テキスト処理'15 ＃3 – 計算量とアルゴリズムの工夫

久野　靖 *

2015.6.9

1　本日の内容

今回は前回の積み残しですが、少しスキップしてパネルデータのところからやり、集計上必要なところでハッシュの話をします。

データの分析作業が繰り返し実行されるのと比べ、データの抽出処理は 1 回やって必要なデータが取り出せればいいので、性能に関する要求は分析ほどシビアではありません。しかしそれでも、ナイーブな方法を使っていると時間が掛かりすぎて実用的でないことはよくあります。また、処理時間の問題のほか、これまでのように全データをメモリに読むのが困難な量であることもあり得ます。今回は時間計算量についてひととおり学びます。これはお話だけで簡単にします。

その後メモリ負荷の問題とメモリに全部読まずに順番に処理していく方法を扱います。最後に「名寄せ」問題について取り組んでみます。

今回も、例題に使用しているデータ等は以下から取れます。

```
http://w3in/~kuno/tproc15/                                  (学内)
http://www2.gssm.otsuka.tsukuba.ac.jp/staff/kuno/tproc15/   (学外)
```

2　パネルデータの処理

今度はもう少し実務に近そうなデータとして、パネルデータの抜粋を処理してみます (元が非常に大きなものなので、演習用に特定の 1 日ぶんだけ取り出して小さくしてあります)。形は次のようになっています。パネル番号とはポイントカードの ID のような、個人を識別する番号です。

パネル番号, 店番号, レシート番号, 発行時, 品数, 金額, 商品名

各欄の区切りは「,」で、商品名は文字列ですが、それ以外の欄はすべて整数になっています。1 文字目が「#」の行はコメントです。冒頭の数行を見てみましょう。

```
#panel,shop,receipt,time,count,amount,itemname
15360,20,643,20,1,138,ほうれん草
22568,22,7093,11,1,98,長ねぎ
28192,22,5043,16,1,98,長ねぎ
```

商品名の中に「,」は含まれないので、単純に「,」の箇所で区切ることでデータを抽出できます。では、このデータの中から各購入について品数、金額、商品名を取り出し、そのまま打ち出すという例をやってみましょう。

*経営システム科学専攻

```
def readdata
  r = []
  open('ITEM1.csv') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  return r
end
def pickdata(x)
  r = []
  x.each do |s|
    if s =~ /^#/ then
      # do nothing
    elsif s =~/^(\d+),(\d+),(\d+),(\d+),(\d+),(\d+),(.+)$/ then
      a,b,c,d,e,f,g = $1,$2,$3,$4,$5,$6,$7
      r.push([a.to_i, b.to_i, c.to_i, d.to_i, e.to_i, f.to_i, g])
    else
      $stderr.puts("wrong format: #{s}");
    end
  end
  return r
end
def show1(d)
  d.each do |a|
    printf("%3d %8d %s\n", a[4], a[5], a[6]);
  end
end

$data1 = readdata
$data2 = pickdata($data1)
show1($data2)
```

上のコードを動かしたところの冒頭部分を示します。

```
1      138 ほうれん草
1       98 長ねぎ
1       98 長ねぎ
1       98 長ねぎ
1       78 長ねぎ
1       68 長ねぎ
2       52 たまねぎ（バラ用）
3       96 たまねぎ（バラ用）
...
```

演習 13 このデータに対して次の処理をするプログラムを紙に書きなさい。readdata、pickdata は同じままでよいはず。

a. この日の売り上げ合計を表示。★
b. この日に売れた全商品の平均販売価格を表示。★
c. この日に売れた最も高額な商品の販売価格価格と商品名を表示。
d. この日に売れた最も安い商品の販売価格価格を商品名を表示。
e. この日の販売で同じ商品のまとめ買い数の最大と商品名を表示。

演習 14 紙に書いたプログラムを実際に動かして動作を確認しなさい。

3 ハッシュを使った集計

次は「一番多く買った人は誰か」を調べてみます。パネルデータは品目ごとにバラバラになっていますから、それぞれの人 (パネル番号) ごとに累計していき、累計が一番高い人を見つける必要があ

ります。この「○○ごとに累計」のような処理をおこなうのに適した機能が「ハッシュ(hash)」と呼ばれるデータ構造です。ハッシュは配列によく似ていますが、配列が添字が順に並んだ整数に限られるのに対し、バラバラの値や文字列など任意の値が添字にできます。概念としては、図 1 のような鍵 (key) と値 (value) の対になった「表 (テーブル)」を想像すればいいでしょう。

