情報科学 2006　久野クラス　＃11

久野　靖*

2007.1.19

はじめに

今回も残っている話題の落ち穂拾いということで「動的計画法」と「パターン認識」について扱います。

1　動的計画法

1.1　動的計画法とは

前にフィボナッチ数の計算をやったとき、最初に次のような再帰的定義を示し、それをそのまま再帰関数にしたのでは遅すぎる、という話をした。

$$fib(n) = \begin{cases} 1 & (n = 0 \; or \; n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

遅すぎる理由は、この定義通りだと 1 段階の再帰ごとに自分自身を 2 回呼び出し、同じパラメタに対する値を何回も重複して実行してしまうためだった。

それを防ぐ 1 つの方法は、たとえば配列 `fib[`i`]` を用意しておき、一度計算した値をそこに蓄えておけば、2 回目からは計算しないでそれを持って来るだけだから無駄な計算がなくなる、というものである。このように、一度実行した結果を覚えておくことを**メモ化** (memorize) と呼ぶことがある。

しかしそもそも、配列を使うのだったら、いちいち計算する代わりに、最大 30 番目までのフィボナッチ数だったら最初に順番に計算してしまい、あとはそれを参照するだけの方が分かりやすい。

```
int[] fib = new fib[31];
fib[0] = fib[1] = 1;
for(int i = 2; i < 31; ++i) { fib[i] = fib[i-1] + fib[i-2]; }
```

このように、ある問題に対して、その問題だけを解く代わりに小さい問題から順に全ての問題を答えを記録しつつ解くことで、再計算や堂々めぐりを避けて必要な解を求める手法のことを**動的計画法** (dynammic programming) と呼ぶ。動的計画法をうまく適用することで、他の方法では計算量が多すぎて大変な問題が効率よく計算できる場合がある。なお、これは単なる 1 つの手法であり、特別に動的でも特別にプログラミングでも何でもない。単に誰かがそういう名前をつけたというだけである。

1.2　コイン問題

フィボナッチ数ではありがた味が分からないと思うので、もう 1 つ別の例として「コイン問題」を考える。たとえば、日本ではコインの金額が 1、5、10、50、100、500 となっているので、おつりを出す時に迷うことはあまりないが、1、10、12、25、100 とかヘンな金額になっていたとすると、ある金額 M を指定したとき、最も少ない枚数の硬貨で済ませる方法を求めるのは簡単ではない。あなたなら、どのような方針でプログラムを作りますか?

素朴な方法として、次のようにしらみ潰しに調べることもできる。

*筑波大学大学院経営システム科学専攻

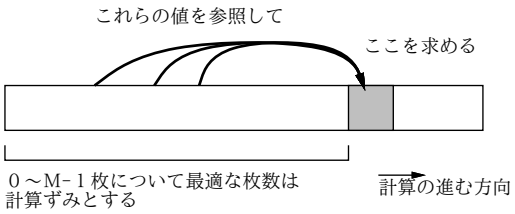

図 1: 動的計画法の考え方

```
int min = m+1; // M 円は 1 円玉 M 枚で支払えるので…
for(int c1 = 0; c1 < m; ++c1) {
  for(int c2 = 0; c2 < m; ++c2) {
    for(int c3 = 0; c3 < m; ++c3) {
      for(int c4 = 0; c4 < m; ++c4) {
        for(int c5 = 0; c5 < m; ++c5) {
          if(c1*1+c2*10+c3*12+c4*25+c5*100 == m && c1+c2+c3+c4+c5 < min) {
              //c1～c5 を記録
            }
          }
        }
      }
    }
  }
}
```

このアルゴリズムではコインの種類数が N とすると、$O(M^N)$ の時間計算量ということになり、明らかにひどく遅い。まあ、各コインの枚数上限をもうちょっとまじめに計算するとか、最後の 1 枚は残りの枚数から計算してみるとか、多少は効率化できるだろうが、それほど劇的に良くなるわけではない。

そこで動的計画法である。0～$M-1$ 円については最適なおつりの出し方を出す方法は分かっているものとして、M 円の最適なおつりを出すのには、「$M-1$ 円の場合に 1 円玉を追加」「$M-10$ 円の場合に 10 円玉を追加」「$M-12$ 円の場合に 12 円玉を追加」「$M-25$ 円の場合に 25 円玉を追加」「$M-100$ 円の場合に 100 円玉を追加」の 5 通りのうち、枚数が最も少ない場合を選べばよい (図 1)。なお、最初は 0 円だが 0 円の場合はもちろん 0 枚出すのが最適。

これを Java プログラムにしたものを示しておこう (最大金額を 1000 円としている)。

```
public class R11Sample1 {
  public static void main(String[] args) {
    int[] coins = new int[]{1, 10, 12, 25, 100};
    int[] total = new int[1001];
    for(int i = 1; i < total.length; ++i) {
      total[i] = total[i-1] + 1;                // 1 円玉を増やす場合
      for(int k = 1; k < coins.length; ++k) {
        int c = coins[k];                       // C 円玉を増やす場合
        if(i >= c && total[i-c] < total[i]) {   // こちらが有利ならば
          total[i] = total[i-c] + 1;            // 書き換える
        }
      }
    }
    int m = new Integer(args[0]).intValue();
    System.out.println(m + ": #coins = " + total[m]);
  }
}
```

