情報科学 2006　久野クラス　＃5

久野　靖*

2006.11.17

はじめに

今回はアルゴリズム解析において重要な「計算量」の概念について学びます。前々回の課題のうち「整列」について解説があと回しになっていましたが、それは前回の内容が多くなりすぎたのと、計算量に関連してゆっくり説明したかったからです。というわけで今回の内容は次の通り。

- 整列のさまざまなアルゴリズム
- 計算量の考え方

1　演習問題解説

1.1　演習 2: 再帰関数

これは簡単に。関数の部分だけ示せばいいですよね。まず階乗。

```
def fact(n)
  if n == 0
    return 1
  else
    return n * fact(n-1)
  end
end
```

次はフィボナッチ数。

```
def fib(n)
  if n < 2
    return 1
  else
    return fib(n-1) + fib(n-2)
  end
end
```

次は組合せの数。

```
def comb(n, r)
  if r == 0 || r == n
    return 1
  else
    return comb(n-1, r) + comb(n-1, r-1)
  end
end
```

*筑波大学大学院経営システム科学専攻

最後に 2 進表現。

```
def binary(n)
  if n == 0
    return "0"
  elsif n == 1
    return "1"
  elsif n % 2 == 0
    return binary(n / 2) + "0"
  else
    return binary((n-1) / 2) + "1"
  end
end
```

まあこのあたりは、再帰に慣れて頂くということで簡単でよかったんじゃないでしょうか。

1.2 演習 6/7 — 素数の計算

素数については前回もやったが、それを配列を使ってもっと効率よく、という話だった。これもプログラムだけ示しておく。まず「見つかった素数を覚えておいてそれだけで割ってみる」方法。

```
def isprimememo(n, a)
  limit = Math.sqrt(n).to_i
  a.each do |i|
    if i > limit then return true end
    if n % i == 0 then return false end
  end
  return true
end

def primememo(n)
  a = []; puts 2
  3.step(n, 2) do |i|
    if isprimememo(i, a) then a.push(i); puts(i) end
  end
end
```

2 は別扱い。配列 a には今までに見つかった (2 以外の) 素数が入っていて、isprimememo ではこれを利用して渡された n を a の各要素で順次割ってみて素数かどうか調べる。割ってみる数が n の平方根を超えたらこれ以上やってもしかたないので「素数だ」と返事。割れたら「素数ではない」と返事。最後まで割れなかったら「素数だ」と返事。この下請けがあれば、あとは (2 は別扱いとして) 3 以上の奇数について素数なら a に追加するとともに出力するだけ。

もう 1 つの方法は「エラトステネスのふるい」と呼ばれるアルゴリズム。

```
def primesieve(n)
  limit = Math.sqrt(n).to_i
  a = Array.new(n+1, true)
  puts(2)
  3.step(n, 2) do |i|
    if a[i]
      puts(i)
      if i <= limit
        (i*2).step(n, i) do |j| a[j] = false end
```

```
      end
    end
  end
end
```

配列 a は最初から n+1 要素 (最大添字が n) ぶんすべて「はい」に初期化。2 は別扱いとして、3 以上の奇数 i について順次、もし a[i] が「はい」なら素数として出力し、その倍数番目の要素をすべて「いいえ」に書き換える。ただし i が n の平方根を超えたら印は全部つけ終わっているはずなので印つけは省略。

1.3 演習 4: さまざまな図形の生成

とりあえず、演習 4 の a と b についてだけ。さまざまな図形を描いた例を 1 つのプログラムでまとめて示す (図 1)。

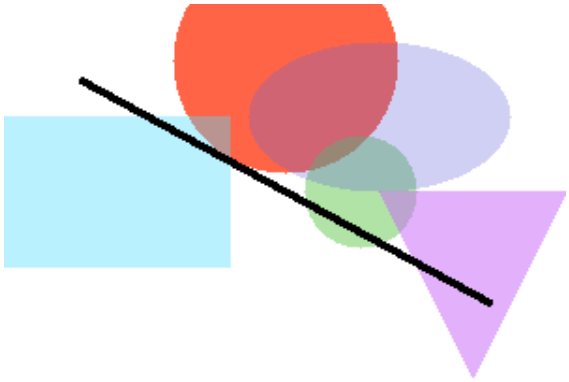

図 1: 生成されたさまざまな図形

まずレコード定義、画像の初期化と書き出しは前回と同じだが再掲しておく。

```
Pixel = Struct.new(:r, :g, :b)

def initimage
  $img = Array.new(200)
  200.times do |j|
    $img[j] = Array.new(300)
    300.times do |i| $img[j][i] = Pixel.new(255,255,255) end
  end
end

def writeimage(name)
  open(name, "w") do |f|
    f.puts("P3 300 200 255")
    200.times do |j|
      300.times do |i|
        p = $img[j][i]; f.puts("#{p.r} #{p.g} #{p.b}")
      end
    end
  end
end
```

