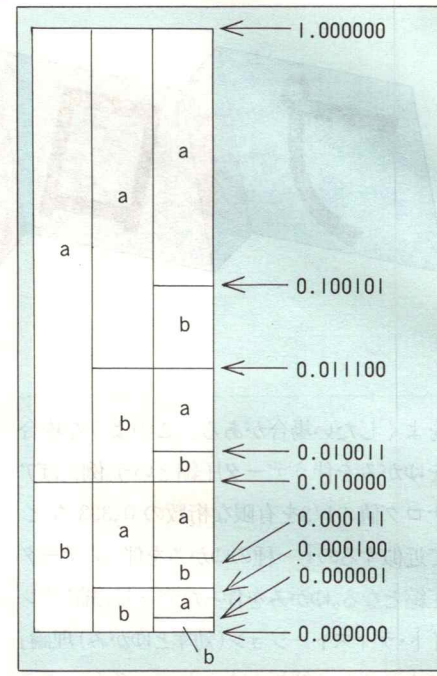

図5　[0, 1]の分割

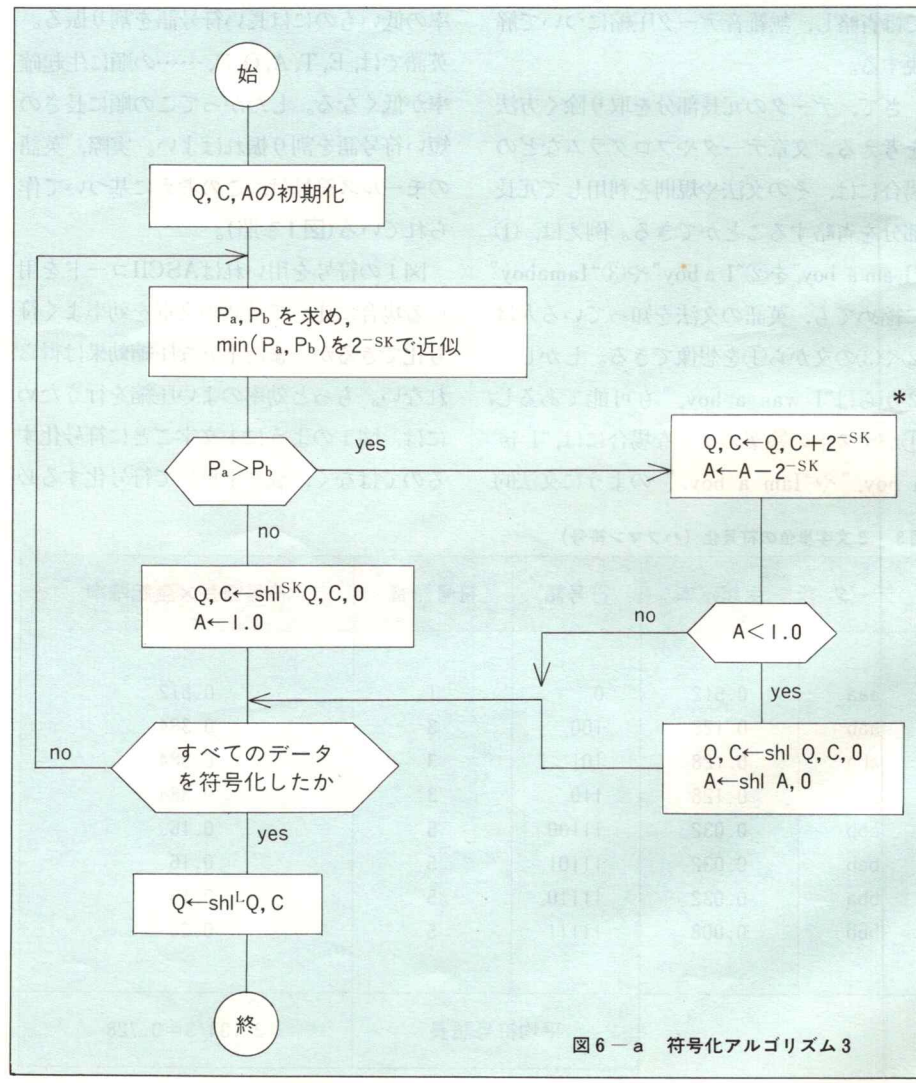

図6－a　符号化アルゴリズム3

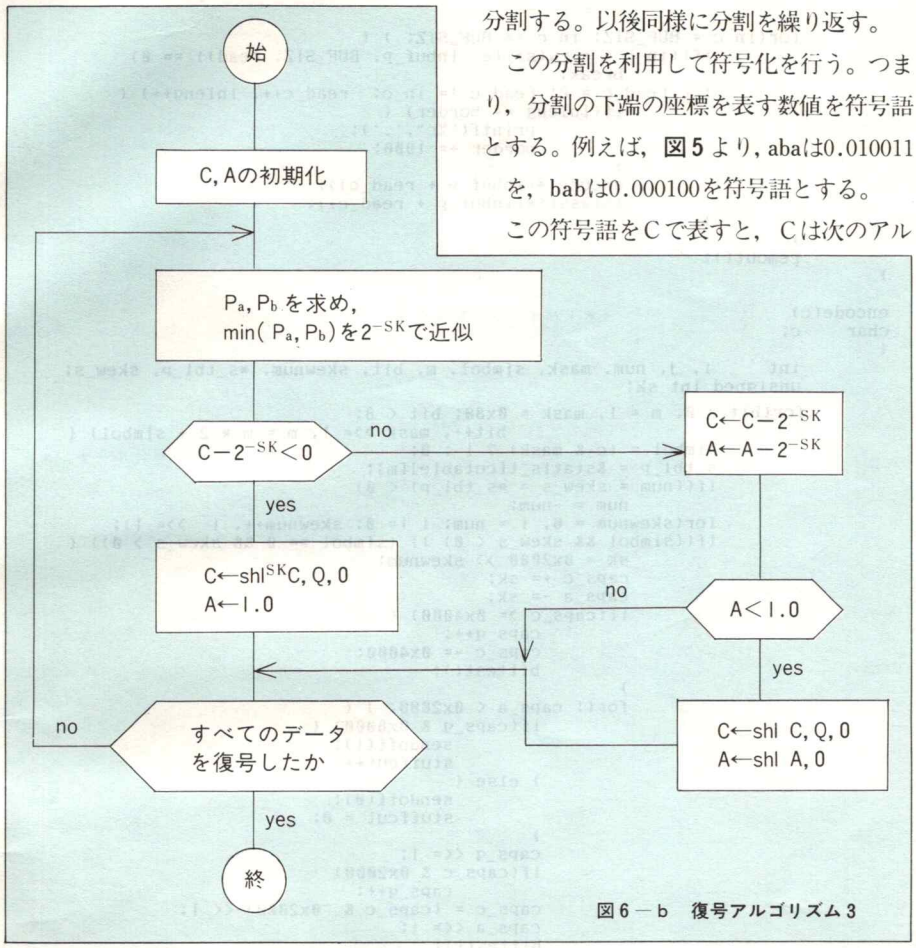

図6－b　復号アルゴリズム3

分割する。以後同様に分割を繰り返す。

この分割を利用して符号化を行う。つまり，分割の下端の座標を表す数値を符号語とする。例えば，**図5**より，abaは0.010011を，babは0.000100を符号語とする。

この符号語をCで表すと，Cは次のアルゴリズムで求められる。

［符号化アルゴリズム1］

① 初期化　C←0，A←1

② データから1文字読み込む

③ 文字がbの時，
A←A×P_b，②へ戻る

④ 文字がaの時，
C←C＋A×P_b，A←A×P_a（＝A－A×P_b）
②へ戻る。

ここで，Aは図5における分割の幅に対応している。

例　abaの場合

- 初期化　C←0，A←1
- a：C←0＋1×0.01＝0.01
  A←1×0.11＝0.11
- b：A←0.11×0.01＝0.0011
- a：C←0.01＋0.0011×0.01＝0.010011
  A←0.0011×0.11＝0.001001

Cからの元のデータを復号するには次のアルゴリズムで行える。

［復号アルゴリズム1］

① 初期化　C←符号語，A←1

算術符号の符号化アルゴリズムを説明する前に，その基礎となった考え方を示す。ただし，簡単のためにデータは｛a, b｝の2文字からなり，a, bの生起確率をそれぞれP_a＝3/4（2進数で0.11），P_b＝1/4(0.01)として説明する。