動かした様子は次のとおり:

```
% java java R11Sample1 20
20: #coins = 2
% java R11Sample1 21
21: #coins = 3
% java R11Sample1 22
22: #coins = 2
% java R11Sample1 23
23: #coins = 3
% java R11Sample1 24
```

```
24: #coins = 2
% java R11Sample1 25
25: #coins = 1
%
```

なお、このままだと「最適の枚数」は分かるが、「何円玉を何枚か」は分からない。それを知るためには、`total[]` と同じ大きさのたとえば `amount[]` という配列を用意し、`total[i]` を更新するときに `amount[i]` にも「何円玉を使ったか」を記録しておく。この情報があれば、この金額を減らすことで「1 つ前の金額」に到達できるので、これを次々に続けることで使ったコインの金額をずべて挙げることができる。これを**トレースバック** (逆追跡) と呼ぶ。

演習 1 上の例題プログラムをそのまま打ち込んで動かせ。動いたら、コインの種類を変えてその場合もうまく計算されることを確認せよ。さらにできれば、枚数だけでなく「何円玉をどれだけ使ったか」が表示されるように直せ。表示の形式は自由でよい。たとえば「12 12 1」のようにコインの額がずらっと並んで表示されるだけでもよい。

1.3 ナップサック問題

今度は、次のような問題を考える (**ナップサック問題**と呼ばれる)。「サイズ S のナップサックに、いくつかの品物を入れる。品物毎に (s, v)(s は大きさ、v は価値) が決まっている。大きさの合計が S を超えない範囲で、価値の合計が最大となるような入れ方を求めよ。それぞれの品物の個数には制限がないものとする。」

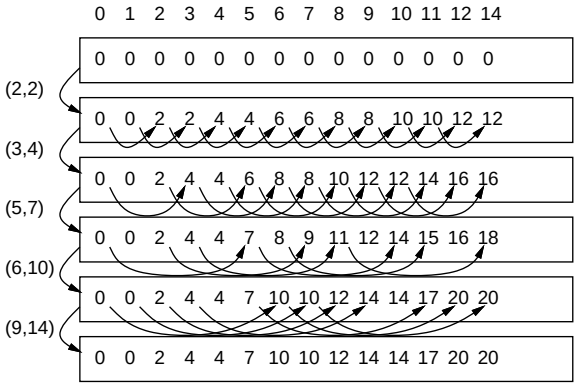

図 2: ナップサック問題

たとえば、品物が「$(2,2)(3,4)(5,7)(6,10)(9,14)$」の 5 種類だったとする。$S = 7$ のとき、たとえば大きさ 2 と 5 の品物を選ぶと価値合計は 9 になるが、それよりは大きさ 6 の品物 1 個を選んだ方が価値合計が 10 だから優っている。先に実行例を挙げておこう:

```
% java R11Sample2 7
7: total value = 10
% java R11Sample2 8
8: total value = 12
% java R11Sample2 9
9: total value = 14
% java R11Sample2 10
10: total value = 14
% java R11Sample2 11
11: total value = 17
%
```

これも図 2 のように配列を用いるが、今度はコイン問題と違って配列を更新して行く。すなわち、まず 1 種類も品物を入れない場合は全部 0 である。次に、$(2,2)$ の品物を考慮する場合は、`i = 2`〜`100` の範囲に渡って順に、`v = total[i-2] + 2` を計算し、現在の `total[i]` より価値が高いなら (最初は 0 だから当然高い) `v` の値に書き換える。次は $(3,4)$ の品物について、`i = 3`〜`100` の範囲に渡って順に、`v = total[i-3] + 4` を計算し、同様に書き換える。これを Java のコードにしたものは次のようになる。

```
public class R11Sample2 {
  public static void main(String[] args) {
    int[] sizes = new int[]{2, 3, 5, 6, 9};
    int[] value = new int[]{2, 4, 7, 10, 14};
    int[] total = new int[101];
```

```
    for(int i = 0; i < sizes.length; ++i) {
      for(int j = sizes[i]; j < total.length; ++j) {
        int v = total[j-sizes[i]] + value[i];
        if(v > total[j]) { total[j] = v; }
      }
    }
    int s = new Integer(args[0]).intValue();
    System.out.println(s + ": total value = " + total[s]);
  }
}
```

演習 2 この例題を打ち込んでそのまま動かせ。動いたらトレースバックを追加して品物の種類が表示されるようにせよ。

1.4 コインや品物の個数が限られる場合は…

上のコイン問題やナップサック問題では、コインや品物はいくつでも使えるものとしていた。しかし、たとえばこれらが限られている場合は、配列の使い方をちょっと違えることで対応できる。具体的には、先の問題で「それぞれの種類の品物が 1 個しかない」場合は、それらを順番に検討して行き、「それを入れない場合」(縦の矢線) と「それを入れる場合」(斜めの矢印) のうちで有利な方を選択すればよい。