さて次だが、色に「透明度」をつけて塗れるようにするには、現在の色と塗りたい色を α (透明度) に応じて比例配分すればよい。それを行う手続き setcolor を用意してあとはすべてこれを使うこととした。また、このとき指定した座標が配列の範囲を超えていたら「無視する」ことにした (縁に掛かった図形でも問題なく表示できるようにするため):

```
def setcolor(x, y, r, g, b, a)
  if x < 0 || x >= 300 || y < 0 || y >= 200 then return end
  $img[y][x].r = ($img[y][x].r * a + r * (1.0 - a)).to_i
  $img[y][x].g = ($img[y][x].g * a + g * (1.0 - a)).to_i
  $img[y][x].b = ($img[y][x].b * a + b * (1.0 - a)).to_i
end
```

円を描く fillcircle はこれを呼ぶように直した。あと、画面全体について調べるのでなく、XY 座標の最小〜最大の範囲内だけ調べるように直した (この方が大きい画面のときは効率がよいはず):

```
def fillcircle(x, y, rad, r, g, b, a)
  j0 = (y-rad).to_i; j1 = (y+rad).to_i
  i0 = (x-rad).to_i; i1 = (x+rad).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x)**2 + (j-y)**2 < rad**2 then setcolor(i, j, r, g, b, a) end
    end
  end
end
```

なお、XY 座標や半径に小数点つきの値が入れられても動作するように、調べる範囲を計算したときに結果を .to_i で整数にしている。

長方形を描く fillrect は、円よりもっと簡単で、単にその範囲全部を setcolor するだけ。回転させたい場合は後の「直線」を援用する。

```
def fillrect(x, y, w, h, r, g, b, a)
  j0 = (y-0.5*h).to_i; j1 = (y+0.5*h).to_i
  i0 = (x-0.5*w).to_i; i1 = (x+0.5*w).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i| setcolor(i, j, r, g, b, a) end
  end
end
```

楕円を描く fillellipse は、円と同様でただし縦横をそれぞれ縦横の半径で割ってから半径 1 の円に入っているかどうかで判定すればよい:

```
def fillellipse(x, y, rx, ry, r, g, b, a)
  j0 = (y-ry).to_i; j1 = (y+ry).to_i
  i0 = (x-rx).to_i; i1 = (x+rx).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x).to_f**2/rx**2 + (j-y).to_f**2/ry**2 < 1.0
        setcolor(i, j, r, g, b, a)
      end
    end
  end
end
```

三角形を描く filltriangle は、凸多角形を塗る fillconvex というのを作ってそれを呼ぶだけにする。なお k fillconvex は X 座標、Y 座標をそれぞれ配列で渡し、最後には最初と同じ要素を重複して入れておくことにする。また、点を指定する順序は「左回り」である必要がある。これらの理由は後述。

```
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b, a)
```

```
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b, a)
end
```

fillconvex だが、座標の範囲は配列に入っている X 座標や Y 座標の最大と最小を求める必要がある。これは前回演習でやったようなものだが、実は配列にはメソッド max と min があって最大と最小を計算してくれる。その後、各点についてそれが図形の内側にあるなら塗る。図形の内側にあるかどうかは isinside で判定する。

```
def fillconvex(ax, ay, r, g, b, a)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then setcolor(i, j, r, g, b, a) end
    end
  end
end
```

isinside は、与えられた点が「いずれかの辺の右側にある」なら図形の外にある、そうでなければ内側にあるか線上にある、と判断する。右側にあるかどうかは、辺の線分のベクトルと、線分の起点から調べるべき点までのベクトルの外積を計算して、負なら右側と判定する (このために左回りで周囲を指定するという条件が必要になっていた):

```
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i], ay[i+1]-ay[i], x-ax[i], y-ay[i]) < 0
      return false
    end
  end
  return true
end


def oprod(a, b, c, d)
  return a*d - b*c;
end
```

線を描く fillline だが、2 点の XY 座標と「線の幅」を指定する。線分のベクトルからそれと直交するベクトルを計算し、その長さが線の幅の半分になるようにする。あとは線分の両端点と幅ベクトルを加減することで細長い長方形ができるので、それを fillconvex で塗ればよい:

```
def fillline(x0, y0, x1, y1, w, r, g, b, a)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
             [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b, a)
end
```

