情報科学 2007　久野クラス　＃6

久野　靖*

2007.11.14

はじめに

さて皆様、(時間) 計算量の課題はどうでしたか。計算量について一応知っているということは、まっとうなプログラムを組む人 (別にプロという意味はなく、アマチュアでも) にとって必須なので忘れないようにお願いします。さて今回は、「誤差」「数値積分」「求解」に続く数値解析の話題として、残っている次のものをやります。

- 連立一次方程式の数値解法
- 乱数とアルゴリズム

その前に、計算量の話題の補足を兼ねて演習問題解説を (演習 1 と演習 2 は測るだけですから、演習 3 からです)。

1　演習問題解説

1.1　演習 3: クイックソートの弱点

この演習は、例題で示した quicksort のコードが整列ずみの配列に対しては遅いという弱点を実際に示し、それを改善するというものだった。まず「弱点を示す」方は、次のように 2 回ずつ整列してみればよい:

```
def test
  a = randarray(1000)
  bench(1) do quicksort(a, 0, 999) end
  bench(1) do quicksort(a, 0, 999) end
  a = randarray(2000)
  bench(1) do quicksort(a, 0, 1999) end
  bench(1) do quicksort(a, 0, 1999) end
  a = randarray(3000)
  bench(1) do quicksort(a, 0, 2999) end
  bench(1) do quicksort(a, 0, 2999) end
end
```

これを実行してみると、次のようになった:

個数	1,000	2,000	3,000
1 回目	0.0313	0.0625	0.1094
2 回目	1.9609	8.4375	20.4063

確かに、1 回目は $O(N)$ に近い (実際には $O(NlogN)$ のはず) が、2 回目はずっと遅くて $O(N^2)$ に近いようだ。

では改良は? それは、ピボット値を取る時にいつも「端っこ」から取っていたからまずいので (図 1)、代わりにランダムに取るようにすればよいだろう。

*筑波大学大学院経営システム科学専攻

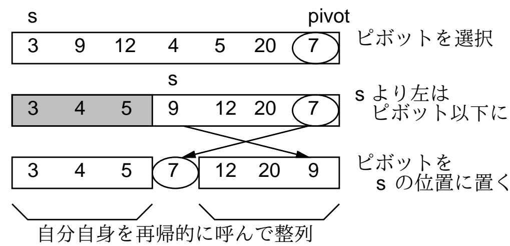

図 1: クイックソートによる整列

```
def quicksort(a, i, j)
  if j <= i then return end
  p = i + rand(j-i+1); a[p],a[j] = a[j],a[p] # select pivot at random
  pivot = a[j]; s = i
  i.step(j-1) do |k|
    if a[k] <= pivot then a[s],a[k] = a[k],a[s]; s = s + 1 end
  end
  a[j],a[s] = a[s],a[j]
  quicksort(a, i, s-1); quicksort(a, s+1, j)
end
```

コメントをつけた 1 行を挿入した。つまりここで、i〜j の範囲の整数 p をつランダムに選び、a[j] と a[p] を交換してからあとは同様に行っている。これを実行してみると、上の表と異なり、1 回目でも 2 回目でもほぼ同じ時間で整列が終わるようになっている。

1.2 演習 4: ビンソート

ビンソートとは、整列する値が整数であり、かつ範囲があまり広くない場合に利用できるのだった (図 2)。「範囲が広くない」という制約の理由は明らかで、その範囲全部をカバーする配列を作っている。たとえば値が 0〜1,000,000 だったら大きさ百万の配列が必要なわけで、それより大きいとちょっと実用上使えない。

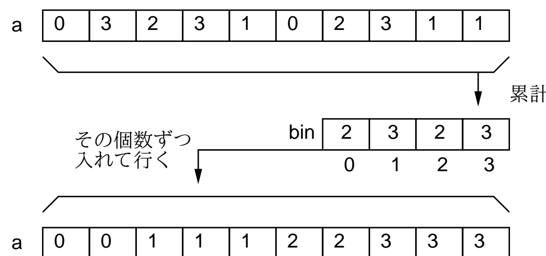

図 2: ビンソートによる整列

その替わり整列作業は簡単である。というのは、すべての要素を順に見ながら、「その値が何個あったか」を数えていく (それですべての値に対応する配列が必要だったわけだ)。そして、最後まで数え終わったら、今度は先頭から順にその個数ずつ値を作り出して元の配列に書き込んで行けばいい。コードを示す:

```
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
```

では、このアルゴリズムの時間計算量はどうなるだろうか。まずすべてのデータを順に操作するから $O(N)$ だが、それだけではない。最後に値の数を E とすると、大きさ E の配列を全部調べながら値を生成するので、このための時間も E が大きいと問題になる。なので、時間計算量は $O(N+E)$ になる。もっとも、値の範囲が 1 万でデータ数が百万ならほとんど $O(N)$ といってよい。

ただしもう 1 つ、E 個ぶんの配列を必要とすることも忘れてはいけない。アルゴリズムによっては、大量のメモリを使うことで時間を速くすることができるが、ビンソートはまさにその例である。ビンソートが要する記憶領域は元のデータ数 N と数えるための配列の数 E を併せたものだから、これを「**領域計算量**が $O(N+E)$ である」という。キーの範囲が広くなると、ビンソートは領域計算量の点でも不利になるわけである。

