情報科学 2007　久野クラス　＃11

久野　靖*

2008.1.11

はじめに

あけましておめでとうございます。この授業も今回を入れてあと 4 回となりました。今回は以下の内容を取り上げます。

- 動的計画法
- パターン認識

そしてこれで Ruby による内容はすべておしまいです。次回は「さまざまな言語」の話ですが、その具体例として Java を取り上げ、その後の 2 回で GUI とかグラフィクスとか Ruby ではやりにくかった内容を Java を用いてやる予定です。

その前にですが、本日締切になっている第 9 回 (言語処理系) の問題解説があります。これも何をやってもいいような演習でしたが、全体的に例題のおもちゃ言語を「普通の言語」に近付ける方向でいくつか改良版を解説します。

1　演習問題解説: 言語処理系

1.1　字句解析の改良

例題で示した字句解析器はプログラムは 1 行ですべての要素は 1 文字だけというさぼったものだったが、「ふつうの」字句解析だとどうなるか見を見てみよう。具体的には次のように直してみた (というより全部書き直しというべきですが)。

- 入力はファイルから読み込み、複数行あってよい
- 空白や改行などは無視する
- 数値定数や名前は何文字あってもよい
- 制御構造等を表す特別な名前 (while、if 等) を扱う

前半の 2 つのために、`Scanner` オブジェクト中にファイルを保持して、次のトークンを取ろうとした時に行の終わりなら次の行を取るようにし、また空白等のときはその文字を飛ばすようにする。空白等を飛ばした結果行が終わりになることがあるのでそれをうまく扱う必要があるのに注意。

後半の 2 つのために、先頭が数字や英字の時は続く数字や英数字の列をまとめて探して取り出す。このとき、メソッド `peek` が返すものは数の時は「`'0'`」、英字のときは「`'a'`」という「印」だけにして、実際の文字列は別のメソッド `getstr` で取得するように直した。また、名前が「if」「while」「read」「print」の時は特別な印として「`'I'`」「`'W'`」「`'R'`」「`'P'`」を返す。従ってこれらの名前は変数としては使えない (こういう「予約された」名前のことを予約語という)。あと、「`==`」のような 2 文字演算子も使えるようにしてみた。ではコードを示そう。

```
class FileScanner
  def initialize(f)
    @file = open(f, 'r')
    @line = ''; @pos = 0; @cur = '$'; @str = ''; @lno = 0
    self.next
```

*筑波大学大学院経営システム科学専攻

```
  end
  def next()
    while @line != nil do
      if @pos >= @line.length then
        @line = @file.gets; @pos = 0; @lno += 1; next
      end
      c = @line[@pos..@pos]; l = @line.length
      if c==' ' || c=="\n" || c=="\r" || c=="\t" then
        @pos += 1; next
      elsif c>='0' && c<='9' then
        p = @pos+1
        while p < l && @line[p..p]>='0' && @line[p..p]<'9' do p += 1 end
        @cur = '0'; @str = @line[@pos..(p-1)]; @pos = p; return
      elsif c>='a' && c<='z' || c>='A' && c<='Z' then
        p = @pos+1
        while p < l &&
             (@line[p..p]>='0' && @line[p..p]<='9' ||
              @line[p..p]>='a' && @line[p..p]<='z' ||
              @line[p..p]>='A' && @line[p..p]<='Z') do p += 1 end
        @cur = 'a'; @str = @line[@pos..(p-1)]; @pos = p
        if @str == 'while' then @cur = 'W'
        elsif @str == 'if' then @cur = 'I'
        elsif @str == 'print' then @cur = 'P'
        elsif @str == 'read' then @cur = 'R'
        end
        return
      else
        @cur = c; @pos += 1
        if (c=='=' || c=='<' || c=='>' || c=='!') &&
            @pos < l && @line[@pos..@pos] == '=' then
           @cur = @cur + '='; @pos += 1
        end
        return
      end
    end
    @cur = '$'
  end
  def peek() return @cur end
  def getstr() return @str end
  def to_s() return "line #{@lno}: char #{@pos} in #{@line}" end
end
```

