マルティメディア情報論 〜ハフマン符号〜

k098579 金城佑典

2009/12/07

目次

ソースコード目次

1　平均情報量（エントロピー）

情報理論の概念で、ある事象が起きた際、それがどれほど起こりにくいかを表す尺度。エントロピー（$H(P)$）以下の式で与えられる。（ただし、P(X) は X の生起確率）

$$H(P) = -\sum P(x) log_2 P(x) \tag{1}$$

2　ハフマン符号化

1952 年にデビット・ハフマンによって開発された符号、 コンパクト符号やエントロピー符号の一つ。

1. データに出現する記号の生起確率を調べる。
   データに出現する記号ごとに出現確率を計算しておく。出現した符号は葉として扱う。
2. 生起確率の順に並べ替える
   生起確率の小さい順にソートする
3. 一番小さい物と二番目に小さいものを結合し、新しい節を作る。
   生起確率が小さいもの２つを結合し節を作る、節の生起確率は２つの生起確率を足したものにする。
4. すべて結合されるまで (1) と (2) を繰り返す。
   すべての節が結合し、ハフマン木の根ができるまで (1) と (2) を繰り返す。
5. 根から順に左右に 0 と 1 の値を割り振る
   ハフマン木の根から葉に向けて、左右に 0 と 1 の値を割り振る、ただし、どちらに 0 と 1 を振るかは任意。

以下に、’A’,’B’,’C’,’D’ の４通りの記号からなる 20 文字のデータを例にハフマン符号化を図示する。データは「ABBDAADBCCBCACABDBAB」の２０個とする。

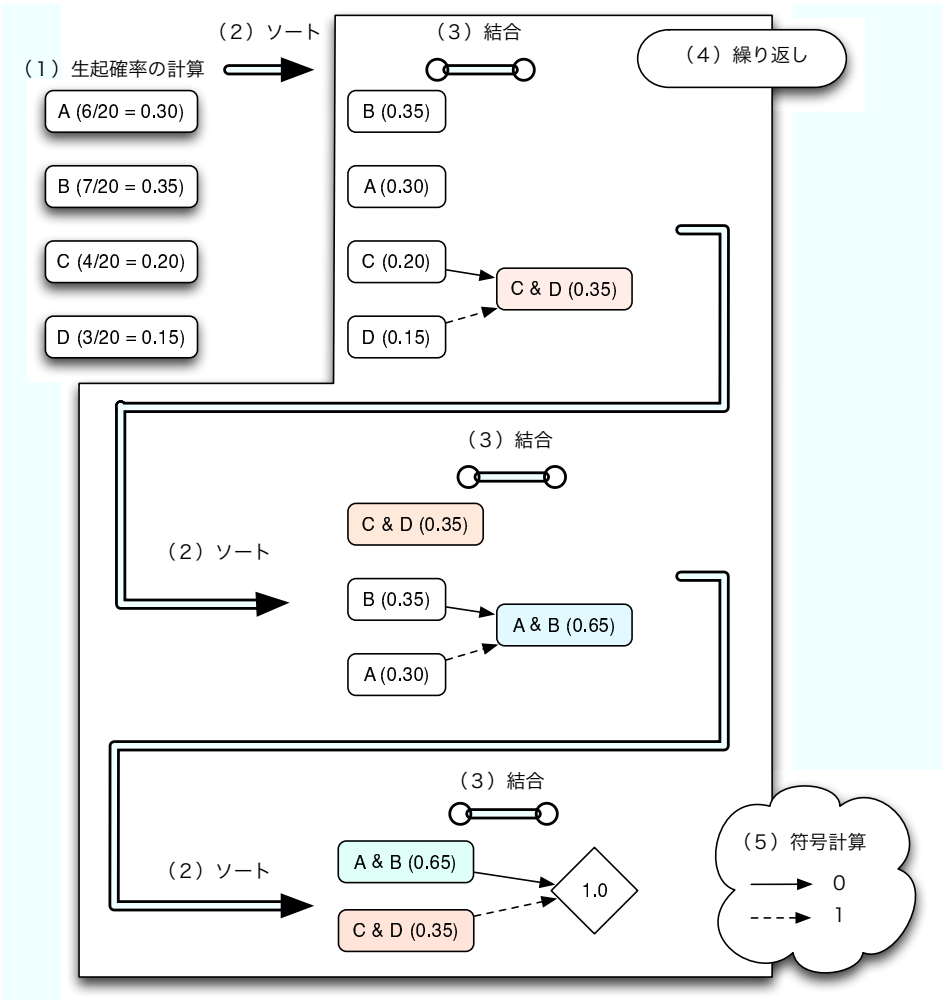

図 1　ハフマン符号化の例

よって、A → 01,B → 00,C → 10,D → 11 となる。

3　平均符号長

符号の平均的な長さ、平均符号長が前述のエントロピーに近いほどよい符号化であるといえる。

その符号が表す記号の生起確率を P(X)、符号長を $H_{len}(X)$ とする、平均符号長 ACL は以下の式で表される。

$$ACL = \sum P(x) * H_{len}(X) \tag{2}$$

4　Exercise 0A and 0B

以下の条件を満たすことハフマン符号を計算するプログラムを作成する。

「言語はなんでもよい」「グレースケール画像を読み込める」「エントロピーが計算できる」「平均符号長が計算できる」。

4.1　プログラムの仕様

- ソートにはバブルソートを用いる。
- 画像への対応は OpenCV を用いる。
- Exercise1 のために CSV ファイルを読み込めるようにする。

4.1.1　記号と生起確率

記号と生起確率は Data 構造体に格納する。data には int 型を用いているため、記号の ID を格納する。生起確率は priority に格納する、精度は double 型の精度に依存する。