なお、これまでに出て来たアルゴリズムのほとんどは領域計算量 $O(N)$ だが、マージソートだけは「別の場所にマージして戻す」ので $O(2N)$ になっている。

1.3 基数ソート

では、基数ソートはどうだろうか。基数ソートでは、まずすべての値を 1 ビット目の 0/1 で振り分け、次に 2 ビット目の 0/1 で振り分け、という風にして、指定したビット数 (値が取る最大ビット数以上) ぶん繰り返すことで整列を行うのだった (図 3)。

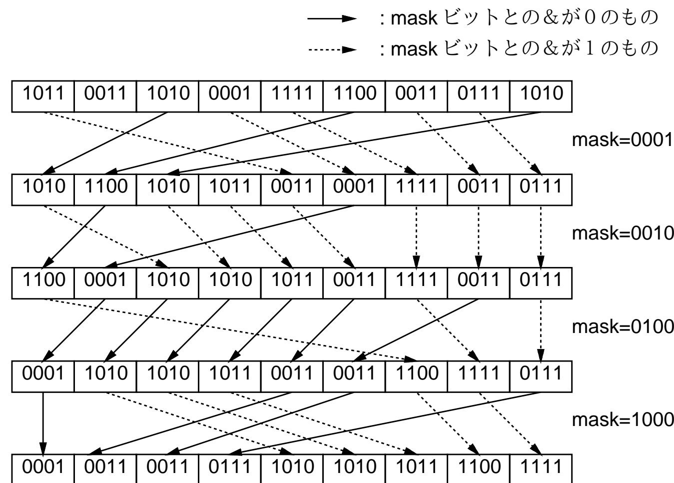

図 3: 基数ソートによる整列

これを Ruby プログラムにしたものを示す:

```
def radixsort(a, bits)
  b = Array.new(a.length); c = Array.new(a.length)
  bits.times do |pos|
    mask = 2**pos; bc = 0; cc = 0
    a.length.times do |i|
      if (a[i] & mask) == 0
        b[bc] = a[i]; bc = bc + 1
      else
        c[cc] = a[i]; cc = cc + 1
      end
    end
```

```
    bc.times do |i| a[i] = b[i] end
    cc.times do |i| a[bc+i] = c[i] end
  end
end
```

b と c は「振り分け」を行うための配列で、元の配列 a と同じ大きさで用意しておく。そしてループで指定したビット数の数だけ周回しながら、1、2、4、…を変数 `mask` に入れていく。その中では a の各要素について、`a[i]&mask` が 0 かどうか (`&`はビット毎の and 演算) で 0/1 を振り分け、配列 b か c に値を入れている。それが終わったら b と c に入れたものを a にコピーし戻している。

そういうわけで、キーの範囲を E とすると、キーのビット数は $\log E$ なので、基数ソートの時間計算量は $O(N \log E)$ ということになる。もっとも、多くの場合、ビット数は最大でも 32 とか 64 程度なので、これを定数と思えば $O(N)$ の時間計算量と思ってもよい (Ruby で大きな整数を使った場合はこの限りでない)。また、領域計算量は $O(2N)$ つまり作業配列 b と c のぶんが必要 (a のほかに b と c があったら 3 倍と思うかも知れないが、ケチるなら b と c を 1 つの配列にして「前から」と「後ろから」データを詰めて行けばよい)。

1.4 整列アルゴリズムの時間計測

各種整列アルゴリズムの計算時間を N を変えて計測してみた (マシンは皆様のマシンとは違う)。

表 1: 単純挿入法とバブルソートの所要時間

データ数 (10^3)	1	2	3	5	10	20	30	50
単純挿入法	305	1,195	2,695	7,453	-	-	-	-
バブルソート	1,547	6,500	14,523	40,836	-	-	-	-
クイックソート	31	55	94	156	367	781	1,273	2,461
マージソート	23	55	94	148	328	719	1,094	1,960
ビンソート	8	9	11	13	20	34	47	74
基数ソート	40	80	121	202	404	808	1,207	2,020

$O(N^2)$ のアルゴリズムである単純挿入法、バブルソートは N が大きくなると急激に遅くなって役に立たなくなる。一方、$O(N \log N)$ のマージソートとクイックソートは十万くらいのデータであれば十分実用になるだろう。

ところで、ビンソートと基数ソートはどうだろうか。ビンソートが異常に速いことは分かるが… もっと大きい N で測ってみると、基数ソートもクイックソートやマージソートより勝っているのがわかるはずである。

1.5 最大公約数

2 つの数 $M, N (M < N$ とする) の最大公約数のベタなバージョン (M からカウンタを 1 ずつ減らして行き、それで両者が割り切れることをチェックする方法) は当然 $O(M)$ になる。

```
def gcdenumerate(x, y)
  min = x; if min > y then min = y end
  (min-1).step(1, -1) do |i|
    if x % i == 0 && y % i == 0 then return i end
  end
end
```

ではおなじみ「大きい方から小さい方を引いて行く」バージョンはどうなのだろう。

```
def gcd(x, y)
  while x != y do
    if x > y
```

```
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

最善の場合は $M = N$ のときで、すぐ終わる。最悪の場合は $M = 1$ のときで、$N - 1$ 回引き算をしないと終わらない。平均は…? そこで、乱数を使って実験してみた。

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.0859375
bench(10000) do gcd(rand(1000)+1, rand(1000)+1) end → 0.1640625
bench(10000) do gcd(rand(10000)+1, rand(10000)+1) end → 0.2734375
bench(10000) do gcd(rand(100000)+1, rand(100000)+1) end → 0.4453125
```

これをみると、値が 10 倍ずつ大きくなるときに N に比例よりは少なく、しかし一定よりはやや多く、時間が増えている。するとたとえば $(logN)^2$ とかだろうか。しかし、M と N の大きさが違うとどうだろう?