［0, 1）の区間をa，bの生起確率の大きさにしたがって図5のように2分割する。bに対応した[0, 0.01)とaに対応した[0.01, 1)をそれぞれさらに生起確率にしたがって2

リスト1　Encode.C

```
#include "com.h"                    /*共通ルーチンを結合*/
main(argc, argv)                    /*Encode. Cのメインルーチン*/
int     argc;
char    *argv[];
{
        long    pos, lseek();
        char    cb[4];
        ex_set(*++argv);
        if((sfile = open(*argv, READ)) == EOF || argc != 2) {
                return(printf("File not found !¥n"));
        }
        if((dfile = creat(fname,WRITE)) == EOF) {
                return(printf("File not created !¥n"));
        }
        printf("Encode : '%s' ==> '%s'¥n",*argv,fname);
        pos = lseek(sfile, 0L, 2);
        lseek(sfile, 0L, 0);
        cb[0] = pos;
        cb[1] = (pos >> 8);
        cb[2] = (pos >> 16);
        cb[3] = (pos >> 24);
        c_rw(dfile, &cb[0], 4, write);
        printf("Source file size = %ld [bytes]¥n", pos);
        parainit();
        m_proce();
        c_close(sfile);
        c_close(dfile);
        printf("¥nCompressed file size = %ld [bytes]¥n", outleng / 8 + 4);
        b = outleng / 8 + 4;
        printf("Compression rate = %.3f¥n",b / pos);
}

m_proce()                           /*符号化メインルーチン*/
{
        int     i;
        printf("Encoding now !");
```

② C'=C−A×P_bを計算する

③ C'<0の時，bと復号

A←A×P_b，②へ戻る

④ C'≧0の時，aと復号

C←C', A←A×P_a(=A−A×P_b)

②へ戻る

例 C=0.010011の復号

●初期化 C←0.010011，A←1

●C'=0.010011−1×0.01=0.000011>0

aと復号，C←0.000011, A←1×0.11

●C'=0.000011−0.0011×0.01<0

bと復号，A←0.11×0.01=0.0011

●C'=0.000011−0.0011×0.01=0

aと復号，C←0，A←0.0011×0.11

=0.001001

上の例からも分かるように，復号アルゴリズム1を用いれば，Cから誤りなく元のデータが復元できる。しかし，この符号化アルゴリズムでは，すべて同じ長さの符号語に符号化され，生起確率の大きいものほど，短い符号語を割り当てるというデータ圧縮の原理を満たしていない。そこで，この条件を満たすようにアルゴリズム1を次のように修正する。

[符号化アルゴリズム2]

① 初期化 C←0，A←1

② データから1文字読み込む

③ 文字がbの時，

A←A*×P_b，②へ戻る

④ 文字がaの時，

C←C+A*×P_b, A←A−A*×P_b

②へ戻る。

[復号アルゴリズム2]