4.1.2　ハフマン木の表現

TreeNode 構造体はハフマン木を表現するのに用いる。

node は前述の Data 構造体である。parent、left、right はそれぞれ親、左の子、右の子を指す。

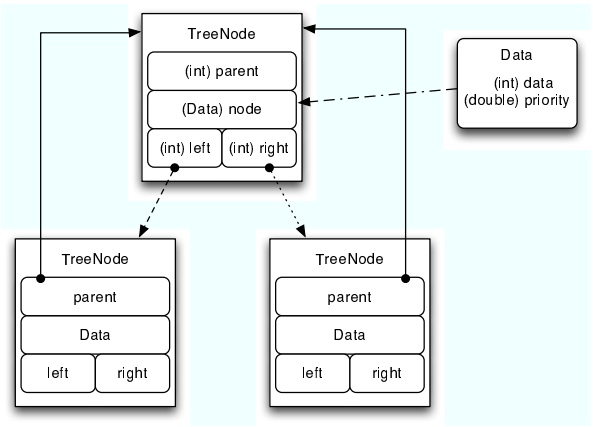

図 2　TreeNode 構造体

ハフマン木は TreeNode 構造体の配列を用いて表現する。

つまり、parent、left、right には実際には対象データが入っている配列のインデックスが代入される。

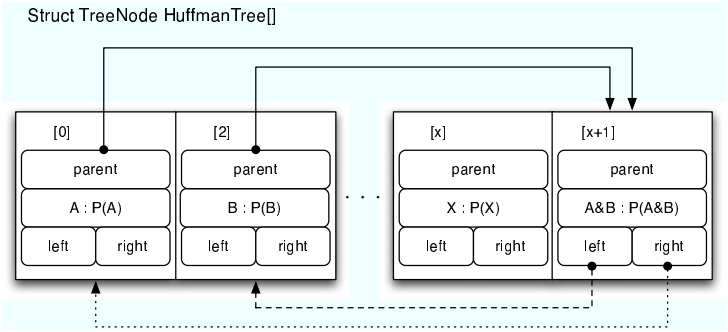

図 3　TreeNede 構造体の配列を用いたハフマン木の表現

4.1.3　ハフマン符号表の表現

ハフマン符号表は CodeMap 構造体の配列で表現する。

CodeMap 構造体では、data に記号 ID を、code に符号を length に符号長をそれぞれ格納する。

4.2　プログラムのソースコード

4.2.1　Makefile と定義部

画像の読み込みモードを有効にするには、OpenCV がインストールされている環境で、コンパイル時に「$-DOPEN_CV$」オプションをつけてコンパイルする必要がある。

以下、「$\#ifdef\ OPEN\_CV$」部分は上記オプションを指定していなければ利用されない。

ソースコード 1　Makefile

```
1 CC = gcc -O2
2
3 ##OpenCV がインストールされてるとき
4 OPENCV = -DOPEN_CV `pkg-config --cflags opencv` `pkg-config --libs opencv`
5 ##OpenCV がインストールされてないとき
6 #OPENCV =
7
```

```
all:
	$(CC) Huffman.c $(OPENCV) -o Huffman
```

プログラムは 3 つの構造体と 12 の関数を持っている。扱える符号長は「MAX_CODE_LEN」で定義しているが、ハフマン符号の性質上 $2^{MAX\_CODE\_LEN}$ 種類の記号が判別できるとは限らない。[*1]

ソースコード 2　定義部

```
#include <stdio.h>
#include <stdlib.h>//exit 用
#include <getopt.h>//オプション取得用
#include <math.h>
#include <string.h>
#include <stdbool.h>//bool 用
#include <limits.h>//INT_MIN 用
#include <float.h>//FLT_MAX 用

#define MAX_CODE_LEN 100

struct Data {
	int data;//!< データ
	double priority;//!< 優先順位(小さい方が優先順位高い)
};

struct TreeNode {
	struct Data node;//!< ノード
	int parent;//!< 親のindex
	int left;//!< 左の子のindex
	int right;//!< 右の子のindex
};

struct CodeMap {
	int data;//!< データ
	char code[MAX_CODE_LEN];//!< 符号
	int length;//!< 符号長
};

#define READ_CSV_MODE 0//!< CSV を読み込むモード
```

[*1] 例えば、記号の種類が 7 つでも、生起確率によっては符号が 6 桁になることもある。

```
31 #define READ_IMG_MODE 1//!< 画像ファイルを読み込むモード
32 // (OpenCV がないと使えない)
33
34 int mode=READ_CSV_MODE;
35
36 bool debug = false;
37
38 int main(int argc,char **argv);
39
40 //ハフマン符号化関連の関数
41 int createHuffmanTree(struct TreeNode * HuffmanTree, struct Data * probDatas,
       int length);
42 void createCodeMap(struct CodeMap * maps, struct TreeNode * HuffmanTree, int
       root, int length);
43
44 //評価指標
45 double calcEntoropy(struct Data* probability,int length);
46 double calcAvarageCodeLength(struct CodeMap * maps, struct Data * input_datas,
        int length);
47
48 //ソート
49 void BubbleSort(struct Data* stream, int length);
50
51 //ファイル読み込み
52 int readProbCSV(struct Data* weighted_array,char* fileName,int type_num);
53 int readProbIMG(struct Data* input_datas,char* fileName);
54 #ifdef OPEN_CV
55         #include <highgui.h>
56         int cvReadProbIMG(struct Data* input_datas,char* fileName);
57 #endif
58
59 //その他
60 double lg(double value);
61 void ShowData(struct Data* stream, int length);
62 void usage(char *myname);
```

4.2.2 全体の流れ

全体の流れは大まかに以下のような手順になる。

1. オプションを読み込む。
2. CSV または画像ファイルから生起確率を計算する。
3. 生起確率に基づいてハフマン木を生成する。
4. ハフマン木をもとにハフマン符号を生成する。
5. エントロピーを計算し、出力する。
6. ハフマン符号表を出力する。
7. 平均符号長を計算し、出力する。

ソースコード 3　main 関数

```
1  int main(int argc,char **argv)
2  {
3          int i;
4          FILE * inputFile;
5          char* inputFileName;
6          //MAX_CODE_LEN 桁の符号しか扱えないので、少なめに定義しとく。
7          //-n オプションで変更可能
8          int type_num=MAX_CODE_LEN;
9
10         inputFileName="data.csv";
11
12 #ifdef OPEN_CV
13         printf("can␣use␣OpenCV␣images\n");
14 #endif
15
16         //オプション設定
17         int option;
18         while( (option=getopt(argc,argv,"hdi:o:n:g")) != -1 ){
19                 switch(option){
20                         case 'i'://入力ファイル
21                                 inputFileName=optarg;
22                                 break;
23                         case 'g'://入力ファイルが画像
24                                 mode=READ_IMG_MODE;
25                                 type_num=256;//8bit 画像のみ対応
26                                 break;
27                         case 'n'://読み込む要素数を指定（しなかったら全て読み込む）
28                                 type_num=atoi(optarg);
29                                 break;
```

```
30                      case 'd'://詳細な出力
31                              debug=true;
32                              break;
33                      case 'h'://Usage を表示
34                      default:
35                              usage(argv[0]);
36                              return 0;
37                              break;
38              }
39      }
40
41      printf("input : %s\n",inputFileName);
42
43      //! ファイルの読み込み
44      struct Data input_datas[type_num];
45      int length=0;
46      printf("\n\nRead File -- ");
47      if(mode == READ_CSV_MODE){//!- 要素と生起確率が書かれた
            CSV ファイルを読み込む
48              printf("CSV Mode\n");
49              length = readProbCSV(input_datas,inputFileName,type_num);
50      }else if(mode == READ_IMG_MODE){//!- 画像ファイルを読み込み、生起確率を計算
51              printf("Image Mode\n\n");
52              length = readProbIMG(input_datas,inputFileName);
53      }
54
55      if(length < 0){
56              printf("[Error] Failed to Calculate Probability\n");
57              exit(-1);
58      }
59
60      printf("\ninput datas ----------\n"); ShowData(input_datas, length);
61
62      //! ハフマン木を作ります。
63      struct TreeNode HuffmanTree[length*2+2];//!< 十分なサイズを確保
64      //ハフマン木は余裕を持たせてある（使わない
            node もある）のでちゃんと初期化しましょう。
65      for(i=0;i<length*2+2;i++){
66              HuffmanTree[i].node.data = INT_MIN;
67              //負にしとくと、うっかり使ったときに先頭になってしまう
```

```
68              HuffmanTree[i].node.priority = DBL_MAX;
69              HuffmanTree[i].parent = HuffmanTree[i].left = HuffmanTree[i].
                    right = INT_MIN;
70      }
71
72      int root = createHuffmanTree(HuffmanTree, input_datas, length);
73
74      //! ハフマン木配列の中身を確認してみませう。
75      printf("\n\nHuffmanTree␣Array␣[root(length):%d]␣--------\n",root);
76      for(i=0; i<root; i++)
77              printf("␣Huffman[%d]␣␣%d(%f)␣␣parent␣%d␣left␣%d␣right␣%d\n"
78                                      ,i,HuffmanTree[i].node.data,HuffmanTree[i
                                            ].node.priority
79                                      ,HuffmanTree[i].parent,HuffmanTree[i].left
                                            ,HuffmanTree[i].right);
80
81      //! ハフマン符号を計算します
82      printf("\n\nCalculate␣Huffman␣Code␣␣(%d␣types)--------\n",length);
83      struct CodeMap codeMaps[length];
84      createCodeMap(codeMaps, HuffmanTree, root, length);
85
86      int data;
87      for(data=0;data<length;data++)
88              printf("%d␣->␣%s\n",codeMaps[data].data,codeMaps[data].code);
89
90      //! 結果を出力する・・・・かも
91      printf("\n\nResult--------\n");
92      double entoropy = calcEntoropy(input_datas, length);
93      printf("etoropy␣=␣%f\n",entoropy);
94
95      for(data=0;data<length;data++) printf("%d␣(%f)␣->␣%s\n"
96                              ,codeMaps[data].data,input_datas[data].priority,
                                    codeMaps[data].code);
97
98      double acl = calcAvarageCodeLength(codeMaps, input_datas, length);
99      printf("Avarage␣Code␣Length␣=␣%f\n",acl);
100
101
102     return 0;
103 }
```