見て分かる通り、ほとんどの仕事は next の中でやっている。@line は行が入るがファイルの終わりになったら nil。ファイルの終わりでないなら、行末だったら行を読み先頭へ (next 文というのは、ループ内でループの先頭へ戻るための文。メソッド名としても next を使ってしまったのはいまいちだった…)。次に空白類を飛ばす場合も 1 文字飛ばしたら先頭へ戻る。この下は何らかの意味のある文字のところに来たので、それが数字なら数字列を取り出し、英字なら英字と数字の並びを取り出す。名前の場合は取り出した後でそれが予約語と一致したら名前の印を予約語の印にとりかえる。最後は 1 文字の演算子だが、ただし 2 文字演算子もちょっとだけ扱う。

以下で動かしてみるプログラムはおなじみ素数の列挙。

```
max = read; n = 2;
while(n <= max) {
  i = 2; sosu = 1;
  while(i < n) {
    if(n % i == 0) {sosu = 0};
    i = i + 1
  };
  if(sosu) print n;
  n = n + 1
}
```

これを新しい字句解析器で読んでみよう。

```
irb(main):002:0> sc = FileScanner.new('test.prog')  ←作る
=> #<FileScanner:0x81dc114 @cur="a", @pos=3, @line="max = read;
n = 2;\n", @lno=1, @file=#<File:test.prog>, @str="max">
irb(main):003:0> sc.peek   ←最は
=> "a"                     ←名前
irb(main):004:0> sc.getstr ←その文字列は
=> "max"                   ←「max」
irb(main):005:0> sc.next   ←次は
=> nil
irb(main):006:0> sc.peek
=> "="                     ←「=」
irb(main):007:0> sc.next   ←次は
=> nil
irb(main):008:0> sc.peek   ←予約語「read」
=> "R"
```

確かに複数文字の名前や予約語が扱えている。

1.2 比較演算や制御構文の増加

動作するものを増やすのは Node のサブクラスを増やすだけなので難しくない。比較演算については、Ruby では「真偽」が判定されるが、おもちゃ言語では 0 か 1 なので「条件 ? 真の時の値 : 偽の時の値」という 3 項演算子を使って 0 と 1 を返すようにしている。if、while、read、print 等は対応する動作を行うだけ。

```
$vars = {}
class Node
  def initialize(l=nil, r=nil) @left = l; @right = r; @op = '?' end
  def to_s() return '(' + @left.to_s + @op.to_s + @right.to_s + ')' end
  def getleft() return @left end
  def getright() return @right end
  def getop() return @op end
end
class Eq < Node
  def initialize(l, r) super; @op = '==' end
  def exec() return (@left.exec == @right.exec) ? 1 : 0 end
end
class Ne < Node
```

```
  def initialize(l, r) super; @op = '!=' end
  def exec() return (@left.exec != @right.exec) ? 1 : 0 end
end
class Gt < Node
  def initialize(l, r) super; @op = '>' end
  def exec() return (@left.exec > @right.exec) ? 1 : 0 end
end
class Ge < Node
  def initialize(l, r) super; @op = '>=' end
  def exec() return (@left.exec >= @right.exec) ? 1 : 0 end
end
class Lt < Node
  def initialize(l, r) super; @op = '<' end
  def exec() return (@left.exec < @right.exec) ? 1 : 0 end
end
class Le < Node
  def initialize(l, r) super; @op = '<=' end
  def exec() return (@left.exec <= @right.exec) ? 1 : 0 end
end
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
class Sub < Node
  def initialize(l, r) super; @op = '-' end
  def exec() return @left.exec - @right.exec end
end
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def exec() return @left.exec * @right.exec end
end
class Div < Node
  def initialize(l, r) super; @op = '/' end
  def exec() return @left.exec / @right.exec end
end
class Mod < Node
  def initialize(l, r) super; @op = '%' end
  def exec() return @left.exec % @right.exec end
end
class Lit < Node
  def initialize(v) super; @op = '#' end
  def exec() return @left end
end
class Var < Node
  def initialize(v) super; @op = '$' end
  def exec() return $vars[@left] end
end
class Assign < Node
```

```
  def initialize(l, r) super; @op = '=' end
  def exec v = @right.exec; $vars[@left.getleft] = v; return v end
end
class Seq < Node
  def initialize(l, r) super; @op = ';' end
  def exec() @left.exec; return @right.exec end
end
class If < Node
  def initialize(l, r) super; @op = 'I' end
  def exec() if @left.exec != 0 then @right.exec end end
end
class While < Node
  def initialize(l, r) super; @op = 'W' end
  def exec() while @left.exec != 0 do @right.exec end end
end
class Read < Node
  def initialize() super; @op = 'R' end
  def exec() print '> '; return gets.to_i end
end
class Print < Node
  def initialize(l) super; @op = 'P' end
  def exec() puts @left.exec end
end
class Noop < Node
  def initialize() end
  def exec() return 0 end
  def to_s() return '?' end
end
```