① 初期化 C←符号語，A←1

② C'=C−A*×P_bを計算する

③ C'<0の時，bと復号

A←A*×P_b，②へ戻る

④ C'≧0の時，aと復号

C←C'，A←A−A*×P_b

②へ戻る

ここで，A*はAを2進数表示した時の有効桁第2桁目以降を切り捨てたものである。このようにAをA*で近似することにより不

```
        for(in_c = BUF_SIZ; in_c == BUF_SIZ; ) {
                if((in_c = c_rw(sfile, inbuf_p, BUF_SIZ, read)) == 0)
                        break;
                for(read_c = 0; read_c != in_c;  read_c++, inleng++) {
                        if(inleng == border) {
                                printf("%c",'.');
                                border += 1000;
                        }
                        encode(*(inbuf_p + read_c));
                        tblassi(*(inbuf_p + read_c));
                }
        }
        remout();
}

encode(c)                       /*1バイト分の符号化ルーチン*/
char    c;
{
        int     i, j, num, mask, simbol, m, bit, skewnum, *s_tbl_p, skew_s;
        unsigned int sk;
        for(bit = 0, m = 1, mask = 0x80; bit < 8;
                                bit++, mask >>= 1, m = m * 2 + simbol) {
                simbol = (c & mask) ? 1 : 0;            /*1ビット符号化ルーチン*/
                s_tbl_p = &statis_t[cutable][m];
                if((num = skew_s = *s_tbl_p) < 0)
                        num = -num;
                for(skewnum = 0, i = num; i != 0; skewnum++, i  >>= 1);
                if((simbol && skew_s < 0) || (simbol == 0 && skew_s > 0)) {
                        sk = 0x2000 >> skewnum;
                        caps_c += sk;
                        caps_a -= sk;
                        if(caps_c >= 0x4000) {
                                caps_q++;
                                caps_c -= 0x4000;
                                bittest();
                        }
                        for(; caps_a < 0x2000; ) {
                                if(caps_q & 0x8000) {
                                        sendoff(1);
                                        stuffcut++;
                                } else {
                                        sendoff(0);
                                        stuffcut = 0;
                                }
                                caps_q <<= 1;
                                if(caps_c & 0x2000)
                                        caps_q++;
                                caps_c = (caps_c & ~0x2000) << 1;
                                caps_a <<= 1;
                                bittest();
                        }
                } else {
                        for(i = 0; i < skewnum; i++) {
                                if(caps_q & 0x8000) {
                                        sendoff(1);
                                        stuffcut++;
                                } else {
                                        sendoff(0);
                                        stuffcut = 0;
                                }
                                caps_q <<= 1;
                                if(caps_c & 0x2000)
                                        caps_q++;
                                caps_c = (caps_c & ~0x2000) << 1;
                                bittest();
                        }
                        caps_a = 0x2000;
                }                                       /*生起確率更新*/
                if(simbol) {
                        if(*s_tbl_p == 1)
                                *s_tbl_p = -1;
                        else {
                                if(*s_tbl_p > 0)
                                        *s_tbl_p >>= 1;
                                else if(*s_tbl_p != -0x1fff)
                                        (*s_tbl_p)--;
                        }
                } else {
                        if(*s_tbl_p == -1)
                                *s_tbl_p = 1;
                        else {
                                if(*s_tbl_p < 0) {
                                        *s_tbl_p = -*s_tbl_p;
                                        *s_tbl_p >>= 1;
                                        *s_tbl_p = -*s_tbl_p;
                                } else if(*s_tbl_p != 0x1fff)
                                        (*s_tbl_p)++;
                        }
                }
        }
}

remout()                        /*符号化終了後レジスタ内の残りビットを送出*/
{
```

必要に桁数が増加するのを抑えることができる。

例　abaの場合

符号化

●初期化　C←0，A←1，A*←1

●a：　C←0+1×0.01=0.01

A←1−1×0.01=0.11，A*←0.1

●b：　A←0.1×0.01=0.001，A*←0.001

●a：　C←0.01+0.001×0.01=0.01001

A←0.001−0.001×0.01=0.00011

A*←0.0001

C=0.01001の復号

●初期化　C←0.01001，A←1，A*←1

●C'=0.01001−1×0.01=0.00001>0

aと復号，C←0.00001，

A←1−1×0.01=0.11，　A*←0.1

●C'=0.00001−0.1×0.01<0

bと復号，

A←0.11×0.01=0.0011，A*←0.001

●C'=0.00001−0.001×0.01=0

aと復号，C←0，

A←0.001−0.001×0.01=0.00011

A*←0.0001

このようにアルゴリズム2を用いれば，abaをアルゴリズム1で符号化した時に比べて1ビット少ない符号語で表現できる。さらにもっと長いデータを符号化すると

bbbabba：0.0000000100001

aaabaab：0.100011

aaaaaaa：0.1101

のような符号語が得られる。このように生起確率の大きいaを多く含むデータは短い符号語に，また生起確率の小さいbを多く含むデータは長い符号語に符号化される。実際このアルゴリズム2を用いて，1000文字とか2000文字とかの長いデータを符号化すれば非常によい圧縮率を達成できる。

