情報科学 2011　久野クラス　＃ 13

久野　靖*

2011.1.27

今回は予告通り、昨年度の試験問題の解説をしますが、解説だけ聞いても面白くないので、バリエーションの演習問題を作りました。時間があれば少し演習してみていただきます。(注意: 以下の解説はあくまでも久野個人によるものであり、公式のものではありません。)

1　問題 1: コンビネーションの配列による計算

この問題は、次の再帰定義によるコンビネーションの計算を動的計画法に置き換えたものと考えることができます。

$$comb(n,r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (otherwise) \end{cases}$$

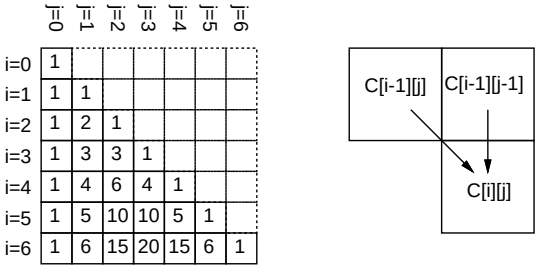

図 1: 配列を用いたコンビネーションの計算

すなわち、2 次元配列 c を用意し、c[i][j] が ${}_iC_j$ になるように計算していきます (図 1 左)。それには、初期値として i == 0 や i == j となる c[i][j] には 1 を入れ、あとは順番に c[i][j] = c[i-1][j-1] + c[i-1][j] により計算を進めます (図 1 右)。

設問 (a)

というわけで、[イ]に何を入れればよいかは明らかだと思います。穴を埋めて、あとコードの書き方をこのクラス風に手直しした版を次に示します。

```
def combination_loop(n, k)
  c = Array.new(n+1) do Array.new(n+1) do 0 end end
  0.step(n) do |i|
    c[i][0] = 1
    1.step(i-1) do |j|
      c[i][j] = c[i-1][j-1] + c[i-1][j]
#     puts "c[#{i}][#{j}] = #{c[i][j]}"
    end
    c[i][i] = 1
  end
  return c[n][k]
end
```

*筑波大学大学院経営システム科学専攻

違う点は次のりです。

- 2 次元配列の生成はよくご存じの通りなので直接書いた。
- for は説明したけれど普段使っていないので step にした。
- 値を返すところは `return` を明示した (式だけを書く流儀もあって、問題文ではそのスタイルで書かれている)。

設問 (b)

次に、(ア) の文を実行したときの添字と代入される値ですが、これはコメントになっている `puts` を実行してみれば分かります (まあ実行してみなくても分かりますが)。

```
irb> combination_loop 3, 2
c[2][1] = 2
c[3][1] = 3
c[3][2] = 3
=> 3
```

設問 (c)

最後に計算量はどうでしょうか。結局、このプログラムの作業というのは図 1 左にあるように三角形の領域の中を順に埋めて行くことに尽きます。それぞれの箇所を埋めるのは、1 を代入するか、2 つのセルの値を足し算して入れるかで、どちらにせよ単純 (定数時間) です。セルの数は n^2 に比例するので、計算量も $O(n^2)$ ということになりますね。

設問 (d)

次に再帰計算を行う版のコードが示されています。これも我々のクラスの流儀に書き直したものを示します。

```
def combination(n, k)
# puts "combination(#{n}, #{k})"
  if k > n
    return 0
  elsif k == 0
    return 1
  else
    retrun combination(n-1, k-1) + combination(n-1, k)
  end
end
```

この再帰計算は冒頭に示した再帰定義と少し違っています。つまり、「`n == k` のとき 1」ではなく「`n < k` のとき 0」としています。これでもよいのは、図 1 で点線の (使っていない) ところに全部 0 が入っているものとみなせば、`c[i][j] = c[i-1][j-1] + c[i-1][j]` を `i==j` に対しても適用することで (`c[i-1][i-1]` には既に 1 が入っているから) ちゃんと 1 が入るわけです。

では、これもコメントを外して出力しながら実行してみます。

```
irb> combination 3, 2
combination(3, 2)
combination(2, 1)
combination(1, 0)
combination(1, 1)
combination(0, 0)
combination(0, 1)
combination(2, 2)
```

```
combination(1, 1)
combination(0, 0)
combination(0, 1)
combination(1, 2)
=> 3
```