4.2.3 ファイルの読み込み

生起確率については、CSV ファイルから読み込むモードと画像ファイルから計算するモードを用意した。

4.2.4 CSV ファイルの読み込み

「(int)data, (double)probabirity」のような形式の CSV ファイル（$fileName$）を読み込む事ができる。data は 0 から $type\_num$ まで連続値で指定されていなければならない。また、指定可能な probabirity の範囲は環境に依存する。

ソースコード 4　CSV ファイルの例

```
0,0.3
1,0.35
2,0.20
3,0.15
```

引数として与えられた Data 構造体配列に読み込んだ結果を保存し、読み込んだケータ数を返す。

ソースコード 5　readProbCSV

```
int readProbCSV(struct Data* input_datas,char* fileName,int type_num){
        int length=0,ret;
        int data=0;
        float prob;

        //! ファイルを開く
        FILE* File = fopen(fileName,"r");
        if( File == NULL ){
                printf( "Can␣not␣Open␣[%s]\n", fileName);
                return -1;
        }

        //! 一行ごとに読み込み
        while( ( ret = fscanf( File,"%d,%f", &data, &prob) ) != EOF && length <
             type_num){
                if(debug) printf( "[readProbCSV]␣%d␣(%f)\n", data, prob);
                input_datas[length].data=(int)data;
                input_datas[length].priority=(double)prob;
```

```
18                length++;
19        }
20        fclose( File );
21
22        return length;
23 }
```

4.2.5 画像ファイルの読み込み

画像の読み込みは OpenCV を利用する場合のみ利用できる。OpenCV がない環境ではエラーを返す。

ソースコード 6　readProbIMG

```
1  int readProbIMG(struct Data* input_datas,char* fileName){
2
3  #ifdef OPEN_CV
4         return cvReadProbIMG(input_datas,fileName);
5  #endif
6
7         return -1;
8  }
```

画像の読み込みでは、画像を 256 階調のグレースケール画像に変換してから読み込む。

すべての画素にアクセスし、各階調ごとの出現数をカウントし、画素の総数で割る。ただし、double の精度しか持たないため、大きな画像を用いると精度不足で誤差が発生するので注意しなければならない。

ソースコード 7　cvReadProbIMG

```
1  #ifdef OPEN_CV
2  int cvReadProbIMG(struct Data* input_datas,char* fileName){
3         int i;
4         int x,y,ch;
5         int gradation=0;
6         double prob=0.0;
7
8         int counter[256];//8bit 画像だけ扱うよ
9         for(i=0; i<256; i++) counter[i]=0;
10
11        //! 画像の読み込み(チャンネル数は任意、グレースケール、8bit:256階調)
```

```
12        IplImage * srcImg=cvLoadImage(fileName, CV_LOAD_IMAGE_GRAYSCALE |
              IPL_DEPTH_8U);
13
14        if(!srcImg){
15          printf("Could␣not␣load␣image␣file:␣%s\n",fileName);
16          return -1;
17        }
18
19        //! 画像の情報を取得
20        int height = srcImg->height;
21        int width = srcImg->width;
22        int channels = srcImg->nChannels;
23        int step = srcImg->widthStep;
24        int depth = srcImg->depth;
25        uchar *imgdata = (uchar *)srcImg->imageData;
26        printf("Processing␣a␣%dx%d␣image␣with␣%d␣channels␣:␣depth=%d␣step=%d\n"
27                      ,height,width,channels,depth,step);
28
29        //! 画素にアクセスして階調ごとの出現数を計算
30        for(y=0;y<height; y++){
31                for(x=0;x<width; x++){
32                        for(ch=1; ch<=channels; ch++){
33                                gradation = (int)imgdata[step*y+x*channels+ch];
34                                counter[gradation] += 1 ;
35                        }
36                }
37        }
38
39        if(debug) for(i=0; i<256; i++) printf("%d␣-␣%d\n",i,counter[i]);
40
41        //! 各階調の生起確率を計算
42        for(gradation=0; gradation<256; gradation++){
43                prob = (double)((double)counter[gradation]/(double)(x*y*
                      channels));
44                if(counter[gradation]!=0)
45                        printf("%d␣-␣%f␣<-␣%d/%d\n",gradation,prob,counter[
                              gradation],(x*y*channels));
46                input_datas[gradation].data=(int)gradation;
47                input_datas[gradation].priority=(double)prob;
48        }
```

```
49
50          if(debug){
51                  cvNamedWindow("\nPress␣any␣Key␣to␣continue\n",1);
52
53                  cvNamedWindow("Source␣Image",1);
54                  cvShowImage("Source␣Image",srcImg);
55                  cvWaitKey(0);
56                  cvDestroyWindow("Source␣Image");
57          }
58
59          cvReleaseImage(&srcImg);
60
61          return 256;
62  }
63  #endif
```