しかし，アルゴリズム2のままでは，長いデータを符号化した場合，小数点以下の桁数が大きくなりすぎて処理しきれなくなる。そこで，符号化が進むにつれて，Cの上位桁を符号語として逐次送り出すように改良する必要がある。このように改良した

```
        int     i;
        for(i = 0; i < 16; i++, caps_q <<= 1)
                sendoff((caps_q & 0x8000) ? 1 : 0);
        for(i = 0; i < 14; i++, caps_c <<= 1)
                sendoff((caps_c & 0x2000) ? 1 : 0);
        c_rw(dfile, outbuf_p, out_c, write);
        if(out_b_c != 0) {
                outchr <<= 8 - out_b_c;
                c_rw(dfile, &outchr, 1, write);
                outleng += 8;
        }
}

bittest()               /*キャリーオーバーチェックルーチン*/
{
        int     test_b, bit_b;
        test_b = 0xffff;
        test_b <<= stuffcut;
        if((~test_b | caps_q) != 0xffff)
                return;
        bit_b = 1;
        bit_b <<= stuffcut;
        caps_q ^= bit_b;
        sendoff(1);
}

sendoff(bit)                    /*符号化されたビット送出*/
int     bit;
{
        if(begcut <= 0)
                begcut++;
        else {
                outchr = (outchr << 1) + bit;
                outleng++;
                out_b_c++;
                if(out_b_c == 8) {
                        *(outbuf_p + out_c) = outchr;
                        out_b_c = 0;









                        if(++out_c == BUF_SIZ) {
                                c_rw(dfile, outbuf_p, out_c, write);
                                out_c = 0;
                        }
                }
        }
}
```

リスト2　Decode.C

```
#include        "com.h"         /*共通ルーチンを結合*/
main(argc, argv)                /*Decode.Cのメインルーチン*/
int     argc;
char    *argv[];
{
        char    cb[4];
        ex_set(*++argv);
        printf("Decode : '%s' ==> '%s'¥n",fname,*argv);
        if((sfile = open(fname, READ)) == EOF || argc !=2) {
                return(printf("File not found !¥n"));
        }
        if((dfile = creat(*argv, WRITE)) == EOF) {
                return(printf("File not created !¥n"));
        }
        c_rw(sfile, &cb[0], 4, read);
        out_siz += cb[3];
        out_siz = (out_siz << 8) + cb[2];
        out_siz = (out_siz << 8) + cb[1];
        out_siz = (out_siz << 8) + cb[0];
        parainit();
        m_proce();
        c_close(sfile);
        c_close(dfile);
        b = outleng;
        printf("¥nDecoded file size = %5.f [bytes]¥n", b);
}

m_proce()                       /*復号メインルーチン*/
{
        char    c;
```

アルゴリズム3のフローチャートを**図6**に示す。

[アルゴリズム3]

アルゴリズム3では生起確率の小さいほうを2^{-SK}(SKは正整数)で近似し，かけ算をシフト演算で高速処理できるように工夫してある。なお，図6でC, Aは長さLビットのレジスタを表し，Qは図6-aではすでに送り出された符号語を，また，図6-bではまだ復号されていない符号語を示す。

さらに，「Q, C←shl^{SK}Q, C, 0」はQ, C, 0を左にSK回シフトすることであり，「A←shl A, 0」はA, 0を左に1回シフトすることである。つまり，「Q, C←shl^{2}Q, C, 0」は，