設問 (e)

これは単に説明すればいいだけですが、`combination_loop(n, n)` では イ の箇所で `c[n][n]` に直接 1 を入れることで計算しているのに対し、`combination(n, n)` では次のように `(n-1,n-1)`、`(n-2,n-2)`, …と `(0, 0)` まで再帰的にたどって 1 が返され、それ `t(n-1,n)` の 0、`(n-2,n-1)` の 0、…を足すことで最終的に 1 が返されます (図 2 に呼び出し関係の図を示しました)。

```
irb> combination 4, 4
combination(4, 4)
combination(3, 3)
combination(2, 2)
combination(1, 1)
combination(0, 0)
combination(0, 1)
combination(1, 2)
combination(2, 3)
combination(3, 4)
=> 1
```

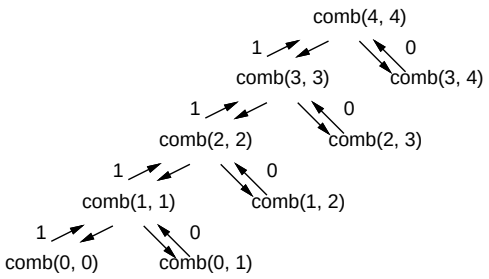

図 2: 再帰による combination(n,n) の呼び出し関係

演習 1 `combination_loop` のプログラムを打ち込んで動かせ。動いたら、パラメタとして次のものを与えたときの puts からの出力がどうなるかまず予想し、次に実行して確かめよ。

a. `combination_loop(4, 2)`

b. `combination_loop(8, 3)`

c. `combination_loop(8, -2)` [1]

演習 2 `combination` のプログラムを打ち込んで動かせ。動いたら、パラメタとして次のものを与えたときの puts からの出力がどうなるかまず予想し、次に実行して確かめよ。

a. `combination(6, 6)`

b. `combination(4, 3)`

c. `combination(5, 3)`

[1] ちなみに、Ruby では配列の添字にマイナスの整数を渡すと「配列の末尾から何番目」という意味になりますので。

2　問題 2: 個数を数える

この問題は前半が線形探索の問題で、後半は値が順に並んでいると線形探索しなくても済むという話になっています。まずは掲載プログラムが読めないと話にならないですが、そこは大丈夫でしょうか。プログラムをこのクラスふうに書き換えたものを再掲します。

```
def intcount(a, b, c)
  a.each do |x|
    j = 0
    while b[j] != 0 && b[j] != x
      j = j + 1
    end
    if b[j] == 0
      b[j] = x
      c[j] = 1
    else
      c[j] = c[j] + 1
    end
  end
end
```

配列 a から要素を 1 個ずつ変数 x に取り出し、その値を配列 b，c の組で表現されている表に記録していきます (図 3)。その際、表の何番目の位置に記録するかを判断するために、変数 j を使って表の先頭から探して行き、x と同じ値が登録されている箇所があれば、そこをカウントアップします。探して行って b[j] が 0 になったら、まだ登録されていないわけですから、新たにそこに登録します。この 2 つを 1 つのループで兼ねるため、while ループで「0 でもなく、x でもない間」j を増やしていき、ループから出てから「0 なら新規登録、そうでなければカウントアップ」するわけです。

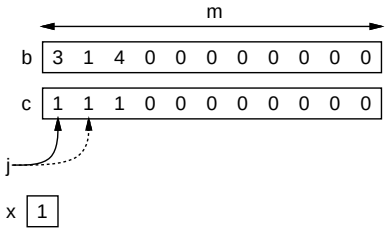

図 3: 線形探索による表への登録

設問 (a)

実際に測定してみてから考えましょう。測定のためのコードを示します。bench はおなじみのものです。test1 は 1〜m までの乱数 n 個から成る配列 a と、大きさ m で初期値 0 の配列 b，c を用意して intcount を呼び、所要時間を計測しています。

```
def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  return t2-t1
end
def test1(n, m)
  a = Array.new(n) do rand(m)+1 end
  b = Array.new(m, 0)
```

```
  c = Array.new(m, 0)
  return bench(1) do intcount(a, b, c) end
end
```