1.3 構文の改良

構文の改良といっても、文を増やしたりするのは大して難しいことはなく、やればいいだけ。ちょっと面倒なのは演算子の順位を分けることだが、これは「式」→「因子」の 2 レベルだったものを「代入式」→「条件式」→「加算式」→「乗算式」→「因子」とレベルを増やしてやるとできる。なぜこれでいいかというと、たとえば加算式の「+/-」で結ばれたそれぞれの部分式は乗算式なので、その中には乗算が入るがそれ以外の演算は (かっこで囲んでない限り) 入らない、だから加減算より乗除算が必ず優先される、というふうになるわけだ。なお、素数をやるため剰余演算も追加している。

```
def prog(sc)
  s = stat(sc); c = sc.peek
  if c == '$' || c == '}' then return s
  elsif c == ';' then sc.next; return Seq.new(s, prog(sc))
  else puts('STAT:' + sc.to_s); return Noop.new
  end
end
def stat(sc)
  c = sc.peek
  if c == '{' then
    sc.next; p = prog(sc)
```

```
    if sc.peek != '}' then puts('NO_}:' + sc.to_s); return Noop.new end
    sc.next; return p
  elsif c == 'W' then sc.next; e = expr(sc); return While.new(e, stat(sc))
  elsif c == 'I' then sc.next; e = expr(sc); return If.new(e, stat(sc))
  elsif c == 'P' then sc.next; e = expr(sc); return Print.new(e)
  else return expr(sc)
  end
end
def expr(sc)
  e = cmpexpr(sc); c = sc.peek
  if c == '=' then sc.next; return Assign.new(e, expr(sc))
  else return e
  end
end
def cmpexpr(sc)
  e = addexpr(sc); c = sc.peek
  if c == '<' then sc.next; return Lt.new(e, cmpexpr(sc))
  elsif c == '<=' then sc.next; return Le.new(e, cmpexpr(sc))
  elsif c == '>' then sc.next; return Gt.new(e, cmpexpr(sc))
  elsif c == '>=' then sc.next; return Ge.new(e, cmpexpr(sc))
  elsif c == '==' then sc.next; return Eq.new(e, cmpexpr(sc))
  elsif c == '!=' then sc.next; return Ne.new(e, cmpexpr(sc))
  else return e
  end
end
def addexpr(sc)
  e = mulexpr(sc); c = sc.peek
  if c == '+' then sc.next; return Add.new(e, addexpr(sc))
  elsif c == '-' then sc.next; return Div.new(e, addexpr(sc))
  else return e
  end
end
def mulexpr(sc)
  e = factor(sc); c = sc.peek
  if c == '*' then sc.next; return Add.new(e, mulexpr(sc))
  elsif c == '/' then sc.next; return Div.new(e, mulexpr(sc))
  elsif c == '%' then sc.next; return Mod.new(e, mulexpr(sc))
  else return e
  end
end
def factor(sc)
  c = sc.peek; sc.next
  if c == 'a' then return Var.new(sc.getstr)
  elsif c == '0' then return Lit.new(sc.getstr.to_i)
  elsif c == '(' then
    e = expr(sc)
    if sc.peek != ')' then puts('NO_):' + sc.to_s); return Noop.new end
    sc.next; return e
```

```
    elsif c == 'R' then return Read.new()
    else puts('FACTOR:' + sc.to_s); return Noop.new
    end
  end
```

read というのは文ではなく特別な因子になることに注意。たとえば「x = read + 1」なんていうものも書ける (へんな言語だ)。最後にこれで上の素数列挙を実行させてみる。

```
irb(main):009:0> tree = prog(FileScanner.new('test.prog'))
(表示は略)
irb(main):010:0> tree.to_s
=> "(((max$)=(R));(((n$)=(2#));(((n$)<=(max$))W(((i$)=(2#));(((sosu$)=(1#));
((((i$)<(n$))W(((((n$)%(i$))==(0#))I((sosu$)=(0#)));((i$)=((i$)+(1#)))));
(((sosu$)I((n$)R));((n$)=((n$)+(1#))))))))))"
irb(main):011:0> tree.exec
> 20
2
3
5
7
11
13
17
19
=> nil
```