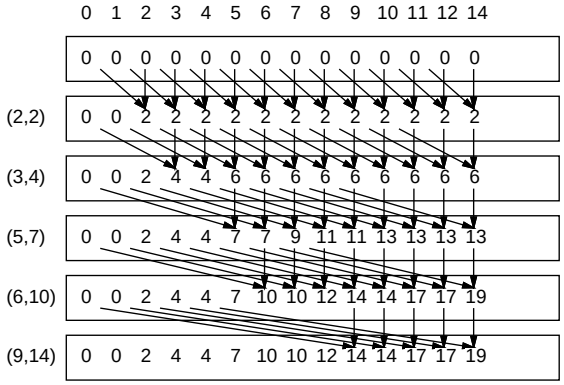

図 3: 資源が限られる場合

これを実際に実現する場合は、配列を 2 次元にしてもよいし、2 つの配列を「交替で」使うようにしてもよい。問題の形によっては、1 つ前だけでなくもっと前の状態を参照する必要があったり、3 次元の配列が必要になるような場合もある。

演習 3 コイン問題やナップサック問題を変形して、コインや品物の個数が限られている場合のプログラムを作れ。もちろん、最適の場合のコインや品物の種類が表示されるようにして欲しい。

1.5 文字列のアラインメント

次は別の種類の問題として、文字列の**アラインメント** (整合度) の計算を取り上げる。この問題は、文字列が「ぴったり同じ」ではない場合に「どれくらい似ているか」を判断するものである。

具体的には、列 s と t が違うとき、s のどこかの文字を別の文字に置き換えたり、削除したり、逆にどこかに文字を挿入したりして t に「直す」ことは (直す数を制限しないなら) 常にできるわけである。そこで、置き換え、および削除/挿入の「数」が少なくて済むほど s と t は「似ている」と考えることにして、そのような箇所に点数をつけて合計することで類似度を計算する。具体的にはここでは次のようにする。

- 削除/挿入 1 箇所につき -2。
- 文字の置き換え 1 箇所につき -1。
- それ以外 (文字が対応させられた場合) に $+2$。

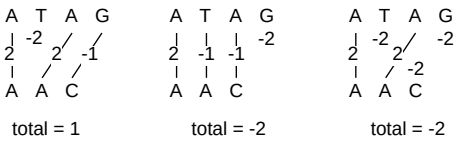

図 4: アラインメントの点数計算

しかし問題なのは、「どのように挿入/削除/置換するか」について複数の選択肢があることである。たとえば、「ATAG」と「AAC」の類似度を計算する際何を対応づけ何を削除/挿入するかで、図 4 のように複数の選択肢があることである。

そこで、「最も合計点数が良くなるように挿入/削除/置換したものとして点数を計算する」こととする (図 4 の場合でいえば一番左のが点数が一番よい)。で、どのように挿入/削除/置換するのが最も点数が高いかについては、動的計画法を用いる。

ところで、図 4 に出て来る「文字」が A と T と G と C しかないのは…もちろん、DNA の塩基配列 (ゲノム) はこの 4 文字の並びで表され、その解析にこういう方法を使っている、という話なのでした。

さて、アラインメントの計算に動的計画法をどのように使うかについて説明する。まず図 5(1) を見ていただきたい。横に (X 方向に) 文字列 s の 1 文字を 1 ます、縦に (Y 方向に) 文字列 t の 1 文字を 1 ますとするように、2 次元配列を用意する。ただし先頭は 1 文字ぶんずつ空けておき、そこに角は 0、縦横とも-2、-4、-6…を埋める。

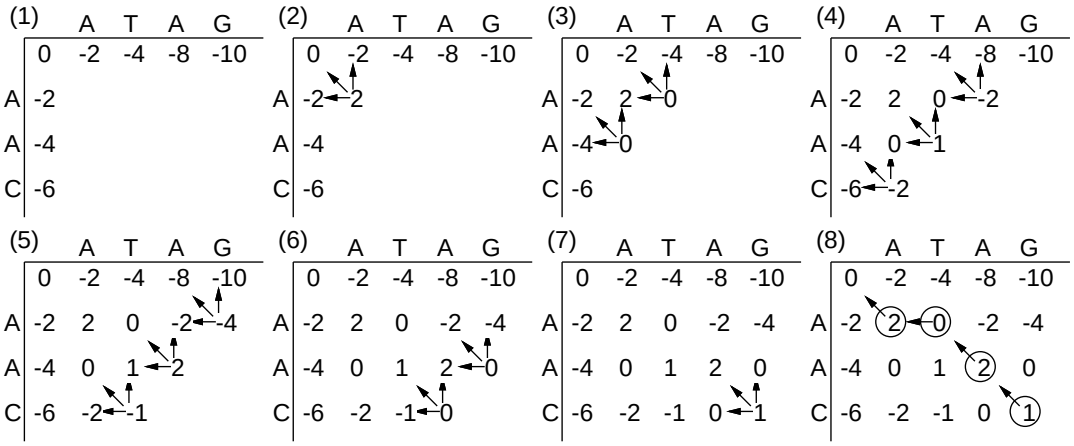

図 5: 動的計画法によるアラインメントの計算

この配列では X 方向/Y 方向に 1 ます進むごとに、対応する文字列を 1 文字ずつどう処理するかを決めて行くので、X 方向に (水平に) 進むことは文字列 s の位置だけ進む、つまり文字列 s から 1 文字削除することに対応し、Y 方向に (鉛直に) 進むことは文字列 t の位置だけ進む、つまり文字列 t から 1 文字削除することに対応する。左上隅の「0」はまだ何も処理していない (点数の増減のない) 状態を意味し、そこから左および下へ行くごとに 1 文字ずつ削除するので点数は-2 ずつ減って行く。