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.0859375
bench(10000) do gcd(rand(100)+1, rand(1000)+1) end → 0.203125
bench(10000) do gcd(rand(100)+1, rand(10000)+1) end → 1.328125
bench(10000) do gcd(rand(100)+1, rand(100000)+1) end → 12.1171875
bench(10000) do gcd(rand(100)+1, rand(1000000)+1) end → 130.546875
```

おやおや、すごく遅く…それは当然で、N が M の 1000 倍なら 1000 回引く必要があるわけだから。ということは M と N の比率も掛ける必要がありそうだ。すると $O(\frac{N}{M} \times (\log N)^2)$ といった感じか? しかし、始めの方は比率が 10 倍になっても時間は 10 倍にならないが、比率が大きくなると 10 倍に近づき、超えている。するともう 1 個項を増やして $O(\frac{N}{M} \times \log \frac{N}{M} \times (\log N)^2)$ という感じだろうか (久野はこれの専門家でないのでこれ以上解析できませんでした、すみません)。

では、引き算の代わりに剰余演算を使うユークリッドの互除法では?

```
def gcdeuclid(x, y)
  while true do
    if x > y
      x = x % y; if x == 0 then return y end
    else
      y = y % x; if y == 0 then return x end
    end
  end
end
```

これでまずアンバランスな方から測ってみる。

```
bench(10000) do gcdeuclid(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcdeuclid(rand(100)+1, rand(1000)+1) end → 0.046875
bench(10000) do gcdeuclid(rand(100)+1, rand(10000)+1) end → 0.046875
bench(10000) do gcdeuclid(rand(100)+1, rand(100000)+1) end → 0.0390625
```

まったく変わらない。それはそうで、CPU の割算命令の時間は値が変わってもほとんど一定時間で実行されるから (ただし Ruby で多倍長演算が必要なくらい大きい値になるとそれは 1 命令ではできなくなるのでこのようには行かない)。だから $\frac{N}{M}$ の要素はなし。では値の大きさの影響は?

```
bench(10000) do gcdeuclid(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcdeuclid(rand(1000)+1, rand(1000)+1) end → 0.0546875
bench(10000) do gcdeuclid(rand(10000)+1, rand(10000)+1) end → 0.0625
bench(10000) do gcdeuclid(rand(100000)+1, rand(100000)+1) end → 0.078125
bench(10000) do gcdeuclid(rand(1000000)+1, rand(1000000)+1) end → 0.0859375
```

10 倍になるごとに一定ずつ増えて行くようだ。それは、ループを 1 回まわるごとにだいたい一定比率で大きい方の値が小さくなるだろうから、ループ周回数は値の大きさの log に比例するだろうから。つまり $O(logN)$ ですね。と、まあこういう検討をして頂ければよかったのでして。

1.6 フィボナッチ数

やってみればすぐ分かるが、再帰的定義そのままのフィボナッチ数の計算は、$fib(N)$ の計算に $fib(N-1)$ と $fib(N-2)$ を実行し、それがさらに $fib(N-2)$ と $fib(N-3)$、$fib(N-3)$ と $fib(N-4)$ をそれぞれ呼び、というふうに「倍々」になるので、$O(2^N)$。これは非常にのろい。ループで計算する場合は当然 $O(N)$。

```
def fibloop(n)
  x0 = 1; x1 = 1
  n.times do
    t = x0 + x1; x0 = x1; x1 = t
  end
  return x1
end
```

一応計測してみよう。

```
bench(10000) do fibloop(10) end → 0.0625
bench(10000) do fibloop(100) end → 0.8359375
bench(10000) do fibloop(1000) end → 11.9140625
```

「10 倍になるごとに時間も 10 倍」から外れているが、これは fib(50) くらいからもう結果が多倍長計算が必要な大きさになってしまうから。

では、行列計算を使ってなおかつ行列の N 乗の計算を工夫する方法だとどうだろう。その「工夫」の漸化式を再掲する。

$$Q^n = \begin{cases} E & (n = 0) \\ QQ^{n-1} & (n \text{ が正の奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が正の偶数}) \end{cases}$$

このプログラムを掲載しておく。2×2 行列の積、行列とベクトルとの積を下請けに用意して、それを用いて上の漸化式を使った N 乗を定義し、最後にそれを用いてフィボナッチ数を計算している。

```
def mat22multvec(a, v)
  c = [0, 0]    # 結果用の 1 次元配列
  c[0] = a[0][0]*v[0] + a[0][1]*v[1]
  c[1] = a[1][0]*v[0] + a[1][1]*v[1]
  return c
end

def mat22mult(a, b)
  c = [[0,0],[0,0]]    # 結果用の 2 次元配列
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
```