確かにできているようだ。

2 動的計画法

2.1 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。

```
fib = Array.new(31)
fib[0] = fib[1] = 1
2.step(30) do |i| fib[i] = fib[i-1] + fib[i-2] end
```

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。

2.2 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?

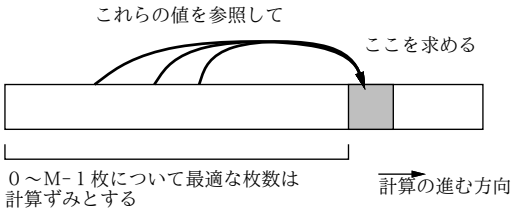

図 1: 動的計画法の考え方

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。

```
min = m+1   # M 円は 1 円玉 M 枚で支払えるので…
m.times do |c1|
  m.times do |c2|
    m.times do |c3|
      m.times do |c4|
        m.times do |c5|
          if c1*1+c2*10+c3*12+c4*25+c5*100 == m && c1+c2+c3+c4+c5 < min
            # c1～c5 を記録
          end
        end
      end
    end
  end
end
```

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 Ruby 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)。

```
$coins = [10, 12, 25, 100]    # 1 円玉は別途
$total = Array.new(1001, 0); $prev = $total.dup
```

```
def prepare
  1.step(1000) do |i|
    $total[i] = $total[i-1] + 1; $prev[i] = 1   # 1 円玉を増やす
    $coins.each do |c|
      if i >= c && $total[i-c] < $total[i]      # C 円玉が有利なら取り替え
        $total[i] = $total[i-c] + 1; $prev[i] = c
      end
    end
  end
end
def change(m)
  puts("no. of coins - #{$total[m]}\n")
  while m > 0 do print(" #{$prev[m]}"); m = m - $prev[m] end
  puts("")
end
```

prepare が動的計画法で表を作成し、change はその表を参照して具体的な金額に対するおつりの出し方を計算する。

@total[i] では枚数しか記録されていないので、その枚数を達成するために最後にいくらのコインを追加したかを @prev[i] に覚えておくようにした。全部のコインを表示するには、繰り返し@prev[i] を参照してコインを列挙していけばよい。この操作を**トレースバック** (逆追跡) という。動かした様子は次のとおり:

```
irb(main):002:0> prepare
=> 1
irb(main):003:0> change(32)
no. of coins - 3
 12 10 10
=> nil
irb(main):004:0> change(33)
no. of coins - 4
 12 10 10 1
=> nil
irb(main):005:0> change(35)
no. of coins - 2
 25 10
=> nil
irb(main):006:0> change(38)
no. of coins - 3
 25 12 1
=> nil
```

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。

2.3 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。」

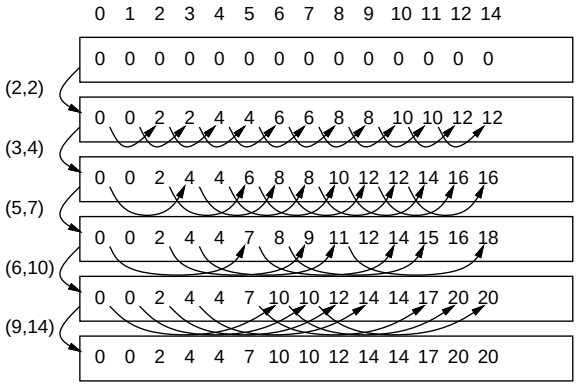

図 2: ナップサック問題

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:

```
irb(main):007:0> prepare
=> 5
irb(main):008:0> calc 8
size = 8, max value = 12
=> nil
irb(main):009:0> calc 9
size = 9, max value = 14
=> nil
irb(main):010:0> calc 10
size = 10, max value = 14
=> nil
irb(main):011:0> calc 20
size = 20, max value = 32
=> nil
```