$$0.\cdots\cdots\overbrace{01011001}^{Q}\ \overbrace{11001}^{C}$$

$$\downarrow$$

$$0.\cdots\cdots01100111\quad 00100$$

にシフトすることであり，「A←shl A,0」は，

A=0.11011の時，A=1.10110

にすることである。

算術符号の基本的なアルゴリズムを前節で紹介したが，実際にフロッピーディスク内のファイルを効率よく圧縮するためには，次のような工夫をしなければならない。

①図6－aの＊印の演算で，すでに送り出しているQの部分にCから桁上が生じた時の処理（キャリーオーバー問題）

②生起確率(つまり，SKの値）を圧縮するファイルから逐次的に求める。

③ASCIIコード，JISコード，オブジェクトコードなどファイルは8ビットまたは16ビット単位でコード化されている。そ

```
        printf("Decoding now !");
        read_c = in_c = BUF_SIZ;
        iniget();
        while(outleng < out_siz) {
                if(outleng == border) {
                        printf("%c",',');
                        border += 1000;
                }
                c = decode();
                tblassi(c);
                sendoff(c);
        }
        if(out_c != 0)
                c_rw(dfile, outbuf_p, out_c, write);
}

decode()                        /*１バイト分の復号ルーチン*/
{
        int     i, j, num, simbol, m, bit, skewnum, *s_tbl_p, skew_s, c;
        unsigned int    sk;
        for(c = bit = 0, m = 1; bit < 8;
                bit++, m = m * 2 + simbol, c = (c << 1) + simbol) {
                s_tbl_p = &statis_t[cutable][m];
                if((num = skew_s = *s_tbl_p) < 0)
                        num = -num;
                for(skewnum = 0, i = num; i != 0; skewnum++, i >>= 1) ;
                simbol = (skew_s > 0) ? 0 : 1;      /*１ビット復号ルーチン*/
                sk = 0x2000 >> skewnum;
                if(caps_c >= sk) {      /*event true*/
                        caps_c -= sk;
                        caps_a -= sk;
                        for(;caps_a < 0x2000;) {
                                caps_c = (caps_c & ~0x2000) << 1;
                                caps_a <<= 1;
                                chrget();
                                if(inchr & in_mask) {
                                        stuffcut++;
                                        caps_c++;
                                } else
                                        stuffcut = 0;
                                if(stuffcut == 16) {
                                        chrget();
                                        stuffcut = 0;
                                        for(;;) {
                                                if(inchr & in_mask) {
                                                        caps_c++;
                                                        chrget();
                                                } else
                                                        break;
                                        }
                                }
                        }
                } else {
                        simbol = (simbol == 0) ? 1 : 0;
                        for(i = 0; i < skewnum; i++) {
                                caps_c = (caps_c & ~0x2000) << 1;
                                chrget();
                                if(inchr & in_mask) {
                                        stuffcut++;
                                        caps_c++;
                                } else
                                        stuffcut = 0;
                                if(stuffcut == 16) {
                                        chrget();
                                        stuffcut = 0;
                                        for(;;) {
                                                if(inchr & in_mask) {
                                                        caps_c++;
                                                        chrget();
                                                } else
                                                        break;
                                        }
                                }
                        }
                        caps_a=0x2000;
                }                               /*生起確率更新*/
                if(simbol) {
                        if(*s_tbl_p == 1)
                                *s_tbl_p = -1;
                        else {
                                if(*s_tbl_p > 0)
                                        *s_tbl_p >>= 1;
                                else if(*s_tbl_p != -0x1fff)
                                        (*s_tbl_p)--;
                        }
                } else {
                        if(*s_tbl_p == -1)
                                *s_tbl_p = 1;
                        else {
                                if(*s_tbl_p < 0){
                                        *s_tbl_p = -*s_tbl_p;
                                        *s_tbl_p >>= 1;
                                        *s_tbl_p = -*s_tbl_p;
                                } else if(*s_tbl_p != 0x1fff)
                                        (*s_tbl_p)++;
```

のため第1ビット目，第2ビット目，…で{0,1}の生起確率が異なる。そこで効率よく圧縮するには各ビット目ごとに生起確率を求めなければならない。また，英語ではtの次にはhが生起しやすいなどのバイト単位での文字の続きやすさも考慮に入れて，効率をよくする必要がある。

このような処理を行ったLattice-C用のプログラムを**リスト1，リスト2，リスト3**に示す。「ENCODE. C」が符号化プログラム，「DECODE. C」が復号プログラムである。また，「COM. H」はENCODE. C，DECODE. Cの両方で利用する共通ルーチンである。なお，Optimizing-Cの場合は

#define READ　0×8000→#define READ 4

#define WRITE 0×8001→#define WRITE 5

char→unsigned char

のように変更する必要がある。

ENCODE. C，DECODE. Cをコンパイルした実行形式のファイルをENCODE. EXE, DECODE. EXEとすると符号化および復号は**図7**のようになる。圧縮されたファイルの拡張子はCPFとなる。復号は拡張子がCPFのファイルのみを復号する。Encoding（またはDecoding）now！の後に出力される「・」は，ファイルが1Kバイト処理されるごとに出力され，クロック8MHzで約1秒の処理時間を必要とする。