では最後に、図 1 の絵を描くメソッドを示そう。ここまでに用意したものを順次呼ぶだけだから簡単である:

```
def mypicture
  initimage
  fillcircle(150, 30, 60, 255, 100, 70, 0.0)
  fillcircle(190, 100, 30, 100, 200, 80, 0.5)
  fillrect(60, 100, 120, 80, 80, 220, 255, 0.6)
  fillellipse(200, 60, 70, 40, 100, 100, 220, 0.7)
```

```
  filltriangle(200, 100, 300, 100, 250, 200, 200, 100, 250, 0.5)
  fillline(40, 40, 260, 160, 4, 0, 0, 0, 0.0)
  writeimage("t.ppm")
end
```

なかなか大変だったけど、このように手続きを次々に作っていくことで大きなプログラムでも組み立てて行けることが納得できますね?

2 さまざまな整列アルゴリズム

2.1 整列アルゴリズムの計測

以下では前々回の演習で解説を積み残した「配列の整列」について扱い、また前回は言及しなかったアルゴリズムについてもいくつか説明する。これらについて、所要時間計測を行いたいので、初回に時間計測用に示したメソッド bench を再掲し (ただし puts で直接出力するよう手直し)、また指定した要素数の整数 (0〜9999) の配列を生成するメソッド randarray も示す。

```
def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  puts t2-t1
end

def randarray(n)
  a = []
  n.times do a.push(rand(10000)) end
  return a
end
```

なお、rand(10000) は 0〜9999 の間の整数をランダムに返してくれる。後で実験の内容によってはもっと大きい値にしてみてもよい。

これらを使って整列プログラム時間計測をする場合、次のようにすることになるだろう:

```
irb> a = randarray(1000)  ←データ数 1000 個
...                       ←そのデータが表示される
irb> bench(1) do bubblesort(a) end  ←配列を整列
=> 1.6015625              ←所要時間 (秒) が表示される
```

2.2 演習問題解説: 単純選択法による整列

単純選択法については、考え方は前々回資料に載っていたが、簡単におさらいしよう。

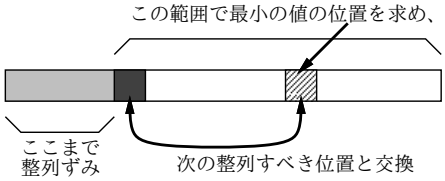

図 2: 単純選択法による整列

まず 0〜N 番目の最小を取り出し、それを 0 番目に置く (元の 0 番目は最小を取り出したところに置く。つまり 2 箇所の値を交換する)。次に残った 1〜N 番のうちの最小を取り出し、それを 1 番目に置く (やはり最小を取り出したところと交換)。これを繰り返して行くことで、小さい順に並ぶのは明らかでさある。ではアルゴリズムを示そう:

- selectionsort(a) — 配列 a を単純挿入法で整列
- n ← a.length
- i を 0 から n-2 まで変えながら繰り返し、
- k ← {a 中の i 番〜n-1 番の間の最小値の添字 }
- a_i と a_k の内容を交換
- 繰り返しおわり。

つまり、左から順に「残っているものの最小」をみつけてそれを「(交換を使って) 次の場所に置く」ことで左から小〜大の順に並べる (図 2)。{…} の間は「これからさらに詳しく書く」ことを意味するものとしよう。で、詳しく書く方法として、その部分にさらに詳しい内容を埋め込むこともできるが、手続きとして分けて書くと疑似コードでもプログラムでも分かりやすい。

- selectmin(a, i, j) — 配列 a の i〜j 番目のうち最小要素の番号
- min ← a_i。k ← i。
- l を i+1 から j まで変えながら繰り返し、
- もし $a_l < min$ なら、min ← a_j。k ← l。
- 繰り返し終わり。
- k を返す。

これに対応する Ruby のコードは次の通り:

```
def selectmin(a, i, j)
  min = a[i]; k = i
  (i+1).step(j) do |l|
    if a[l] < min then min = a[l]; k = l end
  end
  return k
end

def selectionsort(a)
  n = a.length
  (n-1).times do |i|
    k = selectmin(a, i, n-1); a[i],a[k] = a[k],a[i]
  end
end
```

Ruby では変数 (や配列要素) の値を入れ換えるのは非常に楽に書けるようになっていて「`a,b = b,a`」のように書けばよい。他の多くの言語では「`z = a; a = b; b = z`」のように作業変数を使って入れ換える必要がある。