実際に計測してみましょう。

```
irb> test1 1000,1000
=> 0.234375
irb> test1 2000,1000
=> 0.625
irb> test1 3000,1000
=> 0.9921875
irb> test1 4000,1000
=> 1.3828125
irb> test1 4000,2000
=> 2.4140625
irb> test1 4000,3000
=> 3.1484375
```

つまり、n が 2 倍、3 倍になると時間が 2 倍、3 倍になり、さらに m が 2 倍、3 倍になった場合も時間が 2 倍、3 倍になるので、$O(mn)$ の時間計算量であるように見えます。

ここで例題コードを見ると、外側のループは n 回実行されますね。そして内側の if 文は定数時間ですが、内側ループは平均して m/2 回実行されると予測できるので、[2]この部分の実行時間が主となり、$O(mn)$ となるわけです。

設問 (b)

では次に、入力の配列 a が整列されているとしたらどうでしょう。この場合は、「次の値が今まで数えていた数と違う」なら、これまでに出て来たことのない値に決まっているので、表の「次の場所」に新しい数の項目を作ればよく、表の中を探す必要がありません (図 4)。

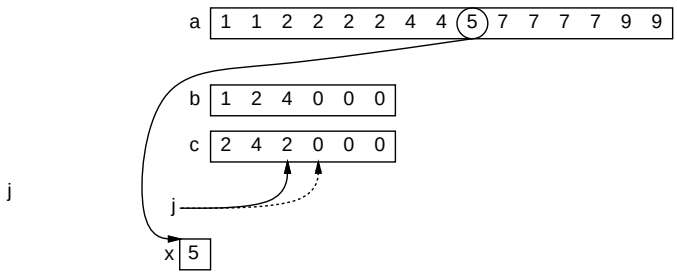

図 4: 入力が整列されている場合

これに対応する改良版は、たとえば次のように書くことができます。このコードでは、表の「現在の数の場所」を変数 j に保持し、その初期値は -1 とします。そして、入力の数を順次 x に取り出しながら、j が負である (最初の数である) か、現在の数と違う場合に、j を進めて新しい項目を作り、b[j] にその数を入れます。あとは、処理が合流したところで c[j] を 1 増やすことで、1 つずつ項目をカウントしていけます。

```
def intcount1(a, b, c)
  j = -1
  a.each do |x|
    if j < 0 || b[j] != x
      j = j + 1
```

[2]ただし n が十分大きい場合。m に比べて n が小さいか同程度だと、表にすべての数が登録されていない状態なので、内側ループの回数はもっと少なくなります。

```
      b[j] = x
    end
    c[j] = c[j] + 1
  end
end
```

では、時間計測してみましょう。整列ずみにするためには binsort を使用します。

```
def binsort(a)
  bin = Array.new(10000, 0)
  a.each do |i| bin[i] = bin[i] + 1 end
  k = 0
  bin.length.times do |i|
    bin[i].times do a[k] = i; k = k + 1 end
  end
end
def test2(n, m)
  a = Array.new(n) do rand(m)+1 end; binsort(a)
  b = Array.new(m, 0)
  c = Array.new(m, 0)
  return bench(1) do intcount1(a, b, c) end
end
```

これで時間計測してみましょう (binsort では値の最大値が 9999 になっていることに注意)。

```
irb> test2 10000, 1000
=> 0.0078125
irb> test2 100000, 1000
=> 0.109375
irb> test2 200000, 1000
=> 0.2265625
irb> test2 200000, 2000
=> 0.234375
irb> test2 200000, 3000
=> 0.234375
```

確かに、今度は実行時間が m の値に影響されなくなり、$O(n)$ になったようです。

演習 3 intcount、intcount1 を時間計測用のメソッドとともに入力し、時間計測を行って計算量について確認してみなさい。

演習 4 別の考え方として、a は整列されていない替わりに、m が「データとして現れる整数の最大値」であるような場合について考えてみましょう。このときは、m があまり大きくなければ図 5 のように配列 c だけを用い、「c[x] に数値 x が出て来た回数を入れる」ようにできます。この方針による intcount2 を作成し、計算量を評価の上、実測して確認してみなさい (この方法は要するに binsort と同じ考え方ですね)。

