# 基礎プログラミング+演習 ＃8 – 計算量 (2)+乱数とランダム性

久野 靖 (電気通信大学)

2017.12.11

時間計算量は難しいところだったので、今回はとりあえず演習問題解説を復習を兼ねてやった後、改めて時間計算量を求める演習を用意しました。その後で以下内容が本題となります。

- 既出アルゴリズムの別解法と計算量
- 乱数とランダムアルゴリズム

## 1 前回演習問題の解説

### 1.1 演習 3(1) — マージソートの時間計算量

マージソートについて再度見直してみましょう。コードを示します。

```
def mergesort(a, i, j)
  if j <= i
    # do nothing
  else
    k = (i + j) / 2
    mergesort(a, i, k); mergesort(a, k+1, j)
    b = merge(a, i, k, a, k+1, j)
    b.length.times do |l| a[i+l] = b[l] end
  end
end
```

マージというのは、2 つの整列済みの列を「合併」させて 1 本の整列ずみの列にする、ということでしたね。では、2 つの整列済みの列はどうやって作ればいいでしょうか? その答えは「自分の担当範囲を前半と後半に分けて自分自身を (再帰的に) 呼び出す」というものでした。再帰がうまく働くためには、自分自身に元より簡単な問題を渡す必要がありますが、今回の場合は「自分の担当範囲の半分の長さの列を渡す」ことで簡単にしています。最後は長さが 1 になり、長さが 1 だったらそのままで整列済みということになりますから。マージも一応再掲しておきます。

```
def merge(a1, i1, j1, a2, i2, j2)
  b = []
  while i1 <= j1 || i2 <= j2 do
    if i1 > j1 || i2 <= j2 && a1[i1] > a2[i2]
      b.push(a2[i2]); i2 = i2 + 1
    else
      b.push(a1[i1]); i1 = i1 + 1
    end
  end
  return b
end
```

では、計算量の検討はどうでしょうか? 最初に「長さ $N$ の配列を 1 個」整列する形で呼び出すと、1 回目の再帰では「長さ $\frac{N}{2}$ の配列を 2 個」整列することになり、2 回目では「長さ $\frac{N}{4}$ の配列を 4 個」整列することになります。ということは、それぞれの段 (「回目」) では、自分自身の再帰呼び出しを除くと、合計で長さ $N$ のデータをマージし、元の配列に書き戻します。ですから、1 段について $O(N)$ の時間が掛かります。では再帰は何段起こるでしょうか? それは、長さを半分ずつにしていって 1 になったらおしまいですから、段数を $L$ とすると、$2^L \sim N$、つまり $L \sim \log_2 N$ ということになります。$O(N)$ の処理が $L$ 段あるのですから、全体としての時間計算量は $O(N \log N)$ ということになります。

## 1.2 演習 3(2) — クイックソート

では、再度クイックソートの説明をしてから、その時間計算量を考えることにします。いきなりコードを示します。

```
def quicksort(a, i, j)
  if j <= i
    # do nothing
  else
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

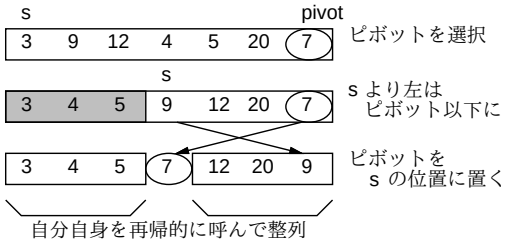


図 1: クイックソートによる整列

非常に短いですが、説明されないと分かりませんね。まず長さ 1 以下なら何もしないのはマージソートと同様です。次に、マージソートと同じく列を 2 つに分けますが、こちらは**ピボット** (pivot) と呼ぶある値 $p$ を選び、「左半分は $p$ 以下、続いて $p$ の値、右半分は $p$ より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。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」になり、整列が完了します (図 1)。

$p$ としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないのでランダムに選ぶこととし、上のコードでは右端 (j 番目) の値を $p$ にしています。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」番号を表しています。そこで、k を i から j-1 まで左から順に調べて、[1]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られます。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きま

[1] j 番はピボットが入っているので保留します。

す。その後、自分自身を再帰的に呼ぶわけですが、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びます。

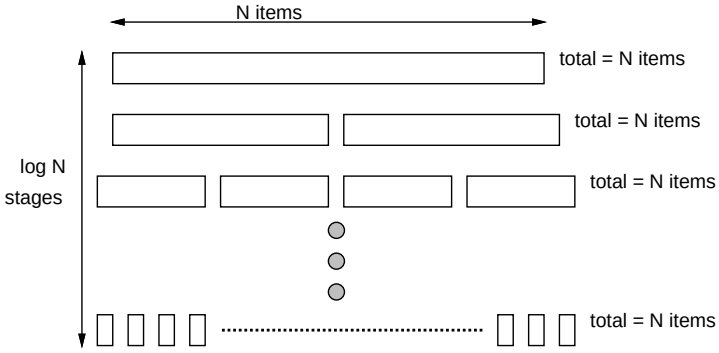


図 2: クイックソートのコード実行回数の検討

では、クイックソートの時間計算量はどうでしょうか。理想的な場合、つまり毎回列がおよそ半分ずつになるとすると、再帰呼び出しの深さは $\log_2 N$ になります (たとえば 16 なら 4 段、64 なら 6 段という感じ)。そして、各深さにおいて、その深さの呼び出しを全部合わせると、$N$ 個のデータ全部をピボットと比較して振り分けることになります。これを合計すると、$N$ 個のデータを振り分けることを $\log N$ 段繰り返しておこなうので、計算量は $O(N \log N)$ となります。

### 1.3 演習 5 — ビンソート

ビンソートのアルゴリズムについて、再度簡単に説明します。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ます。たとえば、整列する値の範囲が 0〜3 の整数だけだったとします (もちろん、そのデータの個数は百万とか一千万とかあるかも知れません)。それなら図 3 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」を数えてしまいます。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換えたのと同じことになるわけです。0〜3 ではあんまり役に立たないと思うでしょうが、実際にはコンピュータのメモリは沢山あるので、今回のように範囲が 0〜9999 とかくらいなら全く問題ありません。

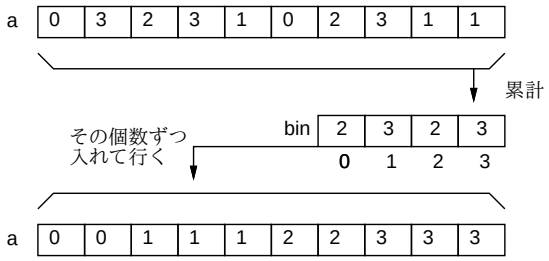


図 3: ビンソートによる整列

ではビンソートのコードを示します。前半のループで数を数え、後半のループで数えた数だけずつ値を並べて行きます。