2.3 演習問題解説: 比較交換法 (バブルソート)

前々回資料で 2 番目の方法として例示されていた比較交換法 (バブルソート) については、直接コードを示そう:

```
def bubblesort(a)
  done = false
  while !done do
    done = true
    (a.length-1).times do |j|
      if a[j] > a[j+1] then a[j],a[j+1] = a[j+1],a[j]; done = false end
    end
```

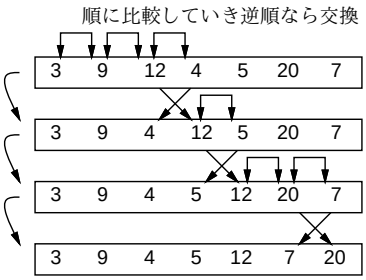

図 3: バブルソートによる整列

```
  end
end
```

この方法では、外側ループは done という旗を用いて、この旗が立っていない間繰り返しを続ける。その中では旗を立てる…つまり、「全部正しく並んでいる」ものととりあえず思う。次に、隣接する各要素について順次調べ、大小が逆のところで交換を行い、また (完全じゃなかったので) 旗を降ろす。このため、一巡するともっとも大きい要素が右端に来ている。再度一巡すると、2 番目に大きい要素が右端から 2 番目に来る。これを繰り返していくと、最後は全部の要素が正しく並んで、旗が降りなくなるので、外側ループを抜ける。疑似コードは省略でいいですよね。

演習 1 計測用メソッド bench、乱数による配列生成メソッド randarray と上に出て来た単純選択法による整列のメソッド selectionsort またはバブルソートによる整列のメソッド bubblesort または自作の整列メソッドを打ち込み、配列サイズ 1000, 2000, 3000, ... で所要時間がどうなるか計測し、考察しなさい。

2.4 マージソート

では、整列アルゴリズムでもっと速いものはないのだろうか。次の方法 (**マージソート**) を見てみよう。これは呼び出し方として次のように、「配列のどこからどこまでを整列する」かを指定する:

```
mergesort(a, 0, a.length-1);
```

その疑似コードを示そう。

- mergesort(a, i, j) — 配列 a の i 番から j 番の範囲を整列
- 　もし j = i+1 なら、
- 　　もし $a_i > a_j$ なら、a_i と a_j を交換。
- 　そうでなくてもし j > i+1 なら、
- 　　k ← (i + j) / 2。
- 　　mergesort(a, i, k)。mergesort(a, k+1, j)。
- 　　{2 つの整列された列をマージ (併合) する }
- 　枝分かれおわり。

考え方としては、まず再帰呼び出しによって列全体を半分ずつにして行き、長さ 1 や 2 のときはすぐ整列できるので、その後戻って来たら 2 つの整列ずみの列をマージすることで長い整列ずみの列にする (図 4)。

なお、**マージ** (併合) とは、$(1, 3, 4, 8)$ と $(2, 5, 6, 7)$ のように 2 つの整列ずみの列があったときに、それを併せて $(1, 2, 3, 4, 5, 6, 7, 8)$ のように 1 つの整列ずみの列にすることを言う。疑似コードにすると長いのでそれは略。では実際にマージソートのコードを示す:

```
def mergesort(a, i, j)
  if j == i+1
    if a[i] > a[j] then a[i],a[j] = a[j],a[i] end
  elsif j > i+1
    k = (i + j) / 2
```

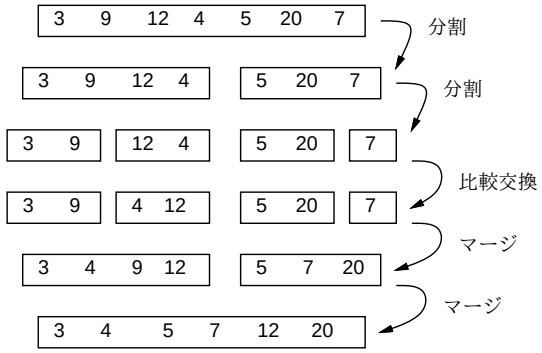

図 4: マージソートによる整列

```
    mergesort(a, i, k); mergesort(a, k+1, j)
    i1 = i; k1 = k+1;  b = []
    while i1 <= k || k1 <= j do
      if i1 > k || k1 <= j && a[i1] > a[k1]
        b.push(a[k1]); k1 = k1 + 1
      else
        b.push(a[i1]); i1 = i1 + 1
      end
    end
    b.length.times do |l| a[i+l] = b[l] end
  end
end
```