```
    c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
    return c
  end

  def mat22power(a, n)
    if n == 0
      return [[1,0],[0,1]]  # 2x2 単位行列
    elsif n % 2 == 1
      return mat22mult(mat22power(a, n-1), a)
    else
      b = mat22power(a, n/2); return mat22mult(b, b)
    end
  end

  def fibmat(n)
    return mat22multvec(mat22power([[1,1],[1,0]], n), [1,1])[0]
  end
```

実験してみよう。

```
irb(main):101:0> bench(10000) do fibmat(10) end → 0.703125
irb(main):102:0> bench(10000) do fibmat(100) end → 1.46875
irb(main):103:0> bench(10000) do fibmat(1000) end → 2.921875
```

10 倍になるごとにだいたい一定ずつ時間が増えている (最後の方は多倍長になるので…)

この場合、「正の奇数」が選ばれるのは N を 2 進表現したときに「1」が現れる回数と等しい。「正の偶数」が選ばれるのは N を (奇数なら 1 を引きながら) 半分ずつにしていき 0 になるまでやるので、2 進表現の桁数つまり $\log N$。上記「1」の数は平均すると桁数の半分くらいだから $\frac{\log N}{2}$、すると全体として $O(\log N)$ となる。

1.7 組み合わせの数

再帰的定義はフィボナッチと同様、倍々の呼び出しになるので、$O(2^N)$。では「パスカルの三角形」の場合はどうだろうか。再掲すると次のものを計算するわけだ。

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 ...
```

これを N 段目まで作るとなると、要素数が $\frac{N(N+1)}{2}$ あるから、計算量としては $O(N^2)$ ということになるはず。コードを書いてみた。

```
def combarray(n, r)
  a = Array.new(n+1, 1)
  1.step(n) do |i|
    (i-1).step(1, -1) do |k| a[k] = a[k-1] + a[k] end
    # p(a)
  end
  return a[r]
end
```

これはどうやっているかというと、N 段までのパスカルの三角形を作るので、要素数 $N+1$ の「1」ばかりが詰まった配列を用意し、それを図 4 のように隣の要素どうし足すことを繰り返して行く。内側ループを値が大きい方から回っているのは、そうしないと「1 つ前の値」を使うことができないから。

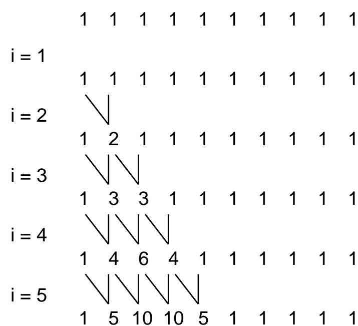

図 4: パスカルの三角形の計算

では時間を計測してみる。

```
bench(10000) do combarray(10,5) end → 0.609375
bench(10000) do combarray(20,10) end → 2.2578125
bench(10000) do combarray(30,15) end → 4.9765625
```

確かに $O(N^2)$ のようだ。ところで、前にやった「普通に掛け算する」バージョンはどうだろう。

```
def combloop(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (n - r + i) / i
  end
  return result
end
```

これならループが r 回だから ($r \sim N$ として)$O(N)$ となる。

```
bench(10000) do combloop(10,5) end → 0.0703125
bench(10000) do combloop(20,10) end → 0.1171875
bench(10000) do combloop(30,15) end → 0.2265625
```

たしかにずっと速い。だから結局、ループで普通に計算するのがいいというオチでした。が…「何回もさまざまな値を」使うのであれば、パスカルの三角形を「2 次元配列」の上で作成しておいて、そこから値を取り出すようにするのが良さそうである。

2 連立方程式の数値解法

2.1 消去法

連立方程式とは何かとかいう説明はいいですね? たとえば次に 3 元連立一次方程式の例を示す。

$$
\begin{array}{ccccccc}
2x_0 & + & 3x_1 & + & 2x_2 & = & 1 \\
2x_0 & + & 5x_1 & + & 4x_2 & = & 4 \\
4x_0 & + & 8x_1 & + & 8x_2 & = & 7
\end{array}
$$

これをどうやって解きますか? まず、一番上の式を 2 番目、3 番目から引く (3 番目については 2 倍してから引く) ことで x_0 の係数を消去できる。

$$\begin{array}{ccccccc} 2x_0 & + & 3x_1 & + & 2x_2 & = & 1 \\ & & 2x_1 & + & 2x_2 & = & 3 \\ & & 2x_1 & + & 4x_2 & = & 5 \end{array}$$

同様にして 2 番目の式を 3 番目から引くことで x_1 の係数を消去。

$$\begin{array}{ccccccc} 2x_0 & + & 3x_1 & + & 2x_2 & = & 1 \\ & & 2x_1 & + & 2x_2 & = & 3 \\ & & & & 2x_2 & = & 2 \end{array}$$

これで $x_2 = 1$ が求まったのでそれを 2 番目、1 番目に代入できる。

$$\begin{array}{ccccc} 2x_0 & + & 3x_1 & = & -1 \\ & & 2x_1 & = & 1 \\ & & x_2 & = & 1 \end{array}$$

引続き $x_1 = 0.5$ が求まったのでそれを 1 番目に代入できる。

$$\begin{array}{ccc} 2x_0 & = & -2.5 \\ x_1 & = & 0.5 \\ x_2 & = & 1 \end{array}$$