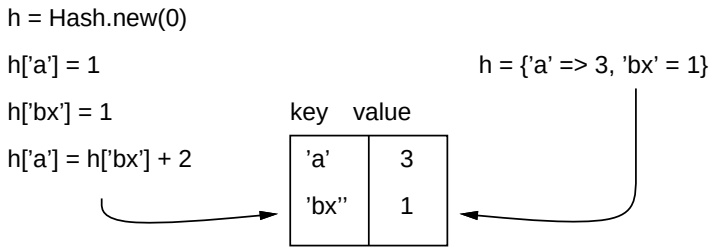

図 1: ハッシュの概念

ハッシュを作るのには `Hash.new(0)` のようにして空っぽのハッシュを作ることができます。その後、図 1 左では文字列を添字として値を代入したり、現在の値を取り出したりしています。「どこの値を」取り出すかが添字 (鍵) によって指定されるわけです)。ところで、`Hash.new(0)` の 0 は何かというと、まだ値を入れていない添字を指定して取り出した場合に出て来る値を 0 にする、という指定です (今回の例では金額などを扱うので 0 が出て来ると便利)。なお、ハッシュの別の作り方として図 1 右のように「{ 鍵 => 値, 鍵 => 値, … } 」のように複数の鍵と値を指定して一度に作るという方法もあります。

では、ハッシュを使って先の「最も多く買った人とその金額」をやってみます。

```
def readdata
  r = []
  open('ITEM1.csv') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  return r
end
def pickdata(x)
  r = []
  x.each do |s|
    if s =~ /^#/ then
      # do nothing
    elsif s =~/^(\d+),(\d+),(\d+),(\d+),(\d+),(\d+),(.+)$/ then
      a,b,c,d,e,f,g = $1,$2,$3,$4,$5,$6,$7
      r.push([a.to_i, b.to_i, c.to_i, d.to_i, e.to_i, f.to_i, g])
    else
      $stderr.puts("wrong format: #{s}");
    end
  end
  return r
end
def show2(d)
  total = Hash.new(0); max = 0; maxid = 0
  d.each do |a|
    total[a[0]] += a[5]
    if total[a[0]] > max then max = total[a[0]]; maxid = a[0] end
  end
  printf("max = %d, panel = %d\n", total[maxid], maxid);
end

$data1 = readdata
$data2 = pickdata($data1)
show2($data2)
```

説明は次の通り。

- 最初にハッシュを作り total という変数に保持。また現在の最大値と最大値を達成したパネル id を保持する変数を用意 (最初は 0)。
- 各データについて「数量×金額」を計算し、現在のその人の total に加算する。
- 加算した結果をこれまでの最大値を比較し、より大きければ最大値を更新するとともにそのパネル id も記録
- 処理が終わったら最後に金額とパネル id を出力。

演習 15 同じデータに対してハッシュを使用して次の処理をするプログラムを紙に書きなさい。readdata、pickdata は同じままでよいはず。

a. 最も売れた件数が多い商品と売れた件数を求めなさい。★
b. 上記に加え、売れた合計金額も求めなさい。★
c. この日に買物した人数を求める (ヒント: ハッシュに対して「`.size`」を参照するとテーブルの項目数が分かる)。
d. 購入金額が最も多い時間帯を求める。
e. 一人でもっとも多くの品物を購入した人のパネル ID と何品購入したかを求める (ヒント: ハッシュの各値としてさらにハッシュを入れる)。

演習 16 紙に書いたプログラムを実際に動かして動作を確認しなさい。

4 時間計算量

そもそも、プログラムが動作するのに掛かる時間はどのようにして評価するのがよいのでしょうか。実際のところ、あるプログラムを書いたとして、それがあるマシンでどれくらい時間を要するのかは、「計って見るしかない」というのが本当のところです。たとえば、足し算を百万回実行するだけのプログラムを見てみます。

```
$n = ARGV[0].to_i
$x = 0
$n.times do |i| $x = $x + 1 end
```

百万回といいつつ、百万とはどこにも書いていないですね。実は ARGV という配列にはコマンド行のパラメタが渡るので、その先頭のものを整数にして `n` に入れることで実行時に回数を指定できるようにしています。これを用いて計ってみます。

```
% time ruby time1.rb 1000000
0.096u 0.000s 0:00.09 100.0%    5+176k 0+0io 0pf+0w
```

もうちょっと回数を増やさないと時間が短かすぎて正確なところが分からなさそうです。

```
% time ruby time1.rb 10000000
0.665u 0.000s 0:00.66 100.0%    5+168k 0+0io 0pf+0w
% time ruby time1.rb 100000000
6.339u 0.000s 0:06.33 100.0%    5+167k 0+0io 0pf+0w
```