この (1) の状態から、空欄になっている配列の各位置の値を埋めて行く。ある位置を埋めるには、次の 3 つの場合を考慮する。

a. 1 つ上の値-2。文字列 t の当該位置を削除することに対応。

b. 1 つ左の値-2。文字列 s の当該位置を削除することに対応。

c. 斜め左上の値をもとに計算。この場合はさらに次の 3 つ。

c1. この欄の上と左の文字が一致→値を+2 する。

c2. この欄の上と左の文字が不一致で、文字の置き換えとして扱う→値を-1 する。

c3. 同様だが、両方の文字を削除として扱う→値を-4 する。

これらの場合のうち、最も合計値が大きくなるものを選んで採用する。最後の「両方を削除として扱う」は今回は常に置き換えより不利なので選ばれることはないが、文字の置き換えで「何を何に置き換えるか」によって値が違うようにした場合、置き換えよりも両方削除の方が有利になり選ばれる場合もあり得る。

上記の方針により、値を埋めて行く過程を図 5 の (2)〜(7) に示した。ただし埋めて行く順番はこの順でなくてもよい (ある空欄は、その上と左と左上が埋めてあるれば埋めることができるので)。このようにして全部埋め終わった (7) において、右下隅の値 (この場合は「1」が 2 つの文字列全部を対応づけ終わったところでの点数の最大値となる。実際にどのように削除/挿入/置換を行ったかは、埋める時に a、b、c1〜c3 のどれを採用したかを覚えておけば (8) のようにトレースバックによって復元できる。

演習 4 ここで説明した方式により、2 つの文字列の類似度を計算する Java プログラムを作成せよ。できれば、トレースバックにより削除/挿入/置換の情報も表示できるとなおよい。

1.6 CYK 構文解析アルゴリズム

動的計画法は構文解析アルゴリズムにも使われている。ここでは **CYK**(Cocke-Younger-Kasami、これらは考案した人の名前) と呼ばれるアルゴリズムを見てみよう。このアルゴリズムは文脈自由文法のうち CNF(Chomskey Normal Form、チョムスキー標準形) と呼ばれる形のものに対する構文解析 (認識) を行える。CNF とは、すべての生成規則が次のいずれかの形であるものをいう。

$S \to \varepsilon$
$X \to AB$
$X \to a$

つまり ε 規則は出発記号 S に対してしか現れず、それ以外の規則はすべて右辺の長さが 2 また 1 で、2 の場合は 2 つとも非端記号、1 の場合は端記号に限られる、というものである。すべての文脈自由文法は同等言語を表すの CNF に変形可能であることが知られている (詳細は略) ので、CYK アルゴリズムは任意の文脈自由文法に対する認識 (解析) 器となる (そのようなものが存在するという証明となっている)。

では CYK について解説しよう。CNF では ε 規則はあっても 1 つだけであり、別に扱えば済むので残りの 2 種類の規則のみを考える。このアルゴリズムは動的計画のために、論理値型の 3 次元配列 $\mathtt{p}[j][i][c]$ を使用する。入力列の長さを n としたとき、これらの添字の範囲は次の通り:

- j — 0〜$n-1$。入力列の各文字 (端記号) の位置に対応。
- i — 1〜n。記号列の長さに対応。
- c — $T \cup N$(端記号と非端記号の集合) に順に番号をつけたものとして、その番号の範囲に対応。

そして $\mathtt{p}[j][i][c]$ は「入力列の位置 j から始まる長さ i の列 α について、$X \Rightarrow^* \alpha$ の時 $\mathtt{p}[j][i][c]$ が `true`」になるように印をつけていく。印をつけ終わった時に $\mathtt{p}[0][n][S]$ を見れば出発記号から入力列が導出できるかどうか分かることになる。

さて動的計画法だが、まず最初は $i = 1$ のものを処理する。これは $X \to a$ の形の規則を見て、入力列の j 番の位置が a_j であれば、$X \to a_j$ なるすべての X につて $\mathtt{p}[j][1][X]$ を真にすればよい。