併合のところがちょっと面倒だが、要するに併合に使う配列 b を用意し、そこに 2 つの整列ずみ部分列からコピーしながら併合し、終わったら元の配列 a にコピーし戻すようになっている。

2.5 クイックソート

もう 1 つ別のアルゴリズムを示そう。これは**クイックソート**といういかにも速そうな名前がついている。

```
def quicksort(a, i, j)
  if j <= i then return end
  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
```

非常に短いけど、説明されないと分からないですよね。まず長さ 0、1 なら何もしないのはマージソートと同じ。次に、マージソートと同じく列を 2 つに分けるが、こちらはある値 p(ピボットと呼ぶ) を選んで「左半分は p 以下、続いて p の値、右半分は p より大きい」状態にしてから、左半分と右半分をそれぞれ整列する。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になるのでこれで完了なわけだ (図 5)。

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストだが、そんなものは分からないのでランダムに選ぶことにし、上のコードでは右端 (j 番目) の値を p にしている。変数 s は「この番号の 1 つ手前までは p 以下のものを詰めてあるので、次に p 以下のものが見つかったらこの位置に入れる」番号を表している。そこで、k を i から j-1 まで

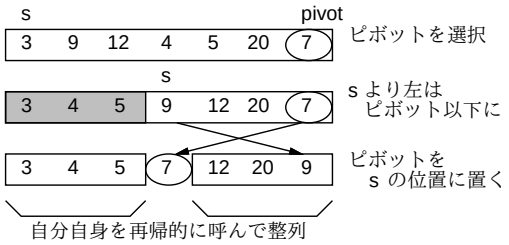

図 5: クイックソートによる整列

(j 番はピボットが入っているので保留) 左から順に調べて、a[k] が p 以下ならそれを s 番目の要素と交換して s を増やすことで、左半分と右半分に分けることがおこなえる。

分け終わったら、最後に j 番目と s 番目を交換することで、保留してあったピボットの値をあるべき位置に置く。その後、自分自身を呼ぶわけだが、s 番目のピボットの位置はこれで合っているので、i〜s-1 と s+1〜j の範囲について自分自身を呼べばよい。

演習 2 マージソートとクイックソートの好きな方 (両方でもよい) を打ち込んで動かし、所要時間を計測してみよ。配列サイズを変化させたときの挙動は単純選択法やバブルソートと比べてどうか考察せよ。

2.6 時間計算量

一般に、「アルゴリズム (やプログラム) がどれくらい時間がかかるかを表す量」のことを**時間計算量**と呼ぶ。

selectionsort を例題にして、これがどれくらい「速い」(遅い?) かを見積もってみよう。データの数を N とすると、selectionsort() の中からは selectmin を N 回呼び出し、値の交換も N 回行う。ループの処理も (当然)N 回。なので、ここで消費される時間はたとえば $T_1 = C_1 N$ と表すことができる。

次に、selectmin の処理を見てみよう。これは、1 回目には N 個の値を調べ、2 回目は $N-1$ 個の値を調べ、3 回目は $N-2$ 個の値を調べ、…のようになるので、1 回調べるのについて時間が C_2 掛かるものとすると、合計では次のようになる。

$$T_2 = C_2 N + C_2(N-1) + \cdots + C_2 1 = \frac{C_2}{2} N(N+1)$$

これらを合計すると

$$T = T_1 + T_2 = \frac{C_2}{2} N^2 + (C_1 + \frac{C_2}{2}) N$$

ここで、仮に C_1 が C_2 の 100 倍くらいあったとしよう (そんなに差があるとはとても思えないが)。だとしても、N として 1000 とか 10000 とかを入れて計算するわけだから、第 2 項 (N の 1 次の項) はほとんど無視できる。

このように、アルゴリズム/プログラムの実行時間を評価するときには、「入力 N や要求する誤差 δ に対してどれくらい時間が掛かるか」を見積もるが、その時もっとも次数の高い項が支配的になるので、低い次数の項は通常無視できる。

さらに、2 つのアルゴリズムがあって、片方が $T = C_1 N^2$、他方が $T = C_2 N^3$ だったとすると、たとえ C_1 が C_2 の 1000 倍だったとしても (そんなこともあまりありそうにないが)、N に 10000 を入れるとそんな差はすぐに逆転してしまう。このため、定数も無視して、N の次数 (オーダー) だけを問題にして「このアルゴリズムの時間計算量は $O(N^3)$ である」などと言う。