4.2.6 ハフマン木の生成

ここでは前述のハフマン符号生成法に基づき、概ね以下のような手順でハフマン木を生成する。

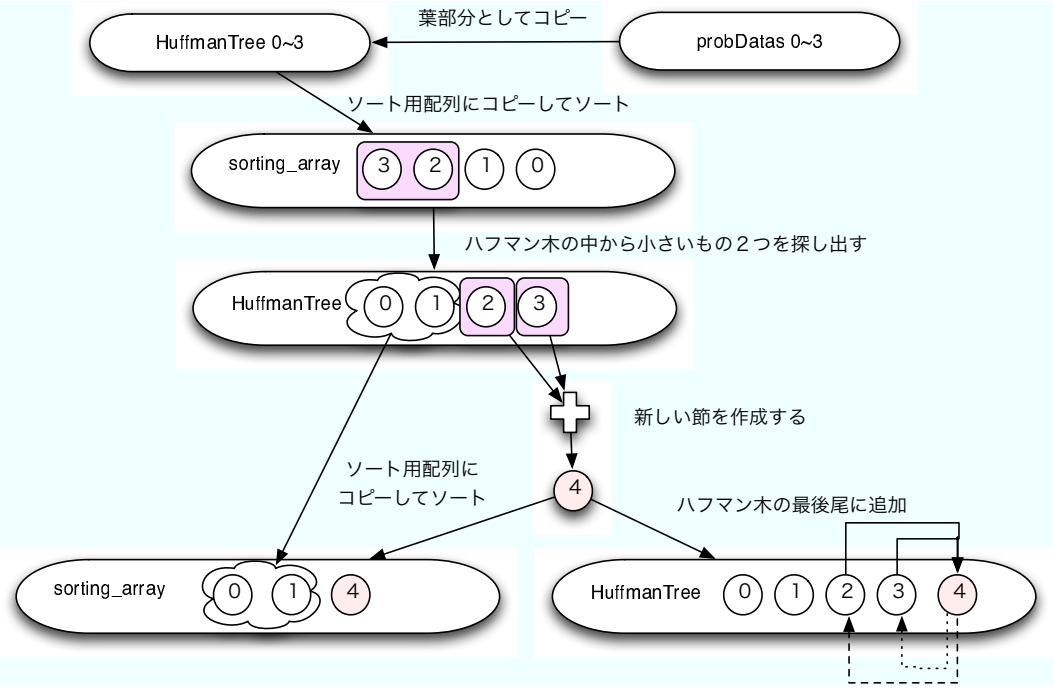

図 4　ハフマン木の生成手順

図をみても分かる通り、ハフマン木そのものはソートされないので、入力配列とハフマン木の葉部分のデータの順序は同じである。

この関数の返り値はハフマン木の根のインデックスである。

ソースコード 8　createHuffmanTree

```
1  int createHuffmanTree(struct TreeNode * HuffmanTree, struct Data * probDatas,
       int length){
2          int i;
3          int freeNode=length;
4          int nowLength=length;
5          int root=0;
6
7          //! ハフマン木の葉を作成
8          for(i=0; i<length; i++) HuffmanTree[i].node=probDatas[i];
9
10         //! 葉をソート
11         struct Data sorting_array[length];
12         for(i=0;i<length;i++) sorting_array[i] = HuffmanTree[i].node;
13         BubbleSort(sorting_array,length);
14
15         if(debug){printf("[createHuffmanTree] \n"); ShowData(sorting_array,
               length);}
16
17         //! ハフマン木を作成
18         while(nowLength > 0){
19                 if(debug) printf("\n sort array len %d free len is %d\n",
                       nowLength,freeNode);
20
21                 //! 生起確率の小さい 2つのノードを繋げて、Huffman 木の新しい節を作成
22                 struct Data newData;
23                 //!- 新しい節のdata は他の data とかぶらないようになってればおけ
24                 newData.data=(sorting_array[0].data+sorting_array[1].data)+
                       length;
25                 newData.priority=((double)sorting_array[0].priority+(double)
                       sorting_array[1].priority);
26                 //!- 親と子を初期化しとく(区別するために負の値で)
27                 //newData.parent = newData.left = newData.right = INT_MIN;
28                 HuffmanTree[freeNode].node = newData;
29
30                 if(debug)
```

```
31              printf("new␣node␣%d(%f)␣<-␣%d(%f)␣+␣%d(%f)\n",
32                              newData.data,newData.priority
33                              ,sorting_array[0].data,sorting_array[0].priority
34                              ,sorting_array[1].data,sorting_array[1].priority
                                    );
35
36
37              //! ハフマン木の中から左右の子を探し出す。
38              //!-
                    sorting_array と HuffmanTree では要素の入っている位置が異なるため、

39              //!- 左右の子の
                    data と priority は分かっても HuffmanTree 内での位置がわからない。
40              if(debug) printf("␣search␣%d(%f)␣␣and␣%d(%f)\n",sorting_array
                    [0].data,sorting_array[0].priority,sorting_array[1].data,
                    sorting_array[1].priority);
41              for(i=0; i<freeNode; i++){
42                      if(debug) printf("␣Huffman␣%d(%f)␣\n",HuffmanTree[i].
                            node.data,HuffmanTree[i].node.priority);
43                      if(HuffmanTree[i].node.data == sorting_array[0].data
44                                      && HuffmanTree[i].node.priority ==
                                            sorting_array[0].priority){
45                              //!- 左の子を登録、左の子の親は新しい節
46                              if(debug) printf("␣--␣left␣%d␣parent␣%d␣\n",i,
                                    freeNode);
47                              HuffmanTree[freeNode].left = i;
48                              HuffmanTree[i].parent = freeNode;
49                              continue;
50                      }
51                      if(HuffmanTree[i].node.data == sorting_array[1].data
52                                      && HuffmanTree[i].node.priority ==
                                            sorting_array[1].priority){
53                              //!- 右の子を登録、右の子の親は新しい節
54                              if(debug) printf("␣--␣right␣%d␣parent␣%d␣\n",i,
                                    freeNode);
55                              HuffmanTree[freeNode].right = i;
56                              HuffmanTree[i].parent = freeNode;
57                              continue;
58                      }
59              }freeNode++; root = freeNode;
```

```
60
61                  //! 新しい順序を計算
62                  for(i=2;i<nowLength;i++) sorting_array[i-2] = sorting_array[i];
63                  sorting_array[nowLength-2] = newData;
64                  if(nowLength <=2) break;//残りが二つなら並び替える必要なし
65                  nowLength--;
66                  BubbleSort(sorting_array,nowLength);
67          }
68          if(debug) {
69                  printf("\n\n[createHuffmanTree] HuffmanTree Array [root(length
                        ):%d] --------\n",root);
70                  for(i=0; i<root; i++) printf(" Huffman %d(%f) \n",HuffmanTree[i
                        ].node.data,HuffmanTree[i].node.priority);
71                  printf("[createHuffmanTree] End --------\n");
72          }
73          return root;
74  }
```

4.2.7 ハフマン符号表の生成

1. 葉から根まで辿り、自分がどうゆう経路を通っているか調べる
   ex) 自分が親から見て左→自分が親から見て右→親は根
2. 0 と 1 に置き換える
   ex) 1 → 0 →終わり
3. 葉から読み込んだので、根からの順番になるように逆順に読み込む
   ex) 符号は 01
4. 符号長を登録
   ex) 符号長は 2