これを見ると、回数を 10 倍にすると時間もおよそ 10 倍になっていますね。まあうなずける結果です。

もうすこし複雑なプログラムで見てみましょう。今度はサイズ n の配列 (中身は区間 $[0, 1)$ の一様乱数) を生成した後、その最大値を求めて打ち出すというものです。

```
$n = ARGV[0].to_i
$a = Array.new($n) do rand end
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
puts(arraymax($a))
```

```
% time ruby time2.rb 1000000
0.9999991959689972
0.213u 0.000s 0:00.21 100.0%    5+169k 0+0io 0pf+0w
% time ruby time2.rb 10000000
0.9999999950265851
1.817u 0.015s 0:01.83 99.4%     5+168k 0+0io 0pf+0w
% time ruby time2.rb 20000000
0.9999999273403023
3.547u 0.086s 0:03.63 99.7%     5+168k 0+0io 0pf+0w
```

これもやはり、サイズ n に比例していますね。このように、扱うデータの量 n に対して比例する時間の掛かるプログラムを「時間計算量が $O(n)$(オーダー n) のプログラム」と言います。

では、どんなプログラムでも $O(n)$ が普通なのでしょうか? そんなことはありません。次のプログラムを見てみましょう。

```
$n = ARGV[0].to_i
$a = Array.new($n) do rand end
def arrayminrange(a, i, j)
  min = a[i]; pos = i
  for k in i+1..j do
    if min > a[k] then min = a[k]; pos = k end
  end
  return pos
end
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
def selectionsort(a)
  for i in 0..a.size-2 do
    swap(a, i, arrayminrange(a, i, a.size-1))
  end
end
selectionsort($a)
```

このプログラムは、配列に生成したデータを単純選択法を呼ばれるアルゴリズムで「昇順に整列」します。そのやり方は次のようになります。たとえばデータ数が 1000 個 (配列の番号でいうと 0 番〜999 番) だとすると:

- まず、0 番〜999 番の中で一番小さい要素の番号を見つける。
- その番号の要素を 0 番の要素と入れ替える。
- 次に、1 番〜999 番の中で一番小さい要素の番号を見つける。
- その番号の要素を 1 番の要素と入れ替える。
- 次に、2 番〜999 番の中で一番小さい要素の番号を見つける。

- その番号の要素を 2 番の要素と入れ替える。
- ...
- 最後に、998 番〜999 番の中で一番小さい要素の番号を見つける。
- その番号の要素を 998 番の要素と入れ替える。

「配列の指定範囲で一番小さい要素の番号を見つける」のが `arrayminrange` で、「配列の i 番と j 番を入れ替える」のが `swap` です。整列全体は `selectionsort` が行いますが、これが上のような動作になっていることは見れば分かりますね。

さて、これの時間を計ってみましょう。

```
% time ruby time3.rb 10000
3.967u 0.000s 0:03.96 100.0%    5+168k 0+0io 0pf+0w   ← 1 万要素で 4 秒弱
% time ruby time3.rb 20000
15.734u 0.007s 0:15.74 99.9%    5+167k 0+0io 0pf+0w   ← 2 万要素で 16 秒
% time ruby time3.rb 30000
35.322u 0.023s 0:35.34 100.0%   5+167k 0+0io 0pf+0w   ← 3 万要素で 35 秒
```

つまり、データが 2 倍、3 倍になると時間は 4 倍、9 倍になります。つまりデータ量の自乗に比例するので、このようなプログラムを「時間計算量が $O(n^2)$ のプログラム」といいます。

なぜそうなるのでしょう? データを調べる回数の一番多いところは、どの値が一番小さいかを探す部分です。これは上の説明を見ると分かるように、データ量が N のとき、まず n 個調べ、次に $n-1$ 個調べ、次に $n-2$ 個調べ…として、最後に 2 個調べます。これらを合計すると、次のようになります。

$$n + (n-1) + (n-2) + \cdots + 3 + 2 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$$

ここで n が大きいときには n^2 の項がほとんどを占めるので、それで所要時間も n^2 に比例するようになるわけです。一般に、データ量 n に対してプログラム中の実行する回数の一番多い箇所を探し、そこを実行する回数の式のうちで最も次数の高いものを取り出して $O(x)$ の形で書くことで、時間計算量 (オーダー) が分かります。それは、次数の小さいところはデータが多くなったときにほとんど影響が無くなるからです。