これで $x_0 = -1.25$ と全て求まった。このように (1) 順に係数を消去していき (前進消去)、最後まで行ったら (2) 値を逆順に代入していく (後退代入) ことで連立一次方程式を解くことができる。これを **Gauss の消去法**と呼ぶ。

実際にこれをプログラムで扱う場合は、変数の名前はどうでもいいわけなので、係数だけをデータとして与える。ここでは 2 次元配列を直接手で打って与えることにする (連立一次方程式なので当然、幅が高さより 1 大きくなければいけない)。Ruby プログラムは、単にこれの上で前進消去、後退代入を行うだけ。そうすると、各行の右端の値が方程式の解になる。ではプログラムを次に示す:

```
def gauss(a)
  n = a.length
  n.times do |i|
    (i+1).step(n-1) do |j|
      r =  a[j][i] / a[i][i].to_f
      i.step(n) do |k| a[j][k] = a[j][k] - a[i][k]*r end
    end
    # p(a)
  end
  (n-1).step(0, -1) do |i|
    a[i][n] = a[i][n] / a[i][i]; a[i][i] = 1.0
    i.times do |j| a[j][n] = a[j][n] - a[j][i]*a[i][n] end
    # p(a)
  end
  return a
end
```

なお「p」というのは配列などを irb の出力と同じように表示してくれるメソッドで、途中経過を見たいときはコメントを外してやるとよい。ではこれを動かしたところを見よう:

```
irb(main):118:0> gauss([[2,3,2,1],[2,5,4,4],[4,8,8,7]])
=> [[1.0, 3, 2, -1.25], [0.0, 1.0, 2.0, 0.5], [0.0, 0.0, 1.0, 1.0]]
```

確かに解が求まっている。ところで、つぎのデータだとどうだろうか。

```
irb(main):121:0> gauss([[2,3,2,1],[2,3,4,4],[4,8,8,7]])
=> [[1.0, 3, 2, NaN], [0.0, 1.0, 2.0, NaN], [0.0, NaN, 1.0, NaN]]
```

何がいけないのだろう。この場合、2 番目の式から 1 番目の式を引くと最初の 2 つの係数がともに 0 になる。そのため、係数 0 を消去しようとして 0 で割り算してしまう。でもこの連立方程式が解けないということは全くなくて、次のように順番を入れ換えれば問題なくできる。

```
irb(main):122:0> gauss([[2,3,2,1],[4,8,8,7],[2,3,4,4]])
=> [[1.0, 3, 2, -0.25], [0.0, 1.0, 4.0, -0.5], [0.0, 0.0, 1.0, 1.5]]
```

もちろん人間が順番を考えるようでは困るので、実際にはプログラムで次に消去しようとする係数が 0 だったら別の行と入れ換えてから消去を行う必要がある。さらに、完全に 0 でなかったとしても、非常に 0 に近い (絶対値の小さい) 値だと誤差が出やすくなるので、常に絶対値の大きい行を選んで来てそれで消去を行うのがよい。これを**ピボット選択** と呼ぶ。[1]ただし、ピボット選択を行おうとしても、すべての係数が 0 で選べないこともある。これは、方程式がもともと不定/不能の場合に起きる。

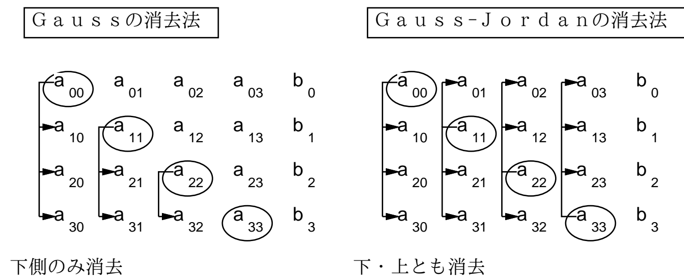

図 5: Gauss-Jordan の消去法

ところで、Gauss の消去法では処理が前進消去と後退代入の 2 つの段階に分かれていたが、消去のときにこれまでのように「自分より下の行について消去する」代わりに「自分以外のすべての行について消去」することで 1 段で解を求めることができ、プログラムがやや簡単になる (図 5)。これを **Gauss-Jordan の消去法**と呼ぶ (ただし計算量はやや不利になる)。

演習 1 上の例題をそのまま打ち込んで動かせ。動いたら、次のように改造してみよ。

a. Gauss-Jordan の消去法に直し、プログラムが短くなることを確認する。
b. ピボット選択を入れ、例題で駄目だったデータが扱えることを確認する。
c. 方程式が不能/不定のときにその旨表示する機能を追加する。

2.2 反復法

連立一次方程式の消去法とは別の原理による数値解法として、反復法と呼ばれるカテゴリのものがある。これは次の原理による。ここまでに見て来たような連立一次方程式を行列・ベクトルを使って書き直すと次のように書くことができる。

$$A\vec{x} = \vec{b}$$

ところで、行列 A を下三角行列 L、対角行列 D、上三角行列 U の 3 つに分解して考えると次のように変形できる。

$$(L+D+U)\vec{x} = \vec{b}$$

$$D\vec{x} = \vec{b} - (L+U)\vec{x}$$

$$\vec{x} = D^{-1}\{\vec{b} - (L+U)\vec{x}\}$$

[1]ただし、$2x+3y=1$ と $200x+300y=100$ とは同じ方程式なので、実際にはまず各行の係数を絶対値の最大が 1 になるように定数倍してそろえておき、それから最大を選択する方がよい。これを**スケーリング**と呼ぶ。