N 個のデータを入力するようなプログラムでは、そのデータの読み込みに $O(N)$ は最低必要である。これを線形時間と呼ぶ。たとえば最大や最小を求めるなど、線形時間のアルゴリズムがあるような問題はコンピュータで簡単に処理できると思ってよい。なお、N 個の値といっても、それを内部的に計算するだけなら、計算を工夫して $O(N)$ より小さいアルゴリズムを構成できる場合がある。

少し込み入ったアルゴリズムは、$O(N^2)$、$O(N^3)$ などの計算量になる。これを**多項式時間**と呼ぶ。さらに時間計算量の大きなものとしては、$O(e^N)$ や $O(N!)$ などもあり、これらだとコンピュータで実用的に扱えるのは小さい N に限られてしまう。

では次に、バブルソートの時間計算量はどうだろうか。内側のループでは $N-1$ 回の比較を行う。そして、**最善の場合**つまり最初から全部並んでいる場合は、1 回内側のループを実行したらそれで完成である。つまり $O(N)$。しかし**最悪の場合**、つまり完全に逆順に並んでいる場合は、内側の 1 回目のループは最も大きい要素を最後の位置に持って来るだ

けで終わってしまう。2 回目は 2 番目に大きい要素を最後から 2 番目の位置に…というわけで、内側のループが N 回必要になる。つまり $O(N^2)$。では平均的にはどうだろうか。平均的には、内側のループの繰り返しは N 回は必要ないだろうが、N に比例する回数 (たとえば $0.1N$ とか) が必要になりそうである。ということは、平均でも (定数倍は無視するので) やはり $O(N^2)$ になるわけである。

ここで単純挿入法とバブルソートの所用時間 (ミリ秒) を手元のマシンで実測してみた (表 1)。これを見ると、いずれも絶対値 (定数倍) の違いはあるが、N が 2 倍、3 倍になったとき所用時間が 4 倍、9 倍になっているので、確かに $O(N^2)$ の時間計算量らしいと言える。

表 1: 単純挿入法とバブルソートの所要時間

アルゴリズム	1,000	2,000	3,000
単純挿入法	0.438	1.711	3.852
バブルソート	1.602	6.344	14.500

ではマージソートの計算量はどうだろうか。1 つの `mergesort` の呼び出しを見ると、単純な場合 (長さが 0、1、2) は一定時間で済む (当然)。長さ N の場合は、それを前半と後半に分けて、それぞれ自分自身を再帰的に呼び出して整列し、最後に併合する。自分自身に掛かる時間は分けて考えるとして、併合は両方の列の先頭を見て小さい方を取ることを繰り返せばいいので、$O(N)$ で済む。さて、再帰呼び出しの方はどうだろうか。長さ N の列を半分にしてそれぞれ`mergesort` を呼ぶのだから、2 段目の呼び出しは $O(N/2) + O(N/2) = O(N)$。3 段目は 4 分の 1 の列について 4 つ呼ぶのでやはり $O(N)$。合計何段あるかというと、「N を何回半分にしたら 1 になるか」だから $log_2 N$。なので、全体では $O(NlogN)$ の計算量となる (計算量では log の底が何かも省略するのが通例)。

これを実行してみると、N が 1000、2000、3000 で 0.031、0.070、0.125 と、比べものにならないくらい速い。つまり、定数倍の違いで頑張るよりも、時間計算量のオーダーが優れたアルゴリズムを見つけるほうが、はるかに効果的だというわけだ。

では、クイックソートの計算量はどうだろうか。1 回ぶんの処理はやはり $O(N)$ で、再帰の段数はピボットの選択が完璧なら $log_2 N$ 回だが、ランダムに選んでいるのでその定数倍と考える。定数倍は無視するので、これも計算量は $O(NlogN)$ になる。

ただし、極めて運が悪い場合、つまりピボットの選択が悪くて毎回列の最大か最小の値をピボットにしてしまうと、段数が N になってしまうので、最悪の計算量は $O(N^2)$ ということになる。そんな運が悪いことはないだろうと思うかも知れないが、既に整列ずみの値を渡されるとまさにそうなってしまう。

演習 3 クイックソートに既に並んでいる配列を与えると計算量が $O(NlogN)$ から $O(N^2)$ になってしまうことを計測により確認しなさい。また、できたら、この弱点を解消する工夫を考えて実現してみなさい (これは必須ではないです)。

2.7 ビンソート

ここまでの方法とは考え方がまったく違う整列アルゴリズムである、**ビンソート**を紹介しよう。このアルゴリズムは、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。たとえば、整列する値の範囲 0〜3 の整数だけだったとする (もちろん、そのデータの個数は 1 万でも 2 万でもあるかも知れない)。