以降は長さ $i = 2, 3, ..., n$ について順に処理していく。入力列の長さ i の部分列について、それをさらに 2 つの部分列 (それぞれの長さは k と $i-k$、$1 < k < i$ に分ける。i について順に処理していくのだから長さ k および $i-k$ については既に配列 p のデータは完成していることに注意。$X \to YZ$ なるすべての規則について、$\mathtt{p}[j][k][Y]$ も $\mathtt{p}[j+k][i-k][Z]$ も真であれば、入力列の位置 j〜$j+k-1$ は Y から導出でき、位置 $j+k$〜$j+i-1$ は Z から導出できるのだから、位置 j〜$j+i-1$ 全体が X から導出できるとわかるため、$\mathtt{p}[j][i][X]$ を真にする。これを k の範囲 1〜$i-1$、j の範囲 0〜$n-i$ に渡ってすべて行うことで、長さ i についてすべての場合をチェックできる。以上が CYK のあらましである。

たとえば、次の文法を考えてみよう (CNF になっていることに注意)。

$S \to ST$
$S \to a$
$T \to US$
$U \to b$

この文法について、入力列「*ababa*」を認識させる場合の配列 p の内容を図 6 に示す。

まず長さ 1($i = 1$) については、$S \Rightarrow a$、$U \Rightarrow b$、なので S と入力の a、U と入力の b に対応する箇所に印がけられる。次に長さ 2($i = 2$) のときは、$T \Rightarrow US$ で、U が $j = 1, i = 1$、S が $j = 2, i = 1$ で印がついているので、T の $j = 1, i = 2$

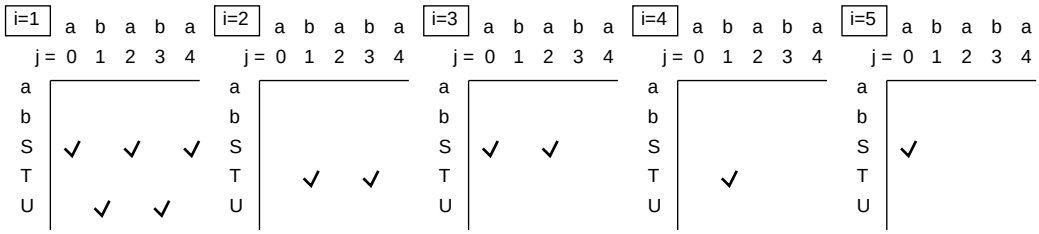

図 6: CYK による構文の認識

に印をつける。同様に U が $j=3, i=1$、S が $j=4, i=1$ で印がついているので、T の $j=3, i=2$ に印をつける。それ以外に長さ 2 で印がつくところはない。長さ $3(i=3)$ のときは、$S \Rightarrow ST$ で、S が $j=0, i=1$、T が $j=1, i=2$ で印がついているので、S の $j=0, i=3$ に印をつける。同様に S が $j=2, i=1$、T が $j=3, i=2$ で印がついているので、S の $j=2, i=3$ に印をつける。それ以外に長さ 3 で印がつくところはない。以下同様にして長さ 4、5 に記入し、そして最後に、S の $j=0, i=5$ に印があるので、入力列 $ababa$ は文法にあてはまると分かる。[1]

これを実際に Java プログラムにしたものを示しておく。記号は a, b, S, T, U にそれぞれ $0, 1, 2, 3, 4$ の番号を割り当てた。また 2 種類の規則はそれぞれ別の 2 次元配列 p1, p2 に格納した。今更ですが変数に配列を入れるところはこれまでの説明だと「new int[][]{ ... }」と書くところだが、変数定義に続き初期化の時は型は変数定義を見れば分かるので「new int[][]」を省略してよい。

```
public class R11Sample3 {
  public static void main(String[] args) {
    String str = args[0], psym = "abSTU"; // a:0, b:1, S:2, T:3, U:4
    int n = str.length(), nsym = 5, start = 2; // S:start symbol
    int[] a = new int[n];
    for(int i = 0; i < n; ++i) { a[i] = str.charAt(i) - 'a'; }
    int[][] p1 = { {2,0}, {4,1} };        // S -> a, U -> b
    int[][] p2 = { {2,2,3}, {3,4,2} };    // S -> S T, T -> U S
    boolean[][][] p = new boolean[n][n+1][nsym];
    for(int i = 0; i < n; ++i) {
      for(int r = 0; r < p1.length; ++r) {
        int x = p1[r][0], y = p1[r][1];
        if(a[i] == y) { p[i][1][x] = true; }
      }
    }
    for(int i = 2; i <= n; ++i) {
      for(int k = 1; k < i; ++k) {
        for(int j = 0; j < n-i+1; ++j) {
          for(int r = 0; r < p2.length; ++r) {
            int x = p2[r][0], y = p2[r][1], z = p2[r][2];
            if(p[j][k][y] && p[j+k][i-k][z]) { p[j][i][x] = true; }
//          System.out.println(i+":"+j+":"+k+":"+x+":"+y+":"+z+p[j][i][x]);
          }
        }
      }
    }
    System.out.println(p[0][n][start]); // true/false(結果) 表示
    for(int i = 1; i <= n; ++i) {       // 以下は配列 p の表示
      System.out.println("---- " + i + " -----");
      System.out.println("  " + str);
      for(int s = 0; s < nsym; ++s) {
        System.out.print(psym.charAt(s) + " ");
        for(int j = 0; j < n; ++j) { System.out.print(p[j][i][s]?"t":"."); }
        System.out.println();
      }
    }
  }
}
```