ソースコード 9　createCodeMap

```
1  void createCodeMap(struct CodeMap * maps, struct TreeNode * HuffmanTree, int
       root, int dataTypeNum){
2          int i;
3          int data,node_number;
4          int bits[root];
5          char result[root];
6          int bitp,parent,bitp2;
7          int code_len;
```

```
8       int length;
9
10      //! 葉の数(データの種類数)だけ繰り返す
11      for(data=0;data<dataTypeNum;data++,code_len=0){
12              //!- 目的のデータが入っている位置を探す
13              for(i=0;i<dataTypeNum;i++){
14                      if(HuffmanTree[i].node.data == data){
15                              node_number = i;
16                              break;
17                      }
18              }
19              if(debug) printf("[createCodeMap]Start␣node␣num␣%d\n",
                    node_number);
20
21              //!- ハフマン木をもとにコードを生成
22              for(i=0;i<root;i++) bits[i]=3;
23              bitp=0;bitp2=0;
24              do{
25                      //!-- 自分が親から見て左の子なら 1、右の子なら 0
26                      parent = HuffmanTree[node_number].parent;
27                      if(debug) printf("[createCodeMap]parent␣%d␣left␣␣%d␣\n",
                            parent, HuffmanTree[parent].left);
28                      bits[bitp++] = ( HuffmanTree[parent].left == node_number
                            ) ? 1 : 0;
29                      node_number=parent;
30
31                      if(i>root){printf("[Error]␣Failed␣to␣Calcurate␣Huffman␣
                            Code\n"); exit(-1);}
32              }while(node_number != root-1);
33
34              //! 符号長を設定
35              code_len = bitp;
36
37              do {
38                      //!-- バックトレース
39                      if (bits[--bitp] == 1) result[bitp2++]='1';
40                      else result[bitp2++]='0';
41              } while (bitp > 0);
42              result[bitp2]='\0';
43
```

```
44                  //!- マップ配列に格納
45                  maps[data].data = data;
46                  maps[data].length = code_len;
47                  strcpy(maps[data].code,result);
48                  if(debug) printf("[createCodeMap]%d␣:␣%d␣->␣%s␣(%d)\n"
49                                          ,data,maps[data].data,maps[data].code,maps
                                                [data].length);
50          }
51  }
```

4.2.8 エントロピーの計算

ソースコード 10　calcEntoropy

```
1   double calcEntoropy(struct Data* probability,int length){
2           int i;
3           double entoropy=0.0;
4
5           //! entoropy = sum -P*lg(P)
6           for(i=0;i<length; i++){
7                   if(probability[i].priority != 0.0){//log(0)は計算できない
8                           entoropy += -(probability[i].priority * lg(probability[i
                                ].priority) );
9                   }else entoropy += 0;
10          }
11
12          return entoropy;
13  }
```

4.2.9 平均符号長の計算

ソースコード 11　calcAvarageCodeLength

```
1   double calcAvarageCodeLength(struct CodeMap * maps, struct Data * input_datas,
        int length){
2           int i;
3           double acl=0.0;
4
5           for(i=0;i<length;i++) acl += maps[i].length*input_datas[i].priority;
6
```

```
7         return acl;
8 }
```

5 Exercise 1

次のような生起確率が与えられたとき、自作プログラムで符号化しなさい。

表 1　生起確率の表

gray scale	0	1	2	3	4	5	6	7
probability (a)	0.19	0.25	0.21	0.16	0.08	0.06	0.03	0.02
probability (b)	0.09	0.13	0.15	0.10	0.14	0.12	0.11	0.16
probability (c)	0.13	0.12	0.13	0.13	0.12	0.12	0.12	0.13
probability (c)	0.07	0.11	0.08	0.04	0.50	0.05	0.06	0.09

5.1 probabirity(a)