しかし、オーダーは分かったとして、実際に掛かる時間を見積もるには? それは、処理の内容や動かしているハードなどにより千差万別となり、一概に言えません。だから見積もる一般的な方法は無いわけです。しかし、たとえば 1000 とか 10000 でやってみて秒数が分かれば、その先データが増えたらどうなるかはオーダーが分かればできますね? そこが有用なところです。

さらに、どのくらいのオーダーなら実用的なプログラムなのか、ということもおおよそ分かっています。たとえば、$O(n)$ とか $O(n \log n)$ のプログラムはある程度大きなデータでも扱えます。しかし $o(n^2)$ だと少しデータ量が増えると処理が難しくなります。$o(n^3)$ とか $o(2^n)$ とかもあって、これらはごく少ないデータでしかプログラムが役に立たないわけです。

5 領域計算量

上では処理時間のことを考えましたが、プログラムが使用するメモリの量も同様に問題になります。次のプログラムを見てみましょう。

```
$n = ARGV[0].to_i
$a = [1]
$n.times do |i| $a = $a + $a end
```

このプログラムは、まず長さ 1 の配列を用意し、それから長さを倍々にすることを n 回おこないます。倍にするときにデータを全部コピーしなければならないので、コピーの回数は次のようになります。

$$2 + 2^2 + 2^3 + 2^4 + \cdots + 2^{n-1} + 2^n = 2^{n+1} - 2$$

つまり、このプログラムの時間計算量は $O(2^n)$ となります。さらに、配列の長さも 2^n になるので、領域計算量 (メモリ使用量のオーダー) も $O(2^n)$ となります。つまり、n が 1 増えるごとに時間もメモリ使用量も倍になります。

実際に動かしてみましょう。

```
% time ruby memory1.rb 24
0.213u 0.142s 0:00.36 97.2%     5+165k 0+0io 0pf+0w
% time ruby memory1.rb 25
0.419u 0.264s 0:00.69 97.1%     5+167k 0+0io 0pf+0w
% time ruby memory1.rb 26
0.817u 0.523s 0:01.35 98.5%     4+163k 0+0io 0pf+0w
% time ruby memory1.rb 27
1.582u 1.071s 0:02.66 99.6%     4+162k 0+0io 0pf+0w
% time ruby memory1.rb 28
2.826u 2.314s 0:05.15 99.6%     4+163k 0+0io 0pf+0w
% time ruby memory1.rb 29
^C^Z
Suspended
% kill -9 %
[1]    Killed
4.994u 4.342s 0:47.61 19.5%     4+162k 0+0io 5746pf+0w
```

$n = 28$ までは確かに時間が倍々になっていますが、29 になるととたんにいくら待っても終わらなくなりました (この実験に使った PC では)。強制終了させてみると、これまで 99%くらいだった CPU 使用率が低くなっています。これは、PC に搭載されているメモリでは足りなくなり、一部を 2 次記憶に退避させたりし始めたことを意味します。現在のコンピュータではこのように、メモリ [1] に入り切らないデータを扱おうとすると極端に性能が下がる性質がありますので、注意が必要です。

6　パネルデータを扱う (2)

では次の練習ですが、先と同じ形のパネルデータを扱って頂きます。

```
100138,22,7165,14,1,58,レモン
103624,12,9826,17,1,398,オレンジ
100403,12,9384,15,2,276,ゴールドキウイ　３０・３３
100085,12,1223,14,1,49,キウイフル- ツ
100636,12,9795,17,4,196,キウイフル- ツ
...
```

データは先頭から「パネル番号」「店舗番号」「レシート番号」「レジ通過時刻 (時のみ)」「購買点数」「金額」「品名」で、最後の品名だけが文字列、あとは整数です。

[1] 仮想記憶機能によって提供される論理的なメモリではなく、プロセスに実際に割り当てられる物理的なメモリ (実メモリ) のことです。

次に、データを処理するのにこれまでは全部配列に入れていましたが、データ量が多いときはメモリに入りきらなくて遅くなることがあります。今回は入り切るのですが、今回は練習として、全部読まないで扱うやり方を使っていただきます。それには最初の readdata で読んでいたところとその先の pickdata 等でデータを分解処理していたところをくっつけて 1 つのメソッドで処理します。そして処理に必要なデータはファイルを読みながら取り出し、整数のものは整数に変換し、後で使うもののみ保管します。こうすることでメモリ使用量を押えることができます。

なお、「後から使うデータを保管」と書きましたが、そのデータの受け渡しは少量のもの (数値とか 1 つの配列 1 とか) なら return で返せばいいのですが、多数あるときはグローバル変数に記録でもよいと思います。