D は対角行列だから逆行列は各要素を逆数にすればいいだけなのに注意。さて、連立一次方程式を解くということは、この式を満たす $\vec{x}$ を求めることに等しい。ところで、この式を漸化式と考え、適当な $\vec{x_0}$ から始めて次の式により値を計算していくものとする。

$$\vec{x_{i+1}} = D^{-1}\{\vec{b} - (L + U)\vec{x_i}\}$$

ここで $\vec{x_i}$ が収束すれば、つまり値が変化しなくなれば、その値 $\vec{x_*}$ は連立一次方程式の解となっている。これを **Jacob 法**と呼ぶ。なお、上の漸化式は常に収束するとは限らないが、行列が対角優位 (対角成分が他の成分に比べて相対的に大きい) の場合には収束することが知られている (厳密な収束条件についてはその筋の本を調べてください)。

これを Ruby プログラムにしたものを示しておく。上の漸化式の計算は面倒そうだが、よく考えれば次の手順で行えることに注意。

- b_i は $a_{i,n}$ に入っている。
- そこから、$\sum_j a_{j,i}x_j$ を引く (ただし対角要素は飛ばす)。
- 最後に $a_{i,i}$ で割る

収束は $\vec{x_i}$ の各成分の前回との差の絶対値の和が 10^{-8} 以下になったことで判定し、100 回反復しても収束しない場合はあきらめるようにした。

```
def jacobi(a)
  n = a.length; x = Array.new(n, 0); count = 0
  while true do
    x1 = Array.new(n, 0); d = 0.0
    n.times do |i|
      v = a[i][n].to_f
      n.times do |j| if j != i then v = v - a[i][j]*x[j] end end
      x1[i] = v / a[i][i]; d = d + (x1[i]-x[i]).abs
    end
    x = x1; count = count + 1
    # p(x)
    if d < 0.000000001 then return x end
    if count > 100 then return nil end
  end
  return x
end
```

では実際に動かしてみよう。

```
irb(main):080:0> jacobi([[2,3,2,1],[2,5,4,4],[4,8,8,7]])
=> nil
```

あれ、収束しないようだ。別のもので試すと…

```
irb(main):079:0> jacobi([[5,2,-1,19],[2,-3,2,-1],[1,1,-2,8]])
=> [2.99999999991689, 0.999999999949859, -1.99999999987636]
```

これは確かに大丈夫のようだ。なお、Jacob 法では「次の」ベクトルを完全に計算し終わってから現在のものと入れ換えるが、上の例で言うと `x1` を使わないで「`x1[i] = …`」を「`x[i] = …`」に直してしまい、計算し終わった成分は以後その新しい値を使う方法を **Gauss-Seidel 法**と呼び、その方が収束が速いことが知られている。

演習 2 上の例題を打ち込んでさまざまなデータで動かし、動いたら次のような検討を行え。

a. どのような場合に解が収束する/しないかを検討。
b. 途中経過を打ち出させ、収束しない場合のふるまいや、収束する場合どのように収束するかを検討。
c. Gauss-Seidel 法に直した場合、収束までの計算回数やふるまいがどのように違うかを検討。

3　乱数とランダムアルゴリズム

3.1　乱数とは

乱数 (random numbers) とは、「でたらめな数」である。ではあんまりだからもっと正確に言うと、乱数列とは「ある分布に従う、互いに独立な事象を表す、確率変数の実現値の列」を言い、乱数とはその中の 1 つの値を言う。互いに独立ということは、ある点までの乱数列が分かったからといって、次の乱数がいくつであるかは予測できないことを意味する。

また、**分布** (distribution) とは、どの範囲の値がどのくらい出現しやすいかを表す (図 6)。たとえば、区間 $[0, 1)$ の一様分布であれば、乱数の範囲は 0 以上 1 未満で、その間のどの数も同じくらいの確からしさで出現する。これを**一様乱数**という。また、偏りのないサイコロを振って出る目の数は 1 以上 6 以下の整数値だが、どの数も同じ確率で出現するため、これも一様乱数である。

図 6: 乱数と分布

この他によく使われる乱数としては、正規分布に従う**正規乱数**がある。正規分布は中央が一番高く両側にすそを引いたツリガネ形の分布であり、試験の偏差値などでおなじみである (試験が受験者の集団に対して易しすぎたり難しすぎたりヘンな問題だったりすると、分布が綺麗な正規分布でなくなるので偏差値による順位推定が役に立たなくて問題だったりする)。

3.2　疑似乱数

疑似乱数 (pseudorandom numbers) とは、計算機である計算を順次行うことによって生成される数値の列だが、乱数列のように思えるようなものを言う。既にさんざん学んで来たように、計算機による計算はプログラムによって完全に決定的に決まるので、「でたらめな」数を生成するのは思いのほか難しい。

疑似乱数のアルゴリズムとして知られているものには、たとえば次のものがある。

- 自乗採中法 — w ビットの数を 2 乗すると $2w$ ビットになるので、そこから中央付近の w ビットを取り出して「次の数」とする。これを繰り返すことで w ビットの乱数列を生成する。古くからあるがあまり良くはない。
- 線形合同法 — $x_{i+1} = (x_i \times a + c) mod m$ により次々に値を生成していく。良い疑似乱数とするためには、パラメタ a, c, m の選定に注意が必要。
- メルセンヌツイスター (MT) — 1997 年に松本　眞、西村拓士が開発した乱数アルゴリズム。

