2006 年 6 月 8 日

AWK による数合てゲームの作成

新潟工科大学 情報電子工学科 竹野茂治

1 はじめに

本稿では、数当てゲームを作成してみることにします。

プログラムの構造は、[2] のタイプ練習ソフトと同様で、

1. 課題の作成
2. ユーザが入力
3. それの正誤を判定し、誤なら 2. へ、正なら 4. へ
4. 結果の出力

という形です。よって、あまりおもしろみはありませんが、1 例として紹介します。

2 ゲームのルール

数当てゲームは、コンピュータが適当に決めた 4 つの数字の並びを当てるゲームですが、ルールは以下の通りとします。

1. 使う数字は 0 から $(L-1)$ までとし、数字列の長さは N とする (デフォルトは $L=10, N=4$)。数字列は、0 が先頭になることもあり得るが、同じ数字が複数回含まれることはない。
2. コンピュータは途中で課題数字列を変更することはない。
3. ユーザは、正解数字列と思われるものを入力すると、コンピュータがその判定結果を返す。間違いであれば、ユーザはその判定結果を元に再び正解と思われるものを入力する。これを正解になるまで繰り返す。
4. ユーザの入力も使う数字は 0 から $(L-1)$ までとし、同じ数字を複数回含んではいけない。
5. コンピュータが返す判定結果は、

- 数も位置も合っているものの総数 $(= H)$
- 位置は合っていないが数が課題数字列に含まれているものの個数 $(= B)$

の 2 つである。例えば、課題数字列が 0524 である場合、1234 は 1H1B と判定し、2450 は 0H4B と判定する。4H0B ならば正解となる。

一応、L や N を変えられるように作っておいて、問題の難易度を容易に変えられるようにしたいと思います。

また、ルール 4. ですが、この数当ての場合、使われていないことが確実に知られている数を知っていると問題がやさしくなりますので、このように設定しておきます。もちろん、これを変更できるようにプログラムを改良することも可能でしょう。

3 おおまかな構造

この数当てゲームのおおまかな構造は [2] とほぼ同じですが、今回はゲームの性格からして時間の計測は行わず、何回で問題が解けたかを成績とすることとします。ただし、時間を計測するようにプログラムを改良することも容易でしょう。

```
BEGIN{
  #（1）課題文を作成してゲーム開始
}
{ ## ここはキーボードからの 1 行の入力が済んだ後に実行される
  #（2）入力文字列を確認
  #（3）入力行と課題文とを比較
  #（4）違いがなければ終了し、
  #       違いがあれば判定結果を表示
}
END{
  #（5）途中でやめた場合はそのような表示をする
  #（6）正常終了した場合はそのような表示をする
}
```

後はこの (1)∼(6) の部分を実際に作っていきます。[2] と違うのは、主に (1) の課題作成部分と、(2),(3) の入力のチェックの部分です。

4 課題文の作成

今回も乱数を使って課題を作成しますが、タイプ練習ソフトと違うのは同じ数字が使えない、という点です。[2] で紹介した方法で 0 から $(L-1)$ までの乱数を作ることは容易にできますが、同じ数字を使わないようにするには、以下のような方法があります。

1. 新たにランダムにとった数が既に使われていれば、それを捨ててまた新たな乱数を取り直す
2. まだ使っていない数字の中からランダムに選ぶ

1. の方がプログラミングは楽ですが、取り直しが必要なので少し無駄なような気がしますし、選択肢が少なくなってくると取り直しをする回数が増えてきて、いつ取り直しが終わるかわからないとか、何回で終わるかの正確な評価ができない、という欠点があります。

2. は、例えば以下のようにすれば実現できます。

1. $a[0] = \cdots = a[L-1] = 0$ とする
2. $1 \sim L$ の乱数 x を生成し、$a[j] = 0$ であるような x 番目の $j = j_1$ を取り、$a[j_1] = 1$ とする。
3. $1 \sim (L-1)$ の乱数 y を生成し、$a[j] = 0$ であるような y 番目の $j = j_2$ を取り、$a[j_2] = 1$ とする。
4. これを N 回繰り返す。

$a[\,]$ が使用した数字かどうかを保存する配列で、$a[j] = 1$ の場合 j は使用した数字、としています。

乱数が一様乱数であると仮定すれば、最初の数字は L 個の数字からそれぞれ $1/L$ の確率で選び、次の数字は残りの $(L-1)$ 個の数字からそれぞれ $1/(L-1)$ の確率で選び、ということになります。これですべての数字列が確実に等確率で得られることになります。