では例題として、「全売り上げの合計を求める」だけのプログラムを示します。

```
def readprocess
  total = 0
  open('ITEM1.csv') do |f|
    f.each_line do |s|
      if s =~ /^#/ then
        # do nothing
      elsif s =~ /^(\d+),(\d+),(\d+),(\d+),(\d+),(\d+),(.+)$/ then
        id = $1.to_i; shop = $2.to_i; rcpt = $3.to_i; hour = $4.to_i
        num = $5.to_i; amt = $6.to_i; name = $7
        total += amt
      else
        $stderr.puts("wrong format: #{s}")
      end
    end
  end
  puts(total)
end
 readprocess
```

動かしてみましょう。

```
% ruby panel1.rb
2542285
```

演習 17 例題を打ち込み、データファイルをコピーしてそのまま動かしてみなさい。

演習 18 次のような問いに答えるようにプログラムを改訂しなさい。紙に書くので、変えないところは「ここは同じ」みたいに書いておけばいいです。前回やったハッシュ表が必要になるので忘れていたら復習するか質問すること。

a. レシートあたり (=1 回あたり) 売り上げ価格の平均を求める。★
b. 最も販売総額の大きい店舗と小さい店舗の番号を表示する。★
c. 最も 1 点の金額が高い商品とその金額を求める。
d. 最も売れた金額が多い商品と売れた金額を求める。
e. 時間帯ごとの売り上げ金額の一覧表を表示する。

演習 19 紙に書いたプログラムを実際に動かして動作を確認しなさい。

7 ハッシュ表の各データを取り出す

ここまででだいぶ、ハッシュ表にデータを蓄積できるようになったと思います。ハッシュ表に蓄積したデータを取り出す方法について簡単に説明しておきましょう。基本的にはハッシュ表のメソッド each を使うことで値が取り出せます。

```
h.each do | key, val | ... キーと値が順番に受け取れる ... end
```

ただし、キーと値の出て来る順番は制御できないので、(普通やりたいように) 特定の順番で並べたい場合はいちど配列などに入れ直して整列する必要があります。たとえば、商品毎の売り上げ累計を計算した後、上位から 10 件を表示してみます。

```
$nameamt = Hash.new(0)
def readprocess
  open('ITEM1.csv') do |f|
    f.each_line do |s|
      if s =~ /^#/ then
        # do nothing
      elsif s =~ /^(\d+),(\d+),(\d+),(\d+),(\d+),(\d+),(.+)$/ then
        id = $1.to_i; shop = $2.to_i; rcpt = $3.to_i; hour = $4.to_i
        num = $5.to_i; amt = $6.to_i; name = $7
        $nameamt[name] += amt
      else
        $stderr.puts("wrong format: #{s}")
      end
    end
  end
end
def printnamebyamt(n)
  a = []
  $nameamt.each do | n, v | a.push([n, v])  end
  a.sort! do |x, y| y[1] <=> x[1] end
  n.times do |i| printf("%8d %s\n", a[i][1], a[i][0]) end
end
readprocess
printnamebyamt(10)
```

整列については、前回やった配列の sort!を使っています。実行例を見てみましょう。

```
% ruby panel2.rb
  54727 豚ヒレ
  39501 日本ハムシャウエッセン２コ巻
  26312 うなぎ本焼（大）
  22372 ★サトウ　切り餅パリッとスリット
  21780 パールライス　無洗米特別栽培米宮城県産ひ
  20210 バナナ
  19622 刺身用びんちょうまぐろ（解凍）
  18718 ぶなしめじ
  18055 ＰＢセレクション　北海道牛乳
  14900 味の素　ギョーザ
```