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 Ruby のコードにしたものは次のようになる。

```
$sizes = [2, 3, 5, 6, 9]
$value = [2, 4, 7, 10, 14]
$total = Array.new(101, 0)
def prepare
  $sizes.length.times do |i|
    $sizes[i].step($total.length-1) do |j|
      v = $total[j-$sizes[i]] + $value[i]
      if v > $total[j] then $total[j] = v end
    end
  end
end
```

```
def calc(s)
  puts("size = #{s}, max value = #{$total[s]}")
end
```

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。

演習 3 合宿で、1 泊料金が「1 人部屋:5,000 円、3 人部屋:12,000 円、7 人部屋:20,000 円」というヘンなホテルに泊まることになった。どの部屋も数は十分あるものとする。もちろん、収容人数より少ない人数で泊まってもいい。敏腕マネージャのあなたは合計宿泊人数 $N(< 100)$ 人 (全員同性) に対して総額が最も小さい部屋の選択を行う必要がある。これを行うプログラムを作れ (とりあえず総額が分かればよいが、できれば何人部屋をどれだけ選ぶかの情報もある方がいい)。

2.4 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。

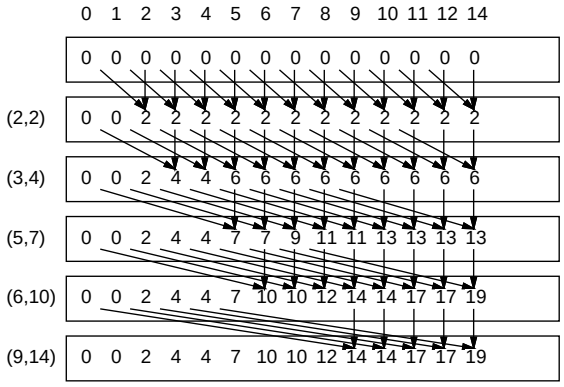

図 3: 資源が限られる場合

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 4 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。

2.5 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。

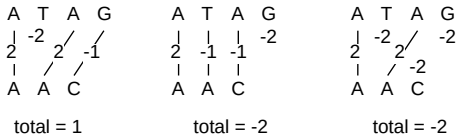

図 4: アラインメントの点数計算

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。

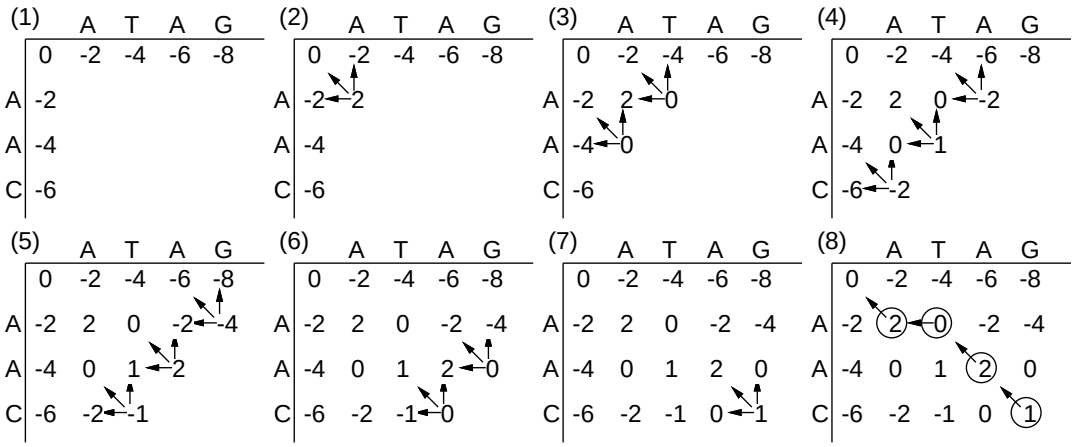

図 5: 動的計画法によるアラインメントの計算

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。

とりあえず、トレースバックは省略してアラインメント値だけ計算するプログラムを示そう。

```
def alignment(x, y)
  a = Array.new(y.length+1) do Array.new(x.length+1, 0) end
  1.step(a[0].length-1) do |i| a[0][i] = a[0][i-1] - 2 end
  1.step(a.length-1) do |j| a[j][0] = a[j-1][0] - 2 end
  1.step(a.length-1) do |j|
    1.step(a[0].length-1) do |i|
      if x[i-1..i-1] == y[j-1..j-1] then a[j][i] = a[j-1][i-1]+2
      else                               a[j][i] = a[j-1][i-1]-1
      end
      if a[j-1][i]-2 > a[j][i] then a[j][i] = a[j-1][i]-2 end
      if a[j][i-1]-2 > a[j][i] then a[j][i] = a[j][i-1]-2 end
    end
  end
  return a[y.length][x.length]
end
```