上記の方法 (サブルーチン) は、以下のようにコード化されます。

```
function mkkadai(N,L,kadai,     j,k,x,m,a,L1)
{
    srand()
    for(j=0;j<L;j++) a[j]=0
    L1=L    # 使っていない数字の個数
```

```
    for(j=1;j<=N;j++){
        x=int(rand()*L1)+1   # 1 から L1 までの乱数
        m=0;
        for(k=0;k<L;k++){    # x 番目の使っていない数の検索
            if(a[k]==0) m++
            if(m==x) break
        }
        if(k>=L) return -1   # エラー
        if(a[k]!=0) return -2   # エラー
        a[k]=1
        kadai[j]=k    # j 番目の課題数字を k とする
        L1--
    }
    return 0
}
```

なお、このコードの中に、「エラー」と書いた文がありますが、今回の設計や考え方が正しければこの if 文の条件に合うことはないはずです。しかし、プログラムの設計段階では誰しも間違いをするものですから、そのようなときに、そのテスト用、デバッグ用にこのような if 文を入れておくと便利です。

「こうなるはずだ」とコーディングしていくと、少しのミスで無限ループやシステムエラーを引き起こしがちですが、このように「万が一エラーが起きた場合」のエラー処理のコードを入れておくことでそれを防ぐことができます。

こういう例外処理を一つ一つ埋めておくことが、「ちゃんとした」コードを作るコツの一つです。

5 チェック関数

5.1 入力チェック関数

今回は、入力されたもののチェック (2 節のルール 4.) と、結果判定 (2 節のルール 5.) の、2 つのチェック関数を作成します。

まず入力チェック関数は、すべての数が 0 から $(L-1)$ までであることと、同じ数が 2 度以上使っていないかを調べます。

同じものが使われていないかどうかは、少し遅いのですが、簡単な全数検索をやることにします。つまり、j 番目と k 番目 $(j<k)$ の数字の比較をすべての j と k について行います。

```
for(j=1;j<N;j++)
  for(k=j+1;k<=N;k++)
    if(input[j]==input[k]) return -1
```

すべての数が 0 から $(L-1)$ までであることのチェックは、

```
if(input[j]<0 || input[j]>=L) return -1
```

でよさそうな気がしますが、確実にするならば配列表か、正規表現を利用した方がいいでしょう。

実は、awk は「文字列」と「数字」の区別がなく (明確ではなく)、状況に応じて適当に変換される、という性質があります。例えば、「if$(A > B)$」の不等式の部分は、

- A,B が数字ならば、数字として比較
- A,B が文字列ならば、文字列としてその辞書式順序で比較 (ASCII コードで)
- 一方が数字、一方が文字列ならば、数字を文字列に変換して文字列として比較

という処理が行われます (cf. [3]、「2–1 パターン」)。よって、例えば

- 「`if("a">10000)`」は真 (10000 が文字列の “10000” になるので辞書式順序は “a” の方が大)
- 「`if(2>"12")`」は真 (“12” が文字列なので、2 が文字列の “2” に変換されて辞書式で比較される)

のようになってしまいます。よって、文字列が入りうる変数に対しては数字との比較を行うのはあまり安全であるとは言えません。

配列表を利用する方法とは、

- awk の配列が「連想配列」で、配列の添字に文字列が使える
- 定義されていない添字に対する配列の値は 0 となる

という性質を利用する方法です。例えば、次のようにします。

```
for(j=0;j<L;j++) check0[j]=1  # 0~(L-1) のチェックのための配列
for(j=1;j<=N;j++) if(check0[input[j]]==0) return -1
```

$check0[\ ]$ という配列は、0 から $(L-1)$ までの添字にのみ値が 1 と定義されているので、それ以外の数字、それ以外の文字列に対しては 0 となります。なお、awk には配列の添字が使われているかをチェックする in という演算子がありますので、それを使って

```
if(check0[input[j]]==0)
```

の部分は、

```
if(!(input[j] in check0))
```

と書き直すこともできます。この“!”は否定を意味しますから、「$input[j]$ が、配列 $check0$ の添字として定義されていなかったら」という意味になります。