演習 20 次のような問いに答えるようにプログラムを改訂しなさい。要点の部分だけ紙に書けばよい。

a. (レシートあたり)1 回あたり売り上げ金額の中央値、第 1・第 3 四分位数を求める (ヒンジでよい)。★

b. 各店の売り上げ総額一覧を金額の大きい順に表示する。★

c. 1 点あたり金額の高いものから順に 10 件、商品名と金額を表示する。

d. 最も売れた金額が少ない (が 0 円でない) 商品を金額とともに少ない順に 10 点表示する。

e. 時間帯ごとの売り上げ金額の一覧を金額の多い順に表示する。

演習 21 紙に書いたプログラムを実際に動かして動作を確認しなさい。

8　名寄せの問題

ここから新しい内容ですが、次はとある会社の特許のデータを扱います。

```
#出願人, 番号, 発明の名称, 出願日, 発明者 1, 発明者 2, 発明者 3, 発明者 4, 発
明者 5, 発明者 6, 発明者 7, 発明者 8, 発明者 9, 発明者 10, 発明者 11, 発明者
12, 発明者 13, 発明者 14, 発明者 15, 発明者 16, 発明者 17, 発明者 18, 発明者
19, 発明者 20, 発明者 21, 発明者 22, 発明者 23, 発明者 24, 発明者 25, 発明者
26,UPC 1,UPC 2,UPC 3,UPC 4,UPC 5,UPC 6,UPC 7
Apple Inc.,USD714294,Housing plate,2012.08.17,Dinh; Richard Hung
Minh,Howarth; Richard P.,Myers; Scott A.,Pakula; David A.,Tan;
Tang Yew,,,,,,,,,,,,,,,,,,,,,,D14/439,,,,,,
Apple Inc.,USD713855,Display screen or portion thereof with
graphical user interface,2012.03.06,Roberts; Samuel
Morgan,Ubillos; Randall,,,,,,,,,,,,,,,,,,,,,,,,D14/487,,,,,,
...
```

ここで知りたいのは、この会社で誰が何件の特許の (共同) 出願者となっているか、ということだとします。それには、1 行ずつ CSV を読んで、特許番号と出願者の組を抽出していきます。まずは名前を整列して順に表示するプログラムを見てみましょう。

```
require "csv"
def readdata
  r = []
  num = 0
  CSV.foreach("PAT1.csv") do |a|
    num += 1; if num == 1 then next end
    for i in 4..29 do
      if a[i] !=~ /^$/ then r.push([a[1], a[i]]) end
    end
  end
  return r
end
def countbynames(a)
  h = Hash.new(0)
  a.each do |x| h[x[1]] += 1 end
  return h
end
def printnames(h)
  h.keys.sort.each do |name| puts(name) end
end
$id_name = readdata
$name_count = countbynames($id_name)
printnames($name_count)
```

ここまではカンマ区切りを自分でパターンマッチしてデータを取り出していましたが、今度は CSV データの中にも (とくに人名の一部で) カンマが現れるので、その単純な方法ではできません。そこでここでは、Ruby に備わっている CSV ライブラリを使ってデータを取り出すことにしました。

次に、各特許ごとに発明者 1〜26 は配列の 4〜29 番になるので、それを取り出し、空でないなら「特許番号、名前」の組を結果に追加していきます。ここまでが `readdata` の処理です。

次の `countbynames` はハッシュ表を使い、各人が何件の特許に加わっているかを累計していきます。`printnames` はハッシュ表からキーの並びの配列を取り出し、整列して表示しています。これをやった冒頭部分を見てみましょう。

```
Adam; Chris
```

```
Adams; Steven
Ai; Jiang
Akahane; Ryosuke
Akana; Jody
Akana; Jody Be
Alben; Lauralee A.
Amelse; Adrian J.
Andre ; Bartley K.
Andre; Bart K.
Andre; Bartley K
Andre; Bartley K.
Andre;Bartley K.
Andreini; David
```

これを見ると、同一人物が複数回現れていることが分かります。それはなぜかというと、同一人物でも名前の表記に微妙な揺れがあり、そのため文字列としては同一になっていないからです。

人間が眺めるぶんにはこれでも問題はないのですが、プログラムの処理により一人一人の性質を集約しようとすると (今回の場合は誰が何件の特許に関与したかですね)、このままではまずいわけです。そこで、表記の揺らぎを吸収して「一人のデータは一人として扱う」ようにしたいわけです。このような、さまざまな経緯で分かれてしまっている同一人 (や組織) のデータを統合することを「名寄せ (agglegation)」と言います。

名寄せの 1 つの方法は、人間が実際に目でみて違うものを統一した形に修正することです。たとえば、次のようなファイルを手で作るとします。

```
$normalize = {
'Akana; Jody' => 'Akana; Jody Be',
'Andre ; Bartley K.' => 'Andre; Bartley K.',
'Andre; Bart K.' => 'Andre; Bartley K.',
'Andre; Bartley K' => 'Andre; Bartley K.',
'Andre;Bartley K.' => 'Andre; Bartley K.',
...
}
```

このようなファイルがあったとすれば、それをプログラムの一部に組み込んでおき、名前を読み込んだ直後に次のようにして表記のゆれを統一できます。

```
if $normalize[name] != nil then name = $normalize[name] end
```

ここで問題は、目で見て同じ名前の書き換え表を作るみたいな単純労働は人間がやるべきことではない、ということです。そこで今回は、ある程度までコンピュータを使って名寄せをサポートすることを考えてみます。

9 類似度の判別

要するに今回必要なのは、「全く同一ではないが、類似している文字列について、類似していると判断する」ことです。そのような 1 つの方法として、アラインメント (alignment、対応づけの計算) があります。たとえば、「akasaka」「sakasama」「isasaka」という 3 つの語で、互いにどれとどれが「一番似て」いるでしょうか?