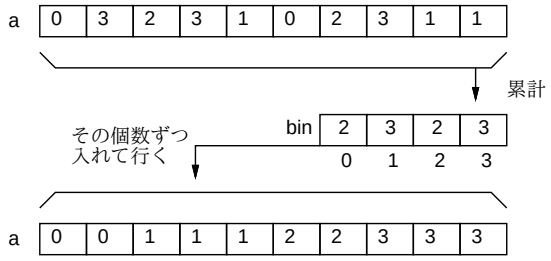

図 6: ビンソートによる整列

それなら図 6 のように、まず 0、1、2、3 それぞれの値について「何回現れるか」を数えてしまう。そして数え終わったらこんどは「0 が 2 回、1 が 3 回、…」のように数えた個数ずつその値を繰り返せば、確かに元のデータを並べ変えたのと同じことになる。0〜3 じゃあんまりな、と思うでしょうが、実際にはコンピュータのメモリは沢山あるので、0〜9999 とかでも全く問題ないし、それなら使い道は結構ありそうでしょう? (先に掲げた `randarray` が生成するデータもこの範囲の整数であることに注意。)

演習 4 ビンソートのプログラムを作成し、所要時間を計測しなさい。もちろん、ビンソートの時間計算量についても検討すること。

2.8 基数ソート

ビンソートの弱点は、現れる値の範囲があまりに広いと (100 万とか 1000 万とか) 巨大な配列を必要とし、効率も悪くなることである。そこで、やはり値が整数である必要があるが、ビンソートよりも値の範囲に対する許容度が高い整列アルゴリズムである**基数ソート**を紹介しよう。ここでは簡単のため、負の値はないものとして説明する。

基数ソートでは、整列する値を 2 進で表した時に「下から i ビット目が 1 であるか否か」を調べる必要がある。これを Ruby でどう書くかを説明しておこう。

Ruby では「<<」という演算子は「左シフト」つまりビット列である整数値を 1 ビットぶん左にずらす働きがある。だから `1 << 2` は $100_{(2)}$ だから 4 だし、一般に `1 << i` で `i` 番目のビットだけが 1 になった数値を得ることができる。

次に、「`&`」という演算子は「ビット単位の and 演算」つまり 2 つの数の 2 進表現で「両方とも 1」の位置だけが 1、それ意外は 0 であるような 2 進表現に対応する数が得られる (条件の「かつ」は「`&&`」だったがアンド記号が 1 個の場合はまったく別の意味になるわけだ)。たとえば次の例を見てみよう:

```
  110100 --- 52
&)011101 --- 29
--------
  010100 --- 20
```

だから `52 & 29` の結果は 20 ということになる。

ではようやく、基数ソートの説明をすることができる。たとえば、変数 `mask` に上で行ったような 1 ビットだけが「1」になっている値を入れ、その 1 の位置を一番右 (下位) から順に左に移していく。そして、その `mask` との`&`の結果が 1 か 0 かで、データを右半分と左半分に分ける (図 7)。そうするとあらふしぎ、一番上のビット (ここでは 4 ビットとした) までやったときには、すべての数は小さい順に並んでいる。

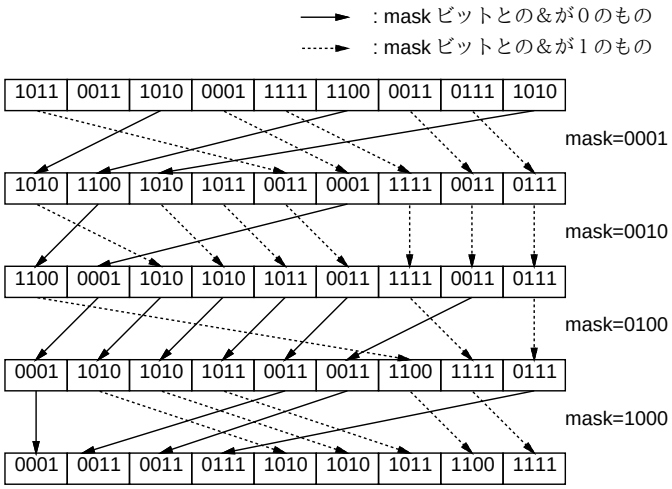

図 7: 基数ソートによる整列

これはなぜかというと、1 回目では一番下のビットが 1 のものが左、0 のものが右になるように振り分け、それについて 2 回目に下から 2 ビット目が 1 のものが左、0 のものが右になるように振り分けるわけだが、2 回目の振り分けをして