どのようなアルゴリズムでも、計算方法が決まっている以上、順次 x_i を生成していくうちに前に出て来たものが再度現れたら、それ以降の列は前と同じものの繰り返しになる。これを周期といい、もちろん周期が長いものが望ましい。MT は 32 ビットで $2^{19937} - 1$ という長い周期を保証できる点で画期的だった。周期の他に統計的独立性の検定などもクリアしている。

Ruby では `rand(`N`)` で 0 以上 N 未満の整数の一様乱数、引数なしの `rand()` で区間 $[0, 1)$ の実数一様乱数が得られるが、そのアルゴリズムたぶん線形合同法で、MT ほど良いとは言えないらしい。とりあえず、シミュレーションなどに使ってみる分には問題ないと思われるが。

このほか、最近では OS(コンピュータ上で常に稼働している基本ソフトウェア) が乱数機能を提供してくれている場合が多い。OS の乱数機能は、外部割り込み (ユーザのキーボード入力など) 等に基づいた「ランダムさ」を活用するので、疑似乱数のような周期の問題がない。ただし速い速度で乱数を生成消費すると「ランダムさ」の供給が追い付かなくなることがあるという弱点がある。また、最近では CPU チップ自体に物理的な乱数発生装置を持つものもある。Mac OS-X では「`new FileReader("/dev/random")`」により OS の乱数機能からバイト列を読み出すストリームを作成できる。

3.3 ランダムアルゴリズム

ランダムアルゴリズム (randomized algorithm) とは、(疑似) 乱数を活用して、ランダムなふるまいを持たせたアルゴリズムを言う。これに対し、通常の決定的な動作を行うアルゴリズムは**決定的アルゴリズム** (deterministic algorithm) と呼ばれる。

ランダムアルゴリズムは、設計によっては決定的アルゴリズムよりすぐれた性能を持たせることができる。たとえば、1 億要素の配列で「半分には値 a が入っているがその場所は分からない」場合と「まったく値 a が入っていない」場合とがあり、どちらであるかを判断する必要があるものとする。決定的アルゴリズムでは、たとえば先頭から順番に値 a があるかどうか見て行くことになるだろうが、値 a が全部後半に詰まっている可能性もあるので、最悪で 5 千万要素を見る必要がある。後ろから順に見るとしても、値 a が全部前半に詰まっているかも知れないので同じである。ここで、乱数を用いて 1 億の位置からランダムに 1 つ選び、その値が a かどうかを判断することを 1 万回繰り返したとする。その結果 1 回も値 a に遭遇しなければ、「値 a はない」と判断してまず問題ない。この判断が間違っている確率は $\frac{1}{2^{10000}}$ であり、それはこの計算をするコンピュータが故障する確率よりはるかに小さいのだから。

このような、微小だが 0 でない「間違う」確率を持ったアルゴリズムを**モンテカルロアルゴリズム** (Monte Carlo algorithms) という。[2]これに対し、間違うことはないが運が悪い場合に性能が低下するアルゴリズムを**ラスベガスアルゴリズム** (Las Vegas algorithms) という。[3]たとえば、クイックソートはピボットの選択が悪いと性能が低下することを述べた。そこで、どの要素をピボットとして用いるかを乱数で決めるようにすると、これはラスベガスアルゴリズムとなる。なぜなら、乱数がすべて「悪い要素 (その区間の最大や最小の要素)」を選び続ける確率は非常に小さいので、よほど運が悪くない限り高速に整列が行え、そして運が悪い場合は実行時間が長く掛かることになるが、整列はやはり正しく行えるからである。

3.4 モンテカルロ法

モンテカルロ法とは、**シミュレーション** (simulation) などにおいて乱数を活用する手法を言う。たとえば、交通の流れを実際に観察する代わりに、乱数を用いてランダムに車を (プログラム内で) 発生させ、それらの車がどのように流れて行くかを見ることで、さまざまな方法で交通信号を制御してよい方法を見つけ出すことなどができる。

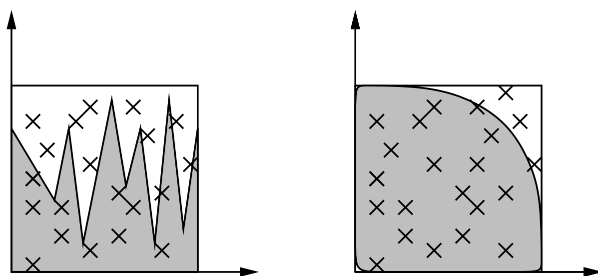

図 7: モンテカルロ法による数値積分

また、モンテカルロ法は数値積分にも使うことができる (図 7)。すなわち、積分する範囲の関数値の最大より大きい値を選んで長方形の領域を考え、その範囲内に乱数で多数の点を打ち、関数値より下にある点の比率を求める。積分とは「その関数の下側の面積を求める」ことだから、長方形の面積にその比率を掛けたものが積分値 (の近似値) として使える。