図8に，Optimizing-Cの各種ファイルを圧縮した時の圧縮率(Rate)を示す。Optimizing-Cは，購入時に圧縮されたファイル

```
                }
            }
        }
        return(c);
}

iniget()                    /*caps_Cに必要な符号語をシフトイン*/
{
        int     i, simbol;
        for(i = 0; i < 14; i++) {
                chrget();
                simbol = (inchr & in_mask) ? 1 : 0;
                caps_c = (caps_c << 1) + simbol;
        }
}

chrget()                    /*1バイト分の符号語を取り出す*/
{
        in_mask >>= 1;
        if(in_mask == 0) {
                in_mask = 0x80;
                inchr = 0;
                if(read_c == in_c) {
                        if(in_c != BUF_SIZ)
                                return;
                        if((in_c = c_rw(sfile, inbuf_p, BUF_SIZ, read)) == 0)
                                return;
                        read_c = 0;
                }
                inchr = *(inbuf_p + read_c++);
        }
}

sendoff(c)                  /*復号したデータを送出*/
char    c;
{
        *(outbuf_p + out_c) = c;
        outleng++;
        if(++out_c == BUF_SIZ) {
                c_rw(dfile, outbuf_p, out_c, write);
                out_c = 0;
        }
}
```

```
A>ENCODE CC1.EXE          <----------CC1.EXEを圧縮して
Encode : 'CC1.EXE' ==> 'CC1.CPF'          CC1.CPFを作る
Source file size = 31110 [bytes]
Encoding now !.............................
Compressed file size = 21402 [bytes]
Compression rate = 0.688

A>DECODE CC1.DEC          <----------CC1.CPFを復号して
Decode : 'CC1.CPF' ==> 'CC1.DEC'          CC1.DECを作る。
Decoding now !.............................
Decoded file size = 31110 [bytes]

A>FC /B CC1.EXE CC1.DEC   <----------MS-DOSのファイル比較プログラム
                                    FC.EXEを用いて，CC1.EXEと
A>                                  CC1.DECの内容が等しいことを確認
```

図7　実行例

リスト3　Com.h（共通ルーチン）

```
#include        "stdio.h"
#define         READ            0x8000
#define         WRITE           0x8001
#define         L_NUM           50
#define         BUF_SIZ         10000
#define         THRESH_NUM      50
#define         TRUE            1
#define         FALSE           0
int             statis_t[L_NUM][256], context[256];
int             sfile, dfile, out_b_c = 0, out_c = 0, in_mask = 0, read_c,in_c;
int             nextassign = 1, cutable = 0, stuffcut = 0, begcut = -15;
unsigned int    caps_c = 0, caps_a = 0x2000, caps_q = 0;
long            inleng, outleng = 0, border = 1000, out_siz;
char            inbuf_p[BUF_SIZ], outbuf_p[BUF_SIZ], fname[128];
char            outchr, inchr;
float           a, b;
int             read(), write();

ex_set(argv)                /*拡張子チェック*/
char    *argv;
{
        char    *p;
        int     i;
        p = fname;
        strcpy(fname, argv);
        for(i = 0; *p != '¥0' && *p != '.'; i++, p++) ;
```

が供給され，それを復号して使用するようになっているが，その圧縮ファイル（Opt. C Sq. Files）との比較も併せて示してある。本方式のほうが，効率よく圧縮できていることが分かる。なお，「COM.H」内のL_NUMの値50とTHRESH_NUMの値50を変更すると圧縮率が若干変化する。メモリ容量の許す範囲で，各自最適値を求めてみていただきたい。

本稿ではデータ圧縮理論の簡単な紹介を行い，算術符号を用いたファイル圧縮プログラムを紹介した。ASCIIファイルや，BASIC言語などのある特殊なファイルのみを圧縮する専用プログラムを除き，任意のファイルを圧縮できる汎用プログラムとしては，本プログラムが最も優れたものの1つであると思われる。

なお，本プログラムの使用に当たっては次のことに注意していただきたい。

①圧縮符号化した後に元のファイルを消去する場合は，先に復号プログラムで誤りなく復号できることを確認してから消去する。

②無断で商用に本プログラムを使用しない。

（徳島大学）

参考文献

[1] G. G. Langdon, Jr,"An introduction to Arithmetic Coding", IBM J. Res. Develop., vol. 28, no. 2, March 1984