正規表現を使う場合は、例えば以下のような感じになるでしょう。

```
if(input[j] !~ /^[0-9]$/ || input[j]>=L) return -1
```

正規表現の説明の前に、この文自体を少し説明します。

まず、A||B は A or B を意味します。つまり、「if(A||B) C」という文は、A が真、または B が真ならば C が実行されます。C 言語同様、A が真だった場合は B の評価自体が行われませんので、A||B と B||A とは厳密には少し違います。

A ~ /B/, A !~ /B/ はいずれもマッチングを調べる演算子で、

- A ~ /B/ は A が B にマッチする場合真
- A !~ /B/ は A が B にマッチしない場合真

となります。この // で囲まれている部分には正規表現を書きます。

正規表現とは、特別な意味を持つ記号を用いることにより、複数の文字列を表現するやり方で、主にマッチングで利用されます。正規表現は、Unix 上の他のツールである sed, (e)grep, perl などでも利用できますが、使える表現に少しずつ違い (拡張) があったりします。awk で使える正規表現については A 節にまとめますが、ここでは上に使われている正規表現について説明しましょう。

- ^ = 文字列の先頭を意味する (先頭の文字ではない)
- $ = 文字列の最後を意味する (最後の文字ではない)
- [ ] = [ ] 内のいずれか 1 文字を意味する

- [ ] 内の c_1-c_2 = (ASCII コードが) c_1 から c_2 までの文字全部を意味する

よって、[0-9] は、0 から 9 までのいずれか一文字、を意味し、^[0-9]$ によって、0 から 9 までのいずれか一文字からなる一文字だけの文字列、を意味することになります。結局、

```
if(input[j] !~ /^[0-9]$/ || input[j]>=L) return -1
```

によって、$input[j]$ が 0 から 9 までの いずれか 1 文字でなければ -1 を返す、そうでなくても $input[j]$ が L 以上ならば -1 を返す、ということになるわけです。

以上をまとめると、入力のチェック関数は以下のようになります。

```
function inputcheck(N,L,input,     j,k)
{
    for(j=1;j<=N;j++)
      if(input[j] !~ /^[0-9]$/ || input[j]>=L) return -1
    for(j=1;j<N;j++)
      for(k=j+1;k<=N;k++)
        if(input[j]==input[k]) return -2
    return 0
}
```

5.2 入力判定関数

次は、入力判定関数を作成します。入力数字列の配列 $input[\ ]$ と課題数字列の配列 $kadai[\ ]$ を比較するわけですが、次のように考えればいいでしょう。

1. $A = 0$, $H = 0$ とする (A は $A = B + H$ を意味する)。
2. $kadai[1]$ と同じ数字があるかどうかを $input[\ ]$ の中で探し、それがあれば A を一つ増やし、それが $input[1]$ ならば H も一つ増やす。
3. $kadai[2]$ と同じ数字があるかどうかを $input[\ ]$ の中で探し、それがあれば A を一つ増やし、それが $input[2]$ ならば H も一つ増やす。
4. これを N まで行なう。
5. $B = A - H$ とする。

この、「$kadai[j]$ と同じ数字があるかどうかを... 」 の部分は、以下のようなコードで書けます。

```
for(k=1;k<=N;k++)
  if(input[k]==kadai[j]){
      A++
      if(j==k) H++
      break
  }
```

一致するものがあれば A をカウントし、添字の j と k が等しい場合は場所も合っていることになるので、その場合に H をカウントしています。

なお、これは 1 から N までの全ての k に対して実行してもよいのですが、すでに配列の値の数字は重複しないことが保障されているので、もし一致した数字があればその後の k に対しては実行する必要がありません。よって、一致した場合に break 文で for 文を中断しています。break 文は、for 文や while 文などのループから途中で抜けるための命令で、break 文が呼び出されると、最も内側のループ一つから抜けだします。

あとは、これを全ての j に行えばいいので、結局次のような関数で判定ができます。

```
# 一致したら 1，そうでなければ 0 を返す
# B,H は大域変数としてそこに結果を代入する
function judgeinput(N,input,kadai,j,k,A)
{
    H=0  # 数も場所も合っているものの個数
    A=0  # 数が合っている（H も含む）ものの個数
    for(j=1;j<=N;j++)
      for(k=1;k<=N;k++)
        if(input[k]==kadai[j]){
            A++
            if(j==k) H++
            break
        }
    B=A-H
    if(H>=N) return 1
    else return 0
}
```