そんな面倒なことをするよりシンプソンのアルゴリズムでいいのでは? しかし、シンプソンのアルゴリズムなどは、対象とする関数が連続かつ微分可能 (なめらか) でないと使えない。そのような性質が期待できないような分野では、モンテカルロ法が有力な手法の 1 つとなる。

たとえば半径 1 の四分の一円の面積を求めて (それを 4 倍することで) π の近似値を計算してみよう (もちろん円周は十分連続かつ微分可能だけれどそれは置いておいて)。

```
def pirandom(n)
  count = 0
```

[2]モンテカルロはヨーロッパにあるカジノで有名な都市の名前。
[3]ラスベガスは米国にあるカジノで有名な都市の名前。

```
  n.times do
    x = rand(); y = rand()
    if x**2 + y**2 < 1.0 then count = count + 1 end
  end
  return 4.0 * count / n
end
```

実行させてみると次の通り。

```
irb(main):002:0> pirandom 10000
=> 3.13
irb(main):003:0> pirandom 100000
=> 3.14444
irb(main):004:0> pirandom 1000000
=> 3.141604
```

有効数字 3〜4 桁ではしょうもないと思いますか? 実際には、3〜4 桁の有効数字が得られれば十分な仕事というのは結構多い。来年の GDP の生長率が 10.1 だろうと 10.2 だろうと 3 桁目はさして重要ではないでしょう?

演習 3 モンテカルロ法で数値積分を行うときの、精度 (有効桁数) と試行の数との関係について考察せよ。円周率の例題を活用してもよいが、できれば別の関数を積分するプログラムを作って検討することが望ましい。

演習 4 乱数で点を打つのでなく、均一の細かさで格子点上に点を打って調べることが考えられる。この方法とモンテカルロ法の善し悪しについて考察せよ。何らかのテストプログラムを作って動かすこと。

3.5 乱数とゲーム

最後にお楽しみの話題としてゲームに言及しておこう。ゲームの中には、将棋や囲碁のように (先手後手を決める以外は) ランダム性を使わないものもあるが、多くのゲームでは (サイコロやカードのシャッフルなどを通じて) ランダム性を採り入れている。ランダム性を採り入れることで、ゲームの「場面」が毎回違ったものになり新鮮さが保たれ、また複数プレーヤで行う場合に「上手下手」以外の要因が入って下手な人にも勝つチャンスが生まれ勝負の行方に興味が持てるようになる。

ここでは簡単なゲームとして「数当て」を作ってみる。そのルールは次の通り。

- 計算機は内部で 4 桁の数を「思い浮かべ」る (4 桁の中に重複があるかも知れない)。
- プレーヤはその 4 桁の数を「当てる」ことをめざして、自分も 4 桁の数を入力する。
- 計算機は 2 つの 4 桁の数を照合して、「同じ位置に同じ数がある (これをヒットと呼ぶ)」個数と、「同じ数があるがただし違う位置にある (これをブローと呼ぶ) 個数とを数えて知らせる。
- プレーヤはその情報を見て再度チャレンジする。
- 10 回以内のチャレンジで当たればプレーヤの勝ち、さもなければプレーヤの負けとする。

Ruby プログラムを次に示す。ゲームの「やりとり」を行うために、キーボードから 1 行入力するメソッド `gets` を使っている。また、`puts` は改行してしまうので、プロンプト (「入力してください」という文字列) を出力するのに `printf` を使った。あと、文字列は配列と同様に添字を指定することで「何文字目」が取り出せたり、添字範囲を指定することで部分文字列が取り出せることも利用している。

```
def kazuate
  a = (rand(10000)+10000).to_s[1..4]; count = 0
  while true do
    printf("your guess? ")
    s = gets; hit = 0; blow = 0
    4.times do |i|
```

```
      4.times do |j|
        if s[i] == a[j] then
          if i == j then hit = hit + 1 else blow = blow + 1 end
        end
      end
    end
    if hit == 4 then puts "you win!"; return end
    count = count + 1
    if count > 9 then puts "you lose! answer = #{a}."; return end
    puts "hit = #{hit}, blow = #{blow}."
  end
end
```

4 桁のランダムな数を作る方法がちょっと分かりづらいかも知れないが、「0〜9999 の乱数を作り、10000 を足して 10000〜19999 の範囲にしてから文字列に変換し、1〜4 文字目 (先頭が 0 文字目) を取る」ことで「'0000'〜'9999' の文字列を作っている。あとは、プレーヤが入力した文字列とすべての位置の組合せで照合し、ヒットとブローの数を数えている。

演習 5 乱数を使ったゲームを何か作ってみよ。上の例題の改良 (改造) 版でもよい。

A 本日の課題 6A

「演習 1」または「演習 2」の小課題から 1 つ選び、今日中に久野までメールで送ってください。

1. Subject: は「Report 6A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 1」または「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 検討結果のまとめ (とできれば簡単な考察)
5. 以下のアンケートの回答。

   Q1. 連立一次方程式がプログラムで解けるという点を理解しましたか。

   Q2. プログラムにおける乱数の活用法について納得しましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B 次回までの課題 6B

次回までの課題は「演習 1」〜「演習 5」(の小課題) から 2 つ以上選んで報告することです。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは授業開始時刻の **10 分前** までに久野までメールで送付してください。

1. Subject: は「Report 6B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 2 次元配列を操作するプログラムが作れるようになりましたか。

   Q2. 乱数を活用したプログラムが作れるようになりましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。