結果の真偽値出力より後の部分は配列 p の内容を表示させてみるためのものである。今更ですが、「真偽値 ? X : Y」というのは真偽値が真のときは X、偽のときは Y を値とするという演算子 (if-else の演算子バージョン) である。

[1] なお、こうして見ると行列の端記号に対応する列に印がつくことがないので、本当は非端記号の分だけで十分なわけだが、記号に通しで番号をつけるものと考え全部入れてある。

ところで、前にやった再帰下降解析器は下向き解析だったが、この解析方法は端記号から始めて逆向きに規則を適用していく、**上向き解析**に相当することが分かる。アルゴリズムの効率であるが、コードを見れば分かる通り、入力列の長さを n としたとき、CYK の時間計算量は $O(n^3)$ となる。これは先の再帰下降解析の $O(n)$ に比べると遅いようだが、再帰下降解析では文法に強い制約があった。CYK は任意の文脈自由文法を (CNF に書き換えた上で) 解析できるアルゴリズムの中では高速なものとして知られている。ただし、本物のコンパイラでは巨大な (何万行もの) ソースプログラムを解析するため、$O(n)$ より計算量の大きいアルゴリズムでは実用になりにくい。プログラミング言語の文法自体もそのことを前提として ($O(n)$ で解析できるように) 設計されている。

演習 5 この例題を打ち込んでそのまま動かせ。動いたら文法を別のものに書き換えて正しく認識がなされることを確認せよ。

2 パターン認識

2.1 パターン認識とは

パターン認識 (pattern recognition) とは、さまざまなデータ (文字、数値、音、画像、…) の中にあるパターンを認識することを言う。パターン認識の中には、人間が得意とするがプログラムにやらせるのは大変だったり難しいことが多く含まれている。たとえば:

- 音声認識 — 「音」から「何を喋っているか」を抽出する。
- 画像認識 — 「画像」から「誰の写真か」「どこに何があるか」などを抽出する。
- 文字認識 — 「画像」や「書く時のデータ」から「何を書いているか」を抽出する。[2]

パターン認識でいう「パターン」とは 1 つのデータではなく、似通ったデータの集まりを意味している。従って、パターンへの「あてはまり」も YES/NO ではなく「どれくらい似ているか」を判断することになる。

パターン認識の難しさとして次のようなものが挙げられる。

- あるパターンに属するデータに大きな多様性がある。「あ」という文字でも実にいろいろな書き方があり得る。
- パターン認識に用いるデータにはノイズが含まれていることがある。紙に汚れがついていたりした状態で文字を読み取るなど。
- そもそもパターンをどのような枠組みで捉えたらよいかを判断することが難しい。音声認識であれば「音素」→「音節」→「単語」→「文」のような構造があり、一番下のパターン認識にも上の構造が参照され得る。「きょうの・てんきは・□れ・ですね。」

人間のパターン認識能力を真似て、人間の神経回路のような構造をプログラム上のデータ構造として構築し、それに「学習」を施してパターンを認識させる、という研究も多く行われている — ニューラルネットワーク、パーセプトロン、等。

ここでは学習させるようなものでなく、ある決まったモデルを想定してそれのパラメタを推定するようなものの例として、隠れマルコフモデルを取り上げる。

2.2 隠れマルコフモデル

マルコフ過程とは、いくつかの状態の間を遷移して行くが、次の状態は現在の状態のみに依存する (過去の履歴には依存しない) ようなモデル→非決定性有限オートマトンに決め打ちで遷移確率が割り当ててあるようなもの。

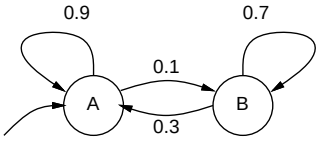

図 7: マルコフ過程の例

[2]「書く時のデータ (ストロークデータ)」は書き順や書く速さなども含まれるので、書いた結果の画像よりも多くの情報を認識に使うことができる。

たとえば図 7 は、最初状態 A から始まり、状態 A の次は確率 0.9 で状態 A、0.1 で状態 B、状態 B の次は確率 0.7 で状態 B、0.3 で状態 A になるというマルコフ過程を示している。これからたとえば次のような状態の系列が得られる。

```
AAAAABBBBBBBBBAAAAAAAAAAAAAAAAAAAAAAABBAAAAAAAAAABBBBBBBBBB
AAABBBBABBBBBAAAAAAAAAAAABBBBBBBAAAAAAAAAAAABBAAAAAABBBBBBBA
AAAAABBBBBAAAAAAAAAABBBBAAAAAAAAAAABBBAAAAAAAAAABAAABAAAAAA
```

さて、現実世界でマルコフ過程によってモデル化できることがらはいろいろあるが、その多くは「どの状態」という情報がそのまま観測できるのではなく、その状態によって「影響を受けた」情報が観測できる。ここでたとえば、状態ごとに確率的に出力が決まるものとしたモデルを**隠れマルコフモデル**と呼ぶ。なぜ「隠れ」かというと、状態そのものは隠されていて観測でないため。