計算順序は簡単のため横一列ずつ全部計算してしまうようにしている。動かしているところを示す。

```
irb(main):002:0> alignment('ATAG', 'AAC')
=> 1
irb(main):003:0> alignment('ATAG', 'ATC')
=> 1
irb(main):004:0> alignment('ATAG', 'ATG')
=> 4
```

演習 5 上の例題をそのまま打ち込み動かせ。動いたら、トレースバックによりどこに挿入/削除/置換があったかを表示させよ (表示方法は各自工夫する)。

演習 6 ファイルを編集したとき、古いファイルと新しいファイルを見比べて「行単位で」どことどこを直したか (どこに行を挿入しどこの行を削除したか) 調べたいことはよくある。上の例題を手直しして (トレースバックも行い)、2 つのファイルの比較を行うプログラムを作れ。表示方法も工夫してほしい。

演習 7 上の例題の方法では 2 つの列の長さが M、N としたとき領域計算量が $O(M \times N)$ になってしまう。しかし、図 5 のように斜めに順序よく計算していけば、$O(M)$ 程度で済む ($M < N$ として)。演習 5 のプログラムをこのように直すことで、ファイルの大きさが数千行でも大丈夫なようにしてみよ (ただしトレースバックを 2 次元配列にしておいたら無意味なので、トレースバック情報の持ち方を工夫する必要がある — かなり高度)。

3　パターン認識

3.1　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。[1]

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。

3.2　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。

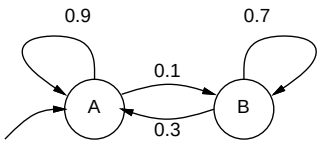

図 6: マルコフ過程の例

たとえば図 6 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
AAABBBBABBBBBAAAAAAAAAAABBBBBBBAAAAAAAAAAAABBAAAAAABBBBBBBA
AAAAABBBBBAAAAAAAAAABBBBAAAAAAAAAAABBBAAAAAAAAAABBAAABAAAAAA
```

[1]「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。

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)。

3.3 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]

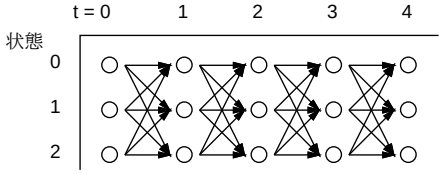

図 7: 前向きアルゴリズム

単にこの通り順番に計算するだけだが (図 7)、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。Ruby プログラムを示しておく。

```
$ns = 2
$init = [1.0, 0.0]
$trans = [[0.9, 0.1], [0.3, 0.7]]
$out = [[1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0],
        [0.1,0.1,0.1,0.1,0.1,0.5]]
def calc(str)
  input = Array.new(str.length)
  input.length.times do |t| input[t] = str[t] - '1'[0] end
  a = Array.new(input.length) do Array.new($ns) end
  $ns.times do |i| a[0][i] = $init[i]*$out[i][input[0]] end
  1.step(input.length-1) do |t|
    $ns.times do |i|
      s = 0.0
      $ns.times do |j| s = s + a[t-1][j]*$trans[j][i] end
      a[t][i] = s*$out[i][input[t]]
    end
  end
  r = 0.0
  $ns.times do |i| r = r + a[input.length-1][i] end
  return r
end
```

3.4 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 8)。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。

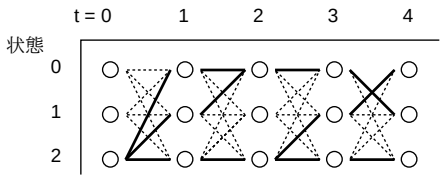

図 8: Viterbi のアルゴリズム

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。