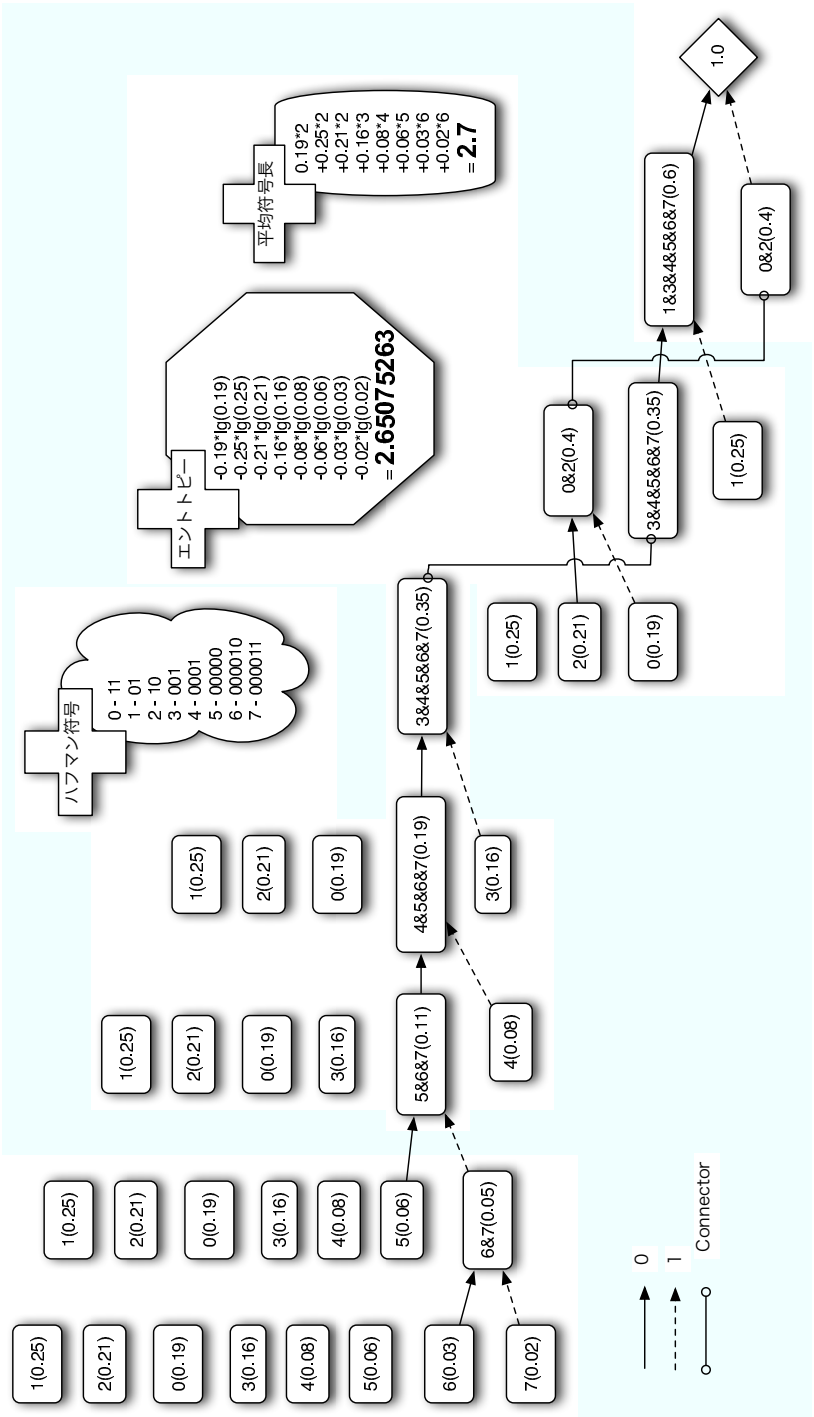

図 5　probabirity (a) 手計算

プログラムの実行結果は以下のようになった。

ハフマン木の「-2147483648」は初期値であり、「設定されてない」という意味である。

ソースコード 12　probabirity(a) 実行結果 抜粋

```
HuffmanTree Array [root(length):15] --------
 Huffman[0] 0(0.190000) parent 12 left -2147483648 right -2147483648
 Huffman[1] 1(0.250000) parent 13 left -2147483648 right -2147483648
 Huffman[2] 2(0.210000) parent 12 left -2147483648 right -2147483648
 Huffman[3] 3(0.160000) parent 11 left -2147483648 right -2147483648
 Huffman[4] 4(0.080000) parent 10 left -2147483648 right -2147483648
 Huffman[5] 5(0.060000) parent 9 left -2147483648 right -2147483648
 Huffman[6] 6(0.030000) parent 8 left -2147483648 right -2147483648
 Huffman[7] 7(0.020000) parent 8 left -2147483648 right -2147483648
 Huffman[8] 21(0.050000) parent 9 left 7 right 6
 Huffman[9] 34(0.110000) parent 10 left 8 right 5
 Huffman[10] 46(0.190000) parent 11 left 4 right 9
 Huffman[11] 57(0.350000) parent 13 left 3 right 10
 Huffman[12] 10(0.400000) parent 14 left 0 right 2
 Huffman[13] 66(0.600000) parent 14 left 1 right 11
 Huffman[14] 84(1.000000) parent -2147483648 left 12 right 13

Result--------
etoropy = 2.650753
0 (0.190000) -> 11
1 (0.250000) -> 01
2 (0.210000) -> 10
3 (0.160000) -> 001
4 (0.080000) -> 0001
5 (0.060000) -> 00000
6 (0.030000) -> 000010
7 (0.020000) -> 000011
Avarage Code Length = 2.700000
```

double 型の精度の問題からエントロピーの値が丸められているが、正しく計算出来ていることがわかる。

5.2 probabirity(b)

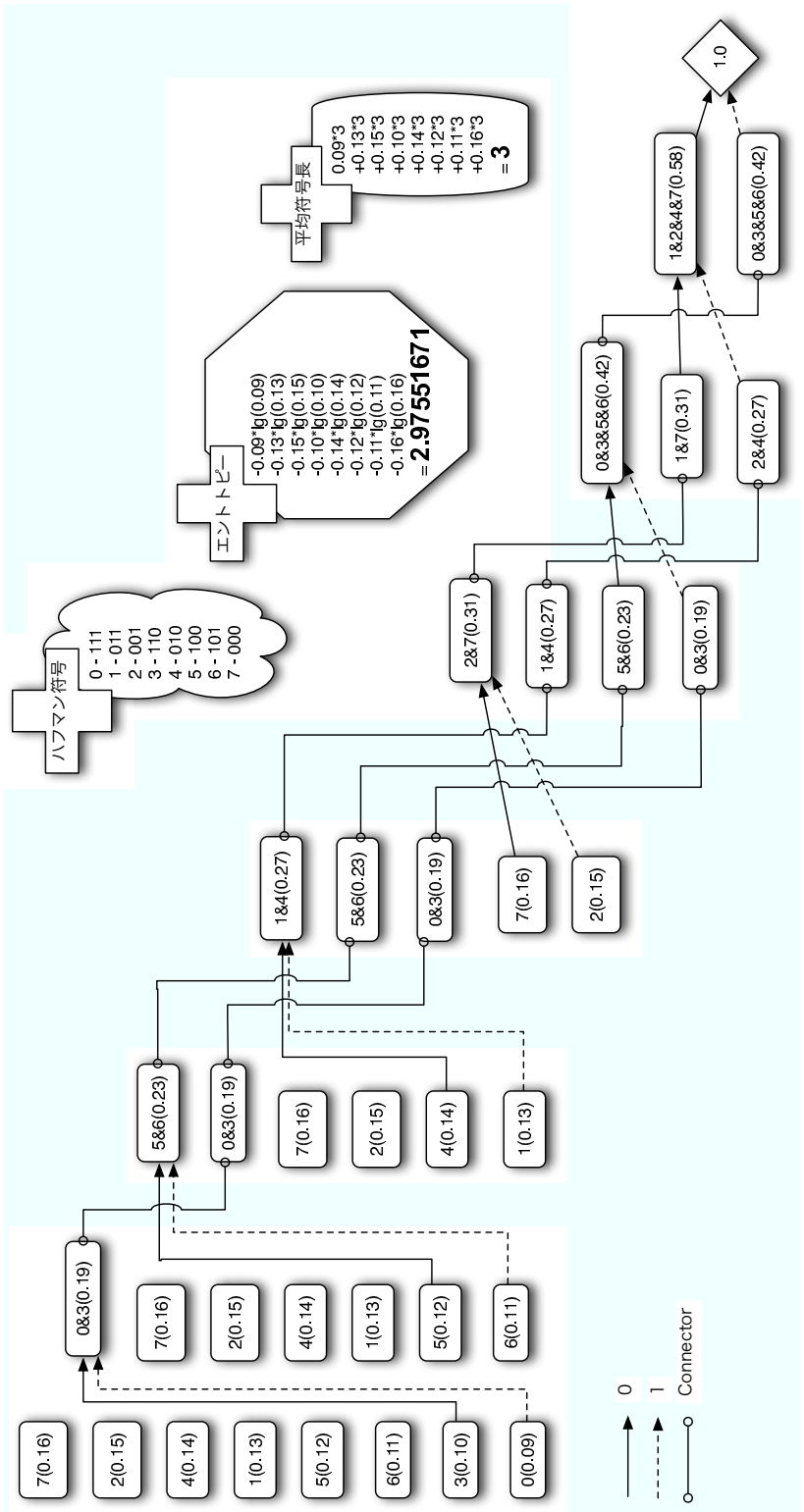

図 6 probabirity (b) 手計算

ソースコード 13　probabirity(b) 実行結果 抜粋

```
HuffmanTree Array [root(length):15] --------
 Huffman[0] 0(0.090000) parent 8 left -2147483648 right -2147483648
 Huffman[1] 1(0.130000) parent 10 left -2147483648 right -2147483648
 Huffman[2] 2(0.150000) parent 11 left -2147483648 right -2147483648
 Huffman[3] 3(0.100000) parent 8 left -2147483648 right -2147483648
 Huffman[4] 4(0.140000) parent 10 left -2147483648 right -2147483648
 Huffman[5] 5(0.120000) parent 9 left -2147483648 right -2147483648
 Huffman[6] 6(0.110000) parent 9 left -2147483648 right -2147483648
 Huffman[7] 7(0.160000) parent 11 left -2147483648 right -2147483648
 Huffman[8] 11(0.190000) parent 12 left 0 right 3
 Huffman[9] 19(0.230000) parent 12 left 6 right 5
 Huffman[10] 13(0.270000) parent 13 left 1 right 4
 Huffman[11] 17(0.310000) parent 13 left 2 right 7
 Huffman[12] 38(0.420000) parent 14 left 8 right 9
 Huffman[13] 38(0.580000) parent 14 left 10 right 11
 Huffman[14] 84(1.000000) parent -2147483648 left 12 right 13

Resulet--------
etoropy = 2.975517
0 (0.090000) -> 111
1 (0.130000) -> 011
2 (0.150000) -> 001
3 (0.100000) -> 110
4 (0.140000) -> 010
5 (0.120000) -> 100
6 (0.110000) -> 101
7 (0.160000) -> 000
Avarage Code Length = 3.000000
```