も 1 回目の振り分けの順序は崩れないので、2 ビット目が 1 のものの中、0 のものの中ではそれぞれ、まず 1 ビット目が 1 のもの、続いて 0 のものという順序が維持されている。3 ビット目、4 ビット目でも同様にそれより下のビットについては順序が維持されているので、結局最後まで来たときには順番が完全に並んだ状態となるわけだ。

演習 5 基数ソートのプログラムを作成し、所要時間を計測しなさい。もちろん、基数ソートの時間計算量についても検討すること。負の数も扱えるようになるとなおよい。

3 既出アルゴリズムの別バージョン

3.1 最大公約数

前に出てきた最大公約数のプログラムは引き算を使っていたが、替わりに剰余演算を使えば演算回数はずっと少なくなる (ユークリッドの互除法)。逆に、もっとベタなアルゴリズムとして、次のようなものも考えられる。

- i を $min(x, y) - 1$ から 1 まで 1 ずつ減らしながら繰り返し、
- 　　x も y も i で割り切れるなら、ループから抜け出す。
- 繰り返しおわり。
- i を打ち出す。

3.2 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すごく遅い。別の方法として、たとえば `x0` と `x1` に 1 を入れておき、それからループで `x0` にはこれまでの `x1`、`x1` にはこれまでの `x0+x1` を入れることを繰り返して計算することが考えられる (図 8)。

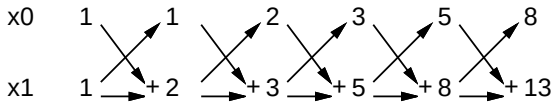

図 8: ループによるフィボナッチ数

もう 1 つこういうのはどうだろうか。

$$\begin{pmatrix} x_{i+1} \\ x_i \end{pmatrix} = \begin{pmatrix} x_{i-1} + x_i \\ x_i \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_i \\ x_{i-1} \end{pmatrix}$$

だから、$x_0 = x_1 = 1$ とおいてあとは上の漸化式で x_i を計算すれば i 番目のフィボナッチ数が求まる。漸化式といっても次々に同じ行列を掛けるだけだから、次の Q、v について $Q^n v$ を求めればよい:

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},\ v = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

そして、Q^n を求める時に次の漸化式を活用すると、馬鹿正直に n 回行列の掛け算をやるよりも速く結果を求められる。(注記: これは漸化式なのだから、Q^{n-1} とか $Q^{\frac{n}{2}}$ のところも「同じ方法で」計算する必要がある。でないとちっとも速くならないので注意。)

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

3.3 組み合わせの数

組合せの数も再帰的定義そのままでは非常に遅い。別の方法として、前回やった掛け算を使う方法がまずある。また、「パスカルの三角形」を作る方法がある:

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

演習 6 何か 1 つ、ある計算に複数のアルゴリズムが存在するようなものを選び、その複数 (3 つ以上あればよい) のアルゴリズムの時間計算量をそれぞれ見積もってみよ。また、実際にそのようになっているかどうか、プログラムを動かして計測し確かめてみよ。予想と合わない場合はその理由も検討すること。

これまでに出てきた題材としては「素数の判定」「素数の列挙」「最大公約数」「組合せの数」「フィボナッチ数」「整列」「平方根」などがあったが、これ以外に自分で考えてもよい。短時間で終わってしまう計算の場合は、その計算を何回も繰り返して実行させて時間を測り、割り算で 1 回あたりの時間を求めればよい。

A 本日の課題 5A

「演習 1」または「演習 2」の計測結果を、計測したプログラムと併せて、今日中に久野までメールで送ってください。

1. Subject: は「Report 5A」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. 計測結果のまとめ (とできれば簡単な考察)
5. 以下のアンケートの回答。

   Q1. 整列アルゴリズムを少なくとも 1 つは理解しましたか。

   Q2. 時間計算量という考え方についてどう思いましたか。

   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 5B

次回までの課題は「演習 3」〜「演習 6」から 2 つ以上選んで報告することです。「演習 6」については、ごく簡単なものでもよいので、頑張りすぎないようにしてください。たとえば掛け算をするのに「*」演算を使うのと繰り返し足し算をするのを比較するとかいかがでしょうか。

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 5B」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. 1 つの計算に複数のアルゴリズムがあることが納得できましたか。

   Q2. 自分で作れる程度のプログラムについてなら、その時間計算量が求められそうですか?

   Q3. 課題に対する感想と今後の要望をお書きください。