```
def viterbi(str)
  input = Array.new(str.length)
  input.length.times do |t| input[t] = str[t] - '1'[0] end
  d = Array.new(input.length) do Array.new($ns) end
  prev = Array.new(input.length) do Array.new($ns) end
  $ns.times do |i| d[0][i] = $init[i]*$out[i][input[0]] end
  1.step(input.length-1) do |t|
    max = 0.0; k = 0
    $ns.times do |i|
      $ns.times do |j|
        if d[t-1][j]*$trans[j][i]>max then max=d[t-1][j]*$trans[j][i]; k=j end
      end
      d[t][i] = max*$out[i][input[t]]; prev[t][i] = k
    end
  end
  max = 0.0; k = 0
  $ns.times do |i|
    if d[input.length-1][i] > max then max=d[input.length-1][i]; k=i end
  end
  ptrace(prev, k, input.length-1); puts(""); return max
end
def ptrace(prev, k, i)
  if i > 0 then ptrace(prev, prev[i][k], i-1) end
  print("#{k} ")
end
```

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。

```
irb(main):013:0> viterbi '15234262664663624532'
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0
=> 2.59708964728546e-16
```

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 8 「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 9 図 9 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。

A　本日の課題 11A

演習問題から 1 つ選び、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。

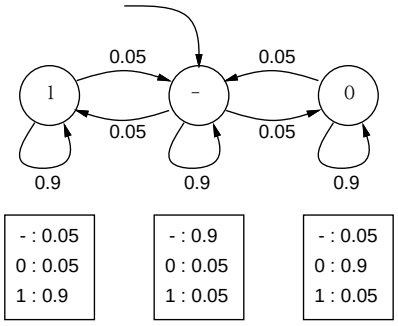

図 9: 練習問題のマルコフモデル

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 (Ruby 終了なので今回限定でない質問です)。

   Q1. 「動的計画法」の考え方は分かりましたか? 今後自力で使えそう?

   Q2. この講義は「情報科学」の学習を目的とし、その確認手段として「Ruby によるプログラミング」を演習して頂いていますが、本クラスにおけるその両者の位置づけやバランス、難易度についてどう思いますか。

   Q3. 講義開始時の自分を振り返り、今回までの講義で自分がどんなことを学んだか簡単にまとめて見てください (全体的な感想も含めてどうぞ)。

既に予告した通り、今回以降、次回までの課題はありません。

B　今回までの内容のあらまし

アンケート記載の参考に、今回までの各回の内容をリストアップしておきます。

1	プログラムとモデル化、手続き型計算モデル、プログラミング言語、Ruby によるプログラミング 数値の表現、整数の有限性、浮動少数点表現、誤差、情報落ち、桁落ち
2	制御構造、枝分かれ/if 文、繰り返し/while 文、計数ループ、数値積分、精度
3	数値積分 (再)、台形公式、シンプソンの公式、打ち切り誤差 方程式の求解、数え挙げ、区間 2 分法、ニュートン法、収束 データ型、基本データ型、複合データ型、論理値型、配列型
4	手続き/関数と抽象化/副作用、再帰関数/再帰的定義 レコード型、レコードの 2 次元配列、画像
5	整列、単純選択法、比較交換法、バブルソート、マージソート、クイックソート、ビンソート、基数ソート、時間計算量
6	連立方程式の数値解法、Gauss 消去法、Gauss-Jordan 法、ピボット選択、反復法、Jacob 法、Gauss-Seidel 法 乱数、擬似乱数、ランダムアルゴリズム、モンテカルロ法、乱数とゲーム
7	常微分方程式の数値解法、Euler 法、Runge-Kutta 法 (2 次、4 次) オブジェクト指向、クラス、インスタンス、有理数クラスの例
8	動的/再帰的データ構造、単連結リスト、エディタバッファ 表と探索、線形探索、2 分探索木、連想配列とハッシュ法
9	式木、抽象構文木、動的分配、継承、再帰による木の評価 字句解析器、BNF による構文定義、構文解析器、再帰下降解析
10	スタックとキュー、配列による表現、連結リストによる表現、構造のたどり 状態と状態遷移、有限オートマトン、ゲーム/パズルと状態空間の探索
11	動的計画法、コイン問題、ナップサック問題、文字列照合問題 パターン認識、隠れマルコフモデル、評価問題、復号問題と Viterbi アルゴリズム