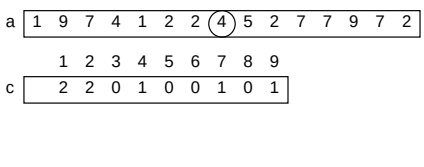

図 5: binsort と同様のアルゴリズム

3　問題 3: 数値計算と誤差

これは (a)〜(c) については知識問題ですので、説明できればよいということですね。

設問 (a)

計算機による実数計算では、扱える有効桁数が限られているため、演算においてその桁数を超えた部分の値は「丸め」られます。[3] これによる誤差を「丸め誤差」といいます。たとえば、8 桁ぶんの有効数字しかない場合、「1.0/3.0 → 0.33333333」となり、およそ 0.000000003333 · · · の丸め誤差が生じます。

設問 (b)

「`0.3 == 0.1 * 3`」の評価結果は「`false` になります。

```
irb> 0.3 == 0.1 * 3
=> false
```

この理由は、0.1 や 0.3 が二進表現では無限小数となり丸めて表されるため、そこにそれぞれ丸め誤差があり、互いに等しくならないからです。

```
irb> printf "%.20g\n", 0.3
0.2999999999999999889
=> nil
irb> printf "%.20g\n", 0.1 * 3
0.30000000000000004441
=> nil
```

設問 (c)

桁落ち誤差とは、非常に近い 2 つの数値を引き算すると、結果の有効数字が基の数値の持っていた有効数字よりずっと減ってしまうことによる誤差を言います。たとえば、1.23451234512345123 45 という値と 1.2345 という値の差は 0.00001234512345123 45 という値になるはずです。しかしこれを 10 桁の精度で演算すると、前者は 1.234512345 となり後者は 1.234500000 となるので、引き算すると 0.000012345 とたった 5 桁しか精度が無くなります。この場合、桁落ち誤差はおよそ 0.00001 ということになり、丸め誤差よりもずっと大きいことが分かります。

設問 (c)

この問題は根の公式による 2 次方程式の求根が題材です。

$$D = b^2 - 4ac, \qquad x = \frac{-b \pm \sqrt{D}}{2a}$$

メソッド `fa` はこれを忠実に計算しています。

```
def fa(a, b, c)
  d = b**2 - 4.0*a*c
  x1 = (-b + Math.sqrt(d)) / (2.0*a)
  x2 = (-b - Math.sqrt(d)) / (2.0*a)
  return [x1, x2]
end
```

[3] よく「四捨五入」と言いますが、これは十進表現だから「4 は切捨て 5 は繰り込む」のでして、二進表現だと「0 は切捨て 1 は繰り込む」「零捨一入」を行います。いちいちそういう言い方をするのも面倒なので一般化して「丸める」というわけです。

これは $-b$ と $\sqrt{D}$ の足し算/引き算を含みます。しかし、b と D の絶対値が非常に近いとき、まさに設問 (c) でやった「桁落ち」の置きやすい状況となり、有効桁数が少なくなります。これを避けるには、「足し算の側」だけを使うことです。それだと根の片方しか求まらないのではと思えますが、2 根を α、β とすると、根と係数の関係により、$\alpha\beta = \frac{c}{a}$ ですから、一方の根から他方の根を除算により求めることができます。この、「どちらが引き算になるか判断し、引き算になる側の根を根と係数の関係を使って計算し直す」のが `fb` で追加されたコードなわけです。

```
def fb(a, b, c)
  d = b**2 - 4.0*a*c
  x1 = (-b + Math.sqrt(d)) / (2.0*a)
  x2 = (-b - Math.sqrt(d)) / (2.0*a)
  if b > 0 then x1 = c.to_f / (a*x2)
           else x2 = c.to_f / (a*x1)
  end
  return [x1, x2]
end
def check(a, b, c, x)
  return [a*x[0]**2 + b*x[0] + c, a*x[1]**2 + b*x[1] + c]
end
```