6　ソースコード全体

あとは、出力用に、配列を文字列に直す以下のような関数を用意しておきます。

```
function array2str(N,array,   j,s)
{
    s=""
    for(j=1;j<=N;j++) s=s array[j]
    return s
}
```

逆に、入力文字列を配列に直すのは、[2] で見たように split(str,input,"") でできます。

以上をまとめると以下のようになります。

```
BEGIN{
    Maxloop=20      # 制限回数
    if(N=="") N=4   # 数字列の長さ
    if(L=="") L=10  # 使用する数字の個数
    Times=1         # 回数を記録する変数
    win=0
    # (1) 課題文を作成してゲーム開始
    printf "数当てゲームを開始します。\n"
    printf "使用する数は 0~%d, 数字列の長さは %d です。\n",L-1,N
    printf "判定結果の意味は以下の通りです。\n"
    printf "  H=数も位置も合ってる数\n"
    printf "  B=数のみ合ってる数\n"
    if(mkkadai(N,L,kadai)<0){ printf "Error.\n"; errorexit=1; exit }
    else
       printf "準備 OK です。%d 個の数の並びを入力してください。\n",N
    printf "[%d] ",Times
}
{ ## ここはキーボードからの 1 行の入力が済んだ後に実行される
    # (2) 入力文字列を確認
    gsub(/[ \t]/,"")
    if(split($0,input,"")!=N || inputcheck(N,L,input)<0){
        printf "入力エラー\n"
        printf "[%d] ",Times   # 改めて入力をうながす
        next
    }
    # (3) 入力行と課題文とを比較
    ret=judgeinput(N,input,kadai)
    # (4) 違いがなければ終了し、
    #     違いがあれば判定結果を表示
    if(ret==1){ win=1; exit }
```

```
    else
      printf "%dH%dB\n",H,B

    if(Times>=Maxloop){ win=-1; exit }
    else printf "[%d] ",++Times
}
END{
    if(errorexit) exit
    # (5) 途中でやめた場合はそのような表示をする
    # (6) 正常終了した場合はそのような表示をする
    if(win==0) printf "途中であきらめましたね。\n"
    else if(win==-1)
      printf "制限回数 (%d) 内に回答できませんでした。\n",Maxloop
    else printf "おめでとう。正解です。\n"

    printf "回数 %d、正解 %s\n",Times,array2str(N,kadai)
}

# 配列 --> 文字列
function array2str(N,array,        j,s)
{
    s=""
    for(j=1;j<=N;j++) s=s array[j]
    return s
}

# 入力チェック関数
function inputcheck(N,L,input,     j,k)
{
    for(j=1;j<=N;j++)
      if(input[j] !~ /^[0-9]$/ || input[j]>=L) return -1
    for(j=1;j<N;j++)
      for(k=j+1;k<=N;k++)
        if(input[j]==input[k]) return -2
    return 0
}

# 回答チェック
# 一致したら 1, そうでなければ 0 を返す
# B,H は大域変数としてそこに結果を代入する
function judgeinput(N,input,kadai,        j,k,A)
{
```

```
    H=0  # 数も場所も合っているものの個数
    A=0  # 数が合っている（H も含む）ものの個数
    for(j=1;j<=N;j++)
      for(k=1;k<=N;k++)
        if(input[k]==kadai[j]){
            A++
            if(j==k) H++
            break
        }
    B=A-H
    if(H>=N) return 1
    else return 0
}

# 課題の作成
function mkkadai(N,L,kadai,     j,k,x,m,a,L1)
{
    srand()
    for(j=0;j<L;j++) a[j]=0
    L1=L   # 使っていない数字の個数
    for(j=1;j<=N;j++){
        x=int(rand()*L1)+1   # 1 から L1 までの乱数
        m=0;
        for(k=0;k<L;k++){    # x 番目の使っていない数の検索
            if(a[k]==0) m++
            if(m==x) break
        }
        if(k>=L) return -1   # エラー
        if(a[k]!=0) return -2   # エラー
        a[k]=1
        kadai[j]=k    # j 番目の課題数字を k とする
        L1--
    }
    return 0
}
```