図 2: 対応つけの例

1 つの考え方として、図 2 のように「対応のつくもの」どうしを線で結ぶとして、「交差しないように引ける線の最大本数」で類似度を計ることができます。問題は、さまざまな線の引き方のうち最大をどうやって求めるかですが、これは「動的計画法」というやり方を使えばできます。以下に説明しましょう。

		s	a	k	a	s	a	m	a
	0	1	2	3	4	5	6	7	8
a	1								
k	2								
a	3								
s	4								
a	5								
k	6								
a	7								

		s	a	k	a	s	a	m	a
	0	1	2	3	4	5	6	7	8
a	1	2	1	2	3	4	5	6	7
k	2	3	2	1	2	3	4	5	6
a	3	4	3	2	1	2	3	4	5
s	4	3	4	3	2	1	2	3	4
a	5	4	3	4	3	2	1	2	3
k	6	5	4	3	4	3	2	2	3
a	7	6	5	4	3	4	3	3	2

図 3: アラインメントの動的計画法

2 次元配列を用い、縦と横にそれぞれ文字列の文字を 1 文字ずつ配置します (一番先頭はあけてあります)。縦と横の位置はそれぞれ、その文字列をどこまで進んだかを意味します。両方の文字列全体を対応させるのですから、最後は右下隅のところまで来ます。

これから、この配列の中に「2 つの (部分) 文字列を対応させるのに、文字列の挿入・削除を何回行ったか」を数えて行きます。まず、左上隅はどちらの文字列も 1 文字も進んでないので、挿入・削除は 0 回なので 0 と記入します。そして上端の列、左端の列は 1 文字進むごとに 1 文字ずつ挿入・削除するので 1 ずつ増える値を記入します (図 2 左)。一般に、右に 1 マス、または下に 1 マス進むということは、その文字を飛ばすことですから 1 個の挿入・削除になります。

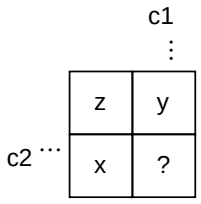

図 4: アラインメントの値の計算方法

さて、ここからが大切なところです。図 4 のような状況で、「?」のますの値を記入したいとします。ここで x、y、z はもう値が求まっているものとします。そうすると、次の 3 つ場合があり得ます。

- x のマスから右に 1 つ動く。値は x+1。
- y のマスから下に 1 つ動く。値は y+1。
- z のマスから右下に 1 つ動く。値は、「そのマスに対応する縦横 2 つの文字 c1 と c2 が同じ文字」ならそのまま対応するので z でよい。「同じでない」なら、c1 を削除して c2 を挿入するの

で z+2。

ここで目的は「できるだけ対応する文字を多くしたい (つまり挿入削除を少なくしたい)」わけですから、上の 3 つの場合のうち「最も小さい値」を「?」のところに記入すればよいわけです。これを左上から右下までずらっと順番に行うと、右下隅のマスに値が求まり、これが 2 つの文字列全体を対応させるときの「できるだけ少ない挿入・削除の回数」になります (なお、この数値のことを「編集距離」と呼びます)。

では、2 つの文字列を渡されて、編集距離を返すメソッドを示します (badness については後で)。

```
def editdist(s, t)
  a = Array.new(s.size+1) do |i| Array.new(t.size+1) do |j| i+j end end
  for i in 1..s.size do
    for j in 1..t.size do
      x = a[i-1][j-1]
      if s[i-1] != t[j-1] then x += 2 end
      a[i][j] = [a[i-1][j]+1, a[i][j-1]+1, x].min
    end
  end
  return a[s.size][t.size]
end
def badness(s, t)
  return editdist(s, t) - (s.size - t.size).abs
end
```

これを動かしてみましょう。

```
irb> load 'editdist.rb'
  => true
irb> editdist 'akasaka', 'sakasama'
  => 3
irb> editdist 'sakasama', 'isasaka'
  => 5
irb> editdist 'isasaka', 'akasaka'
  => 4
```