[2] G.G.Langdon, Jr. and J.Ressanen,"A Simple General Binary Source," IEEE Trans. vol. IT-28,pp. 800-803, Sep. 1982

[3] G.G.Langdon, Jr. and J.J.Ressanen,"A Double Adaptive File Compression Algorithm", IEEE Trans. vol. COM-31, pp. 1253-1255, Nov. 1983

[4] 森田，藤本，北田，有本，"二値算術符号の符号高率について"，情報処理学会論文誌，vol.25, pp. 622-631, July 1984


```
        if(strcmp(p, ".cpf") == 0 || strcmp(p,".CPF") == 0){
                printf("Bad extention !¥n");
                _exit();
        }
        strcpy(p, ".CPF");
}

tblassi(c)                              /*生起確率テーブル割り振りルーチン*/
char    c;
{
        int     *c_tbl_p;
        c_tbl_p=&context[c];
        if(*c_tbl_p < 0) {
                (*c_tbl_p)++;
                if(*c_tbl_p == 0 && nextassign != 0) {
                        *c_tbl_p = nextassign++;
                        if(nextassign == L_NUM)
                                nextassign = 0;
                }
        }
        if(*c_tbl_p <= 0)
                cutable = 0;
        else
                cutable = *c_tbl_p;
}

parainit()                              /*初期化ルーチン*/
{
        int     *num, *enum, i;
        for(num = &statis_t[0][0], enum = &statis_t[L_NUM - 1][255]; num <= enum
; num++)
                *num = 1;
        for(i = 0; i < 256; i++)
                context[i] = -THRESH_NUM;
}

c_rw(file, buffer, length, dio)         /*入出力ルーチン*/
int     file;
char    *buffer;
int     length, (*dio)();
{
        int     status;
        if((status = (*dio)(file, buffer, length)) == EOF)
                c_exit();
        return(status);
}

c_close(file)                           /*ファイルクローズルーチン*/
int     file;
{
        if(close(file) == EOF)
                c_exit();
}

c_exit()                                /*Exitルーチン*/
{
        printf("¥nDisk I/O error !¥n");
        _exit();
}
```

図8　圧縮率の比較

ファイル名	Bytes	Opt.C Sq. Files Bytes(Rate)	本圧縮方法 Bytes(Rate)
CC1.EXE	31110	26356 (0.847)	21402 (0.688)
CC2.EXE	58543	48745 (0.833)	37140 (0.634)
CC3.EXE	57282	46608 (0.814)	35261 (0.616)
CC4.EXE	48975	40855 (0.834)	31483 (0.643)
MARION.EXE	23246	19570 (0.842)	16354 (0.704)
ARCH.EXE	18407	15625 (0.849)	13128 (0.713)
BASE.ARC	65871	45419 (0.690)	37465 (0.569)
DOS2.ARC	45626	32485 (0.712)	25184 (0.552)
DOSALL.ARC	37091	26593 (0.717)	20751 (0.559)
MATH87.ARC	45136	30781 (0.682)	25172 (0.558)
MATHSFT.ARC	28165	19140 (0.680)	15064 (0.535)
ERRON.H	1203	1045 (0.869)	928 (0.771)
STDIO.H	896	827 (0.923)	686 (0.766)
PROLOGUE.H	768	736 (0.958)	585 (0.762)
MODEL.H	256	268 (1.047)	150 (0.586)
EPILOGUE.H	128	153 (1.195)	51 (0.398)
C86BAN.LIB	117248	65648 (0.560)	54014 (0.461)
C86B2N.LIB	115200	64453 (0.559)	53518 (0.465)
C86S2N.LIB	108032	57825 (0.535)	46555 (0.431)
C86SAN.LIB	107008	57242 (0.535)	45916 (0.429)
C86BAS.LIB	133632	79426 (0.594)	65004 (0.486)
C86B2S.LIB	131584	78318 (0.595)	64537 (0.490)
C86S2S.LIB	120320	67213 (0.559)	53617 (0.446)
C86SAS.LIB	119296	66624 (0.558)	52977 (0.444)