ここでは一応 Maxloop$(=20)$ 回を制限回数とし、これ以上かかったら強制終了としています。

また、最初の BEGIN ブロックで、N, L を、

```
if(N=="") N=4   # 数字列の長さ
```

```
if(L=="") L=10  # 使用する数字の個数
```

としていますが、これは実行時に N, L を容易に変更できるようにするための仕組みです。通常は、最初は N, L の値は定義されていませんから、空文字列として "" という値を取りますから、それぞれデフォルトの 4, 10 が設定されますが、awk のコマンドラインオプションで -v オプションを使って、

```
awk -v N=3 -v L=5 -f kazuate.awk
```

のように変数の値の初期値を設定すれば、スクリプトを書きかえずに変数の値を変更できます。

なお、同じように例えば Maxloop を変えようとして

```
awk -v Maxloop=100 -f kazuate.awk
```

としても、$Maxloop$ は BEGIN ブロックで強制的に初期値が設定されますから、このようにして変更することはできません。awk の -v オプションで値を変えたい変数に対しては、上の N, L のように "" との比較をチェックするように書いておく必要があります。

7 最後に

全体の構造は、[2] とほぼ変わりないので、その点ではおもしろみはないですが、

- 正規表現の使い方
- ランダムな順列の作り方
- 文字列との比較の場合の注意
- コマンドラインオプションとして変数を設定する方法

などを追加できたので、多少それらの勉強にはなるのではないかと思います。

A 正規表現

AWK で正規表現で使用できるメタ文字 (特別な意味を持つ文字) は以下の通りです (r 等は正規表現、c 等は 1 文字を表す)。

1. \ : エスケープ用、あるいは特別な文字用 (後述)
2. ^ : 文字列の先頭部分 (先頭文字ではない)
3. $: 文字列の最後尾 (最後尾の文字ではない)
4. . : 任意の 1 文字
5. [$c_1c_2...$] : $c_1c_2...$ のいずれか 1 文字
6. [^$c_1c_2...$] : $c_1c_2...$ 以外のいずれか 1 文字
7. [ ] 内の c_1-c_2 : c_1 から c_2 までの全ての文字
8. r_1|r_2 : 正規表現 r_1 または r_2 にマッチする文字列
9. (r) : 正規表現 r をグループ化
10. (r)? または c? : r (または c) がないかまたは 1 つだけある文字列
11. (r)* または c* : r (または c) の 0 回以上の繰り返し
12. (r)+ または c+ : r (または c) の 1 回以上の繰り返し

グループ化を意味する ( ) は、後ろに ?,*,+ を伴ったり、| を使うときに使われます。例えば、以下の通りです。

- document(style|class) = "documentstyle" かまたは "documentclass"
- (re)?newcommand = "newcommand" かまたは "renewcommand"

* や + 等は、( ) の後ろ以外でも使用できます。

- 1[0-9]* = 1 で始まる任意の桁の数字
- <[^>]+> = > 以外の任意の 1 文字以上の文字列を < > で囲んだもの
- [a-zA-Z]+ = 1 文字以上のアルファベット文字列

エスケープ用に使われる \ は、上記のような特別な文字の意味をなくすのに使われます。例えば、

- \" = " 自体
- \\ = \ 自体
- main\((void)?\) = "main()" または "main(void)"

といった具合です。

またその他にも、C 言語でも用いられることがある以下のような特別な文字を表すのにも使われます。

1. \a：ベル
2. \b：バックスペース
3. \f：改ページ
4. \n：改行
5. \r：復帰
6. \t：タブ
7. \v：垂直タブ
8. *ooo*：8 進数 *ooo* で表される数字 (*ooo* は 0 から 7 までの 3 桁の数字)
9. \x*hh*：16 進数 *hh* で表される数字

参考文献

[1] 竹野茂治、「AWK に関する基礎知識」(2006)

[2] 竹野茂治、「AWK による簡単なタイプ練習ソフト」(2006)

[3] A.V. エイホ、B.W. カーニハン、P.J. ワインバーガー (足立高徳訳)、「プログラミング言語 AWK」、新紀元社 (2004) (元版は 1989)

[4] D.Dougherty、A.Robbins (福崎俊博訳)、「sed & awk プログラミング 改訂版」、オライリー・ジャパン (1997)

[5] 志村拓、鷲北賢、西村克信、「AWK を 256 倍使うための本」、アスキー出版 (1993)