たとえば「いかさまカジノ」すなわち、上記のマルコフ過程において、状態 A では普通のサイコロ (1〜6 が各 $\frac{1}{6}$ の確率で出現)、状態 B ではいかさまサイコロ (6 が $\frac{1}{2}$ の確率、1〜5 が $\frac{1}{10}$ の確率で出現) を使ってサイを振り、その出目の系列の情報が得られるという例を考えてみる。これは (各状態の出力が確率的に決まるため) 隠れマルコフモデルにあてはまる。この「観測されたデータ」から、「何回目はどちらの状態 (どちらのサイコロ) が使われたか」を求める、というのがパターン認識の問題に相当する。

ここでプログラムにするためのデータを少し整理しておく。

- `input`[t] — t 番目の入力 (観測されたデータ)
- `init`[i] — 状態 i が最初の状態である確率
- `out`[i][c] — 状態 i で c を出力する確率
- `trans`[i][j] — 状態 i から状態 j へ遷移する確率

ここで、`init`、`out`、`trans` がマルコフモデルを記述しており、`input` は観測データであり、これらを用いて状態列を推定する、というのが問題設定ということである。

なお、これとは別の問題として**推定問題**、つまりいくつかの出力列を与えてモデルのパラメタ (`init`、`out`、`trans`) を推定する、というのも重要な問題である (今回は扱わない)。

2.3 評価問題

状態列の推定の前に、類似の (しかしもう少し分かりやすい) 問題として、モデル M を前提として、ある観測データの列 w が与えられた時に、その列が出現する確率 $P(w|M)$ を求める、という問題を考える。これを**評価問題**と呼ぶ。

これも動的計画法を使うものとして、今度は `double` 値の配列 `a`[t][i] を「観測列 w の最初の $t-1$ 個を出力して状態 i に到達し、状態 i において t 番目のものを出力した確率」と考える。

まず最初にそれぞれの状態にいる確率は `init`[i] なのだから、そこで最初の文字を出力する確率は (それぞれの状態 i について) 次のようになる:

`a`[0][i] = `init`[i] × `out`[i][`input`[0]]

次の文字からは、それぞれの状態 j にいる確率は `a`[$t-1$][j] と状態 j からから状態 i に遷移する確率 `trans`[j][i] を掛けて合計することで状態 i にいる確率が求まるのでこれを `init`[i] の代わりに使えばよい。

`a`[t][i] = ($\sum_j$ `a`[$t-1$][j] × `trans`[j][i]) × `out`[i][`input`[t]]

そして一番最後に確率 $P(w|M)$ を求めるところは最後に各状態にいる確率の和ということになる (tt はデータの個数とする)。

r = $\sum_j$ `a`[$tt-1$][i]

単にこの通り順番に計算するだけだが (図 8)、このアルゴリズムは**前向きアルゴリズム** (forward algorithm) と呼ばれている。Java プログラムを示しておく。

```
public class R11Sample4 {
  public static void main(String[] args) {
    String str = args[0];
    int tt = str.length(), ns = 2;
    double[] init = {1.0, 0.0};
    double[][] trans = { {0.9, 0.1}, {0.3, 0.7} };
```

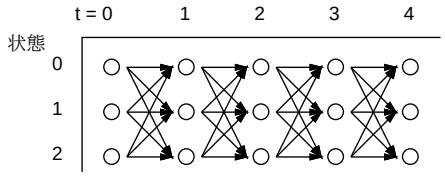

図 8: 前向きアルゴリズム

```
    double[][] out = { {1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0},
                       {0.1,0.1,0.1,0.1,0.1,0.5} };
    int[] input = new int[tt];
    for(int i = 0; i < tt; ++i) { input[i] = str.charAt(i) - '1'; }
    double[][] a = new double[tt][ns];
    for(int i = 0; i < ns; ++i) { a[0][i] = init[i]*out[i][input[0]]; }
    for(int t = 1; t < tt; ++t) {
      for(int i = 0; i < ns; ++i) {
        double s = 0.0;
        for(int j = 0; j < ns; ++j) { s += a[t-1][j]*trans[j][i]; }
        a[t][i] = s*out[i][input[t]];
      }
    }
    double r = 0.0;
    for(int i = 0; i < ns; ++i) { r += a[tt-1][i]; }
    System.out.println(r);
  }
}
```

2.4 復号化問題

さて、いよいよ隠れマルコフの状態を推定する問題に進もう。これは「元の状態列→確率的データ→元の状態列の復元」という形になるので、たとえばノイズのあるチャネルで情報を送信し、受け取った側が元の情報を復元することに相当することから**復号化問題**とも呼ばれる。

その考え方は簡単で、先の評価問題では各状態から来る確率の合計を取っていたのに対し、今度は「最も確率が大きい」1 つの状態を選んでそこから来たものとみなすだけである (図 9)。これは発案者の名前を取って **Viterbi のアルゴリズム**と呼ばれていて、実際に携帯電話の通信などに使われているとのこと。