5.3 probabirity(c)

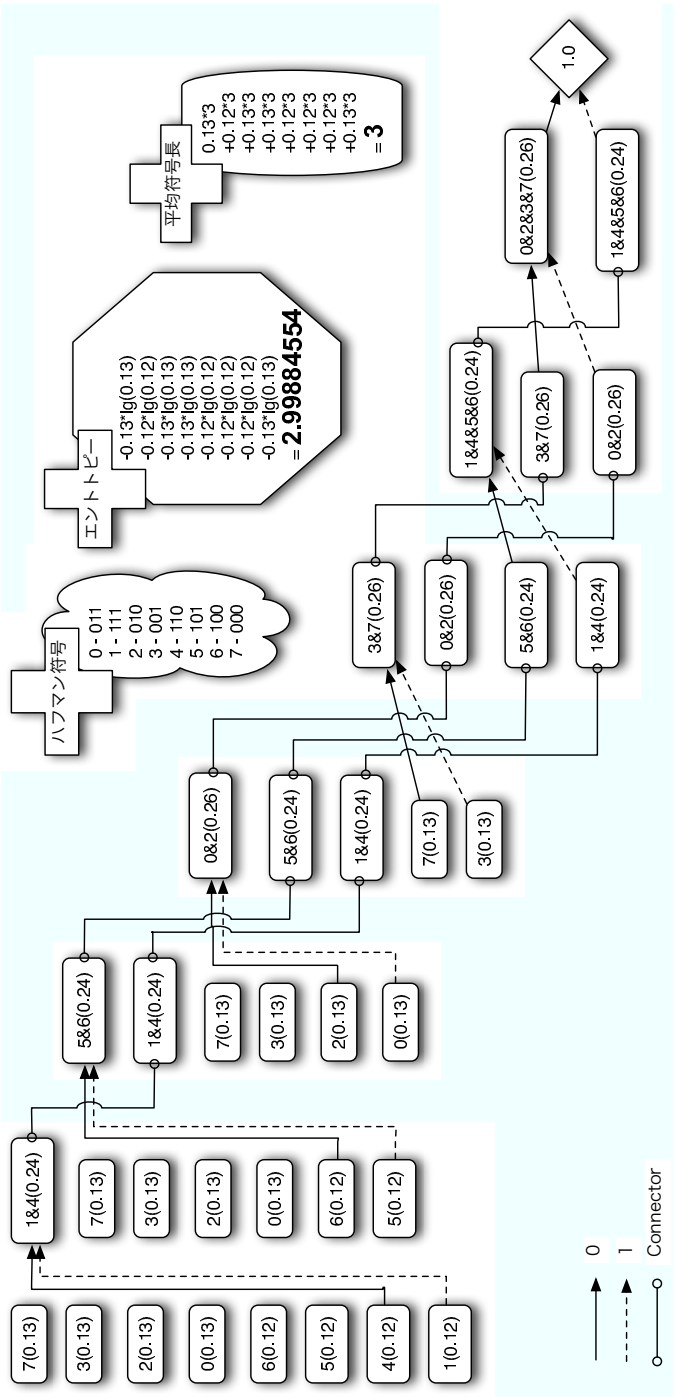

図 7　probabirity (c) 手計算

ソースコード 14　probabirity(c) 実行結果 抜粋

```
HuffmanTree Array [root(length):15] --------
 Huffman[0] 0(0.130000) parent 10 left -2147483648 right -2147483648
 Huffman[1] 1(0.120000) parent 8 left -2147483648 right -2147483648
 Huffman[2] 2(0.130000) parent 10 left -2147483648 right -2147483648
 Huffman[3] 3(0.130000) parent 11 left -2147483648 right -2147483648
 Huffman[4] 4(0.120000) parent 8 left -2147483648 right -2147483648
 Huffman[5] 5(0.120000) parent 9 left -2147483648 right -2147483648
 Huffman[6] 6(0.120000) parent 9 left -2147483648 right -2147483648
 Huffman[7] 7(0.130000) parent 11 left -2147483648 right -2147483648
 Huffman[8] 13(0.240000) parent 12 left 1 right 4
 Huffman[9] 19(0.240000) parent 12 left 5 right 6
 Huffman[10] 10(0.260000) parent 13 left 0 right 2
 Huffman[11] 18(0.260000) parent 13 left 3 right 7
 Huffman[12] 40(0.480000) parent 14 left 8 right 9
 Huffman[13] 36(0.520000) parent 14 left 10 right 11
 Huffman[14] 84(1.000000) parent -2147483648 left 12 right 13

Result--------
etoropy = 2.998845
0 (0.130000) -> 011
1 (0.120000) -> 111
2 (0.130000) -> 010
3 (0.130000) -> 001
4 (0.120000) -> 110
5 (0.120000) -> 101
6 (0.120000) -> 100
7 (0.130000) -> 000
Avarage Code Length = 3.000000
```

5.4 probabirity(d)