挿入・削除は 1 点、違う文字どうしの対応は挿入+削除なので 2 点であることに注意。

ところで、後ろにくっついている badness は何でしょう? 今やりたいのは、英語の人名の「近さ」を調べたいのだということに注意。そうすると、人名の表記の揺れは「Steven」と「Steve」や「S」のように一部を省略する形が多いことが分かります。省略をすると文字列は短くなりますから、このような関係にある 2 つの文字列では「編集距離 (挿入・削除の数) から文字列の長さの差を引く」と 0 になるはずです。これを計算するのが badness なわけです。

10 名寄せのためのハッシュ表生成

では、実際に前述のようなハッシュのリテラルを書き出すプログラムを見てみましょう。先に生成した名前の並びが `names.list` というファイルに入っているものとして、それを読んで処理します。このとき、直接名前の文字列を badness で判定するのではなく、その文字列をコピーしてから「大文字を小文字に直す」「ピリオドを削除」などいくつかの作業をおこなって「正規化」してから処理に入るようにしています。その方が何かと柔軟性が高いので。

```
def readnames
  r = []
  open('names.list') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  $names = r
end
def makecanon
  b = Array.new($names)
  b.each_index do |i|
    b[i] = $names[i].dup
    b[i].tr!('A-Z', 'a-z')
    b[i].sub!(/ +/, ' ')
    b[i].sub!('.', '')
    b[i].sub!(/^ +/, '')
    b[i].sub!(/ +$/, '')
  end
  $canon = b
end
 //editdist, badness here...
def checkcomb
  a = $canon
  $map = Array.new(a.size, -1)
  b = Array.new(a.size, 99)
  for i in 0..a.size-2 do
    for j in i+1..a.size-1 do
      x = badness(a[i], a[j])
      if x < 4 && b[j] > x then $map[j] = i; b[j] = x end
    end
  end
end
def printmap
  puts "$normalize = {"
  $names.each_index do |i|
    k = $map[i]; if k < 0 then next end
    while $map[k] >= 0 do k = $map[k] end
    puts("  '#{$names[i]}' => '#{$names[k]}', // #{k}")
  end
  puts("}")
end
 readnames
 makecanon
 checkcomb
 printmap
```

readnames はファイルから名前一覧を配列$names に読み込みます。makecanon は名前一覧のコピーを作り、大文字を小文字に変換、複数の空白を 1 つに、ピリオド削除、行頭行末の空白削除などを行い、配列$canon に入れます。checkcomb はすべての名前の組について badness を呼んで値が 4 未満なら$map に書き換えるべき相手の番号を入れます (初期値は -1)。最後に printmap は各名前について$map の値が -1 でないならハッシュの項目を生成します (なぜ while ループがあるかというと、A は B、B は C にのような書き換えは全部たどってから生成するため)。これを使って生成した結果は次のような感じです。

```
$normalize = {
  'Akana; Jody Be' => 'Akana; Jody', // 4
  'Andre; Bart K.' => 'Andre ; Bartley K.', // 8
  'Andre; Bartley K' => 'Andre ; Bartley K.', // 8
  'Andre; Bartley K.' => 'Andre ; Bartley K.', // 8
  'Andre;Bartley K.' => 'Andre ; Bartley K.', // 8
```

```
    'Batailou; Jeremy' => 'Bataillou; Jeremy', // 24
    ...
  }
```

演習 22 上記のプログラムをコピーして動かしてみなさい。動いたら今度は「名寄せに加わらなかった名前の一覧」を表示してみなさい (または Unix コマンドでやってもよい)。その上で、さらに名寄せをうまくやるにはどういう工夫を加えたらいいと思うか、検討しなさい。

演習 23 演習 8 の検討結果を実装し、評価しなさい。

演習 24 名寄せプログラムは耐えられないとまでは言わないが、かなり遅い。もし badness の限界を 4 とするのなら、予め「2 つの文字列の長さの差+4」を editdist に与えることにより、これよりも値が大きくなりそうならすぐにやめて戻るようにすることで、時間を短縮できるはずである。具体的にどのようにしたらいいと思うか、検討しなさい。

演習 25 演習 10 の検討結果を実装し、評価しなさい。

11 第 3 回レポート課題

2 つ以上の演習のプログラムを選んで「プログラムを実際に動かす」課題をおこない、レポートとして提出しなさい。レポートではプログラムごとに次の内容を含むこと。

- 学籍番号、氏名、提出日付 (これらは最初に 1 回でよい)
- 問題の再掲
- 作成したプログラムとその説明
- 実行結果と考察

とくに考察において、「ロジックのどこが難しいか、どのように工夫したか」をきちんと書くこと。レポート課題は次回授業開始時までに、`gssm.k-kiso` に投稿すること。ただし★がついている問題 (易しい問題) を含む場合はは本日中。