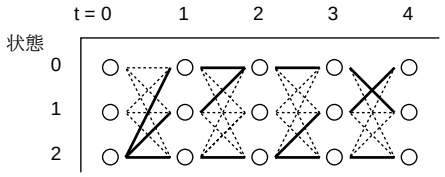

図 9: Viterbi のアルゴリズム

計算内容であるが、今度は配列名を d[t][i] として、最初の値は先と同じ。

d[0][i] = init[i] × out[i][input[0]]

2 番目以降は上述のように合計ではなく最大を選ぶ。

d[t][i] = (max_j d[$t-1$][j] × trans[j][i]) × out[i][input[t]]

最後の値も最大を選ぶ。

r = max_j d[$tt-1$][i]

実際には状態列が知りたいので、配列 prev[t][i] を用意しておき、各状態ごとに「最大値を取った j の値」を記録していく (トレースバック)。Java プログラムを次に示す。

```
public class R11Sample5 {
  public static void main(String[] args) {
    String str = args[0];
    int tt = str.length(), ns = 2;
    double[] init = {1.0, 0.0};
    double[][] trans = { {0.9, 0.1}, {0.3, 0.7} };
    double[][] out = { {1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0},
                       {0.1,0.1,0.1,0.1,0.1,0.5} };
    int[] input = new int[tt];
    for(int i = 0; i < tt; ++i) { input[i] = str.charAt(i) - '1'; }
    double[][] d = new double[tt][ns];
    int[][] prev = new int[tt][ns];
    for(int i = 0; i < ns; ++i) { d[0][i] = init[i]*out[i][input[0]]; }
    for(int t = 1; t < tt; ++t) {
      for(int i = 0; i < ns; ++i) {
        double max = 0.0; int k = 0;
        for(int j = 0; j < ns; ++j) {
          if(d[t-1][j]*trans[j][i] > max) { max=d[t-1][j]*trans[j][i]; k=j; }
        }
        d[t][i] = max * out[i][input[t]]; prev[t][i] = k;
      }
    }
    double max = 0.0; int k = 0;
    for(int i = 0; i < ns; ++i) {
      if(d[tt-1][i] > max) { max = d[tt-1][i]; k = i; }
    }
    ptrace(prev, k, tt-1); System.out.println(": " + max);
  }
  static void ptrace(int[][] prev, int k, int i) {
    if(i > 0) { ptrace(prev, prev[i][k], i-1); }
    System.out.print(k + " ");
  }
}
```

トレースバックの出力には (逆順に、つまり先頭から出したいため) 再帰関数を使っている (それほど大した問題ではないです)。これを動かした例を見てみよう。

```
% java R11Sample5 15234262664663624532
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 : 7.182554536972047E-17
%
```

確かに、「6」が多いあたりは「いかさまダイス」状態だと推定されている。

演習 6 「いかさまダイス」のデータを乱数で生成し、それを Viterbi のアルゴリズムで復号して、どれくらい正しく状態 (いかさまダイスか否か) が推定できるか検討せよ。

演習 7 図 10 のような「-」「0」「1」という 3 つの状態を持つマルコフモデルを考える。0 と 1 の並んだ列を与え、先頭と末尾と各文字の間に「-」を挿入し、それぞれの状態を 9 回繰り返したら次に行くようにして乱数でデータを生成する。生成したデータを Viterbi のアルゴリズムで復号してから、連続する「-」「0」「1」を 1 つにした後「-」を削除することで、元の 0 と 1 の列が正しく復元できるか試してみよ。うまく行ったら生成したデータに対してランダムに文字を別のものに取り替え、どの程度までノイズがあっても復元できるか実験してみよ。

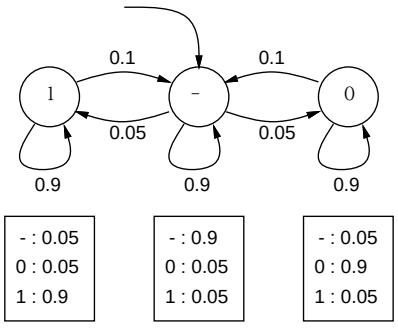

図 10: 練習問題のマルコフモデル

A　本日の課題 11A

「演習 1」〜「演習 3」のどれか 1 つについて、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 11A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

    Q1. 「動的計画法」の考え方は分かりましたか?

    Q2. 「隠れマルコフモデル」「評価問題」「推定問題」はどうでしたか?

    Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題はありません。

A　1/26 の当日課題 12A

次回 1/26 は講義はありませんが、TA さんが出勤して質問を受け付ける時間とします。しかしそれだけではあまりにも何もないので、出席用に「演習 4」をやって頂き、当日課題 12A として出して頂くことにしました。このプログラム作成において分からないことがあれば TA さんに解説してもらってください。どうしても他のものがやりたいなら、「演習 5〜演習 7」から選んでも構いません。また、「当日」課題ではありますが私が出張中で集計できないので「1/29 一杯」を期限とします。

1. Subject: は「Report 12A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 自分が作った「演習 4」のプログラムのソース。
4. 以下のアンケートの回答。

    Q1. アラインメントのプログラム作成はどれくらい大変でしたか。

    Q2. 動的計画法の考え方に慣れましたか。

    Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題はありません。