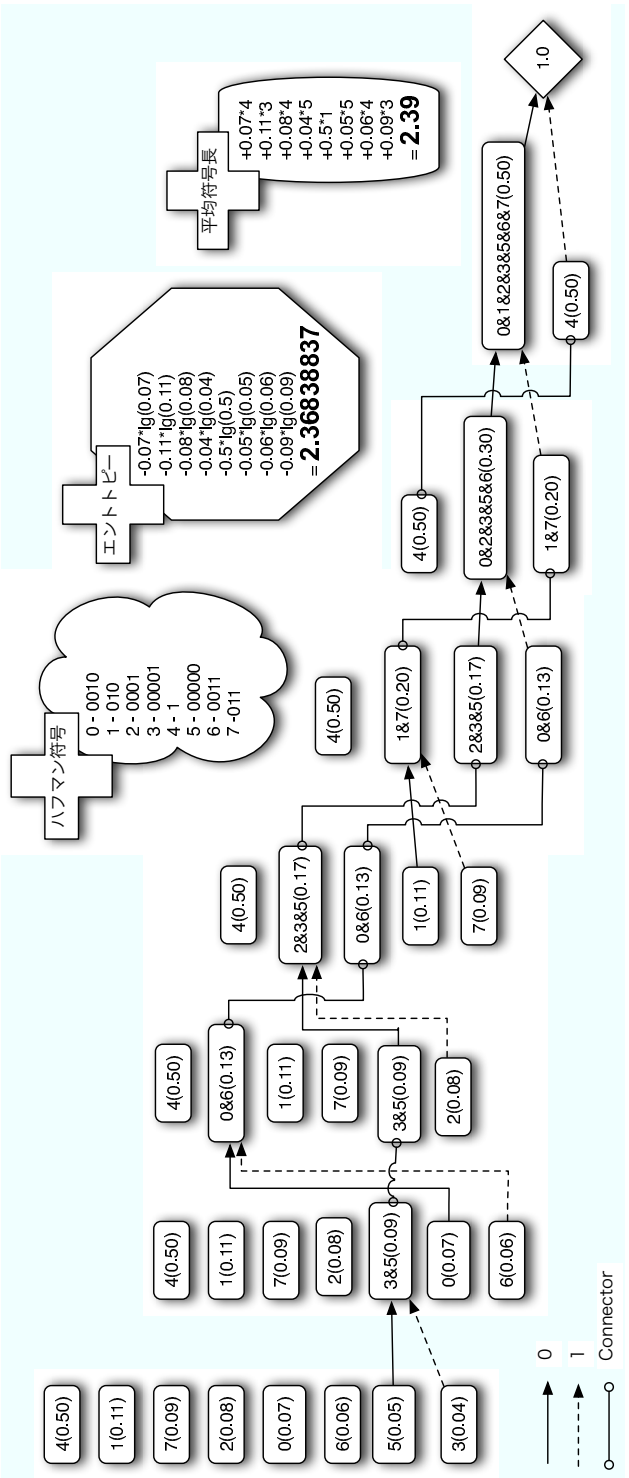

図 8　probabirity (d) 手計算

ソースコード 15　probabirity(d) 実行結果 抜粋

```
HuffmanTree Array [root(length):15] --------
 Huffman[0] 0(0.070000) parent 9 left -2147483648 right -2147483648
 Huffman[1] 1(0.110000) parent 11 left -2147483648 right -2147483648
 Huffman[2] 2(0.080000) parent 10 left -2147483648 right -2147483648
 Huffman[3] 3(0.040000) parent 8 left -2147483648 right -2147483648
 Huffman[4] 4(0.500000) parent 14 left -2147483648 right -2147483648
 Huffman[5] 5(0.050000) parent 8 left -2147483648 right -2147483648
 Huffman[6] 6(0.060000) parent 9 left -2147483648 right -2147483648
 Huffman[7] 7(0.090000) parent 11 left -2147483648 right -2147483648
 Huffman[8] 16(0.090000) parent 10 left 3 right 5
 Huffman[9] 14(0.130000) parent 12 left 6 right 0
 Huffman[10] 26(0.170000) parent 12 left 2 right 8
 Huffman[11] 16(0.200000) parent 13 left 7 right 1
 Huffman[12] 48(0.300000) parent 13 left 9 right 10
 Huffman[13] 72(0.500000) parent 14 left 11 right 12
 Huffman[14] 84(1.000000) parent -2147483648 left 4 right 13

Result--------
etoropy = 2.368388
0 (0.070000) -> 0010
1 (0.110000) -> 010
2 (0.080000) -> 0001
3 (0.040000) -> 00001
4 (0.500000) -> 1
5 (0.050000) -> 00000
6 (0.060000) -> 0011
7 (0.090000) -> 011
Avarage Code Length = 2.390000
```

6　Exercise 2

Exercise 1 の結果から、圧縮効率がよいのはどんなデータだと思われるか？

表 2　ハフマン符号化の結果

gray scale	生起確率の特徴	エントロピー	平均符号長
probability (a)	ランダム	2.650753	2.7
probability (b)	類似性がある	2.975517	3
probability (c)	同じものが複数ある	2.998845	3
probability (c)	1 つが突出している	2.368388	2.39

一部の記号が突出して生起確率が高く、他の多くの記号の生起確率が低い場合に効率が良くなる。ただし、同じ生起確率の記号が多くなると符号長が伸びてしまうので、生起確率の低い記号のグループとと生起確率の高い符号のグループが、各々のグループ内で生起確率に差があることも条件の一つだと思われる。

7　その他の関数

7.1　プログラムの実行方法

利用可能なオプション

h　　ヘルプの表示
d　　詳細情報の表示
n　　入力する記号の種類
i　　入力ファイル
g　　画像ファイルを入力（bmp/jpg/png/ppm/tiff）

ソースコード 16　usage

```
1  void usage(char *myname){
2  #ifdef OPEN_CV
3          printf("%s␣[hdig]\n",myname);
4  #else
5          printf("%s␣[hdi]\n",myname);
6  #endif
7
8          printf("\t␣-h␣:␣show␣this␣help\n");
9          printf("\t␣-d␣:␣show␣detail\n");
10
```

```
11          printf("\t␣-n␣<data␣num>␣:␣data␣type␣num␣(default␣%d)\n",MAX_CODE_LEN);
12          printf("\t␣-i␣<input␣File>␣:␣Input␣File\n");
13  #ifdef OPEN_CV
14          printf("\t␣-g␣:␣when␣Input␣File␣is␣Image\n");
15          printf("\t\t␣can␣use␣bmp/jpg/png/ppm/tiff␣file\n");
16  #else
17          printf("\t␣if␣you␣install␣OpenCV,␣you␣can␣reading␣image␣file\n");
18  #endif
19  }
```

7.2 log2 の計算を行う関数

C 言語は標準では $log_2(x)$ が計算できないので、関数を自作した。
対数の底の変換定理より

$$log_2(x) = \frac{log_{10}(x)}{log_{10}(2)} \tag{1}$$

ソースコード 17　lg

```
1  double lg(double x){
2          return log(x)/log(2);
3  }
```

7.3 データの表示

ソースコード 18　showData

```
1  void ShowData(struct Data* stream, int length)
2  {
3      int i;
4          for (i = 0; i < length; i++) printf("%d(%f)␣,␣", stream[i].data,stream[
               i].priority);
5          printf("\n");
6  }
```

8　画像データの読み込み実行例

画像は 8bit グレースケールで読み込む。このときプログラム中でグレースケールに変換しているため、入力画像はカラーでも構わない。

画像 (Lena 512x512 8bit color) を入力した時の実行結果（抜粋）を以下に示す。

図 9　入力画像

ソースコード 19　画像の読み込み 実行結果 抜粋

```
1 (前略)
2 Result--------
3 etoropy = 7.497905
4 (中略)
5 254 (0.000000) ->
      01110100110010111100011
6 255 (0.000000) ->
      01110100110010111100010
7 Avarage Code Length = 7.518276
```

生起確率 0 のものを含めても、エントロピーと平均符号長の差は 0.02 ポイントで収まっている。

参考文献

[1] [伊藤誠 講義・研究 ノート] ハフマン符号化による圧縮・解凍 http://www.ccad.sist.chukyo-u.ac.jp/ mito/syllabi/compaction/huffman/index.htm

[2] Wikipedia