そして、$x^2 - 100x + 1 = 0$ だと $\sqrt{D}$ と b が非常に近いので、`fa` だと引き算になる側に桁落ち誤差がはっきり現れるわけです。一方、`fb` では桁落ち誤差は発生しないため十分な精度があり、計算した根の値を元の式に入れて丸めるときっちり 0 になります。

```
irb> check(1, -100, 1, fa(1, -100, 1))
=> [0.0, 1.22124532708767e-13]
irb> check(1, -100, 1, fb(1, -100, 1))
=> [0.0, 0.0]
```

演習 5 この 2 次方程式の求根プログラム 2 種と `check` を動かし、どのような 2 次方程式だとナイーブ版 (`fa`) に誤差が現れるのかを検討しなさい。

演習 6 次の 2 つの数式を考えてみる。

$$\frac{\sqrt{x^2+1}-1}{x^2} \qquad \frac{1}{\sqrt{x^2+1}+1}$$

これらは数学的には同じものだが (前者の分子と分母に $\sqrt{x^2+1}+1$ を掛けてみるとわかる)、x の絶対値が 0 に近くなるとき、前者は桁落ちがおきるが後者はおきない。この数式で x を 0 に近づけると値は 0.5 に近づくはずだが、両方の数式で実際に計算してみて比較してみよ。

4 問題 4: 経路の数の動的計画法

この問題は典型的な (易しい) 動的計画法の問題だと言えます。動的計画法では、あるパラメタについての値を、パラメタがそれより小さい値 (複数) に基づいて計算するのでしたね。

設問 (a)

問題設定の様子を図 6 左に示します。ここで、中間位置の座標について見ると、そこに駒が到達する方法は

A. 左のマスからくる

B. 下のマスからくる

C. 斜め左下のマスからくる

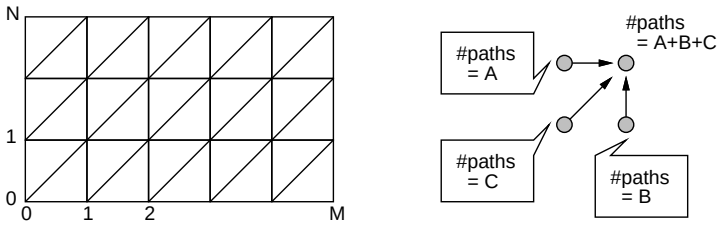

図 6: 経路の数

の 3 通りです。ですから、もしこのそれぞれのマスに到達する経路の数が分かっていれば、それらを合計したものが現在位置に到達する経路の数ということになります (図 6)。

では、いちばん左下のマス、および下/左端のマスはどうでしょうか。これらに到達する方法はすべて 1 通りですね (開始のマスからずっと横ないし縦に動いて行くしかないから)。

設問 (b)

では、これを動的計画法のアルゴリズムにするとどうでしょうか。図 7 のように 2 次元配列を用意し (ここでは先の図に合わせて Y 方向は上にいくほど大きくなるように描きました)、端の部分 (網かけの要素) は 1 に初期化し、あとは `t[m][n] = t[m][n-1] + t[m-1][n] + t[m-1][n-1]` を順番に適用して埋めて行けばいいわけです。そして最後にゴールのマスに相当する要素に記入された値が答えとなります。

3	1	7	25	63	129	231
2	1	5	13	25	41	61
1	1	3	5	7	9	11
0	1	1	1	1	1	1
	0	1	2	3	4	5

図 7: 配列への値の記入

演習 7 設問 (b) のアルゴリズムに対応する Ruby プログラムを作成して動かせ。

A　質問とアンケート 13A

出席は取らないのですが、アンケートはできれば送付頂けると幸いです。

1. Subject: は「Report 13A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 説明内容について質問があれば書いてください (無くてもよいです)
4. 以下のアンケートの回答。

Q1. 今回の説明内容はおおむね理解できましたか

Q2. 今回出て来た Ruby プログラムはすべて読めましたか。読めないとすればどのあたりが難しいですか。

Q3. 最後ですが全体的な感想をどうぞ。