```
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
```

### 1.4 演習 6 — 基数ソート

基数ソートを十進で説明し直します (図 4)。たとえば 3 桁の場合、最初は下から 1 桁目に着目して「0」〜「9」の箱に分類し、次にそのままの順で取り出した後、下から 2 桁目に着目して「0」〜「9」の箱に分類します。再度そのままの順で取り出した後、下から 3 桁目で分類すると全部並んでいます。

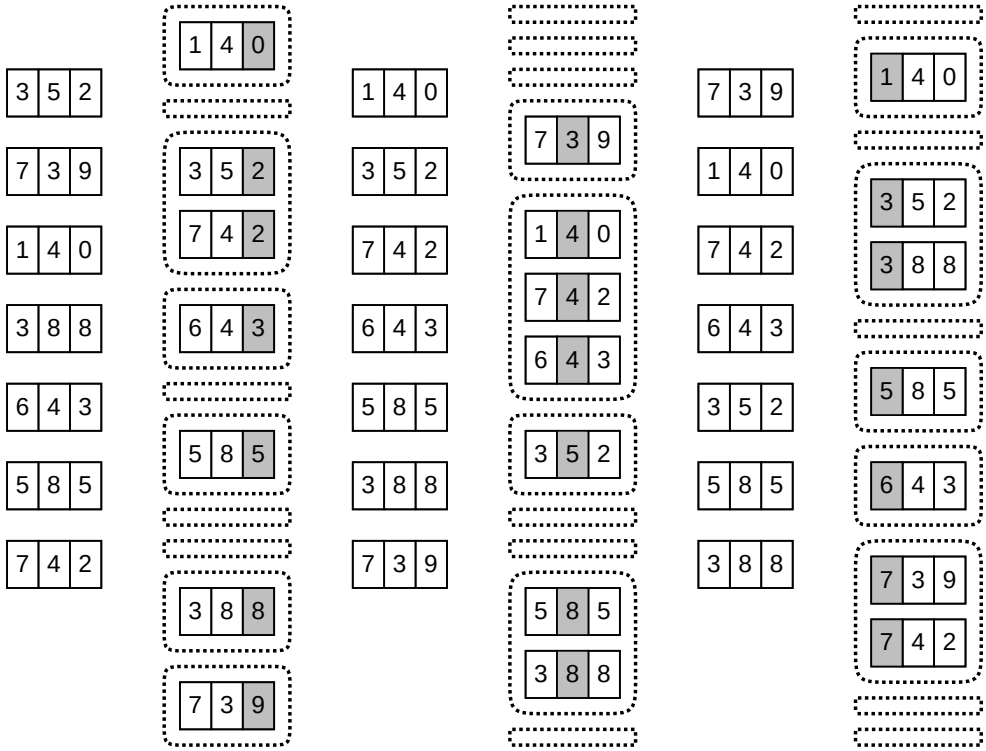


図 4: 十進法の場合の基数ソートの例

なぜこうなるかというと、ある桁について並べ終わった後はその順を崩さないで分類・移動を行っているので、下から $p$ 桁目まで来た時には「それ以降の桁については順番になっている」状態であり、以後もその順が崩れないから全部の桁が終わったら完全に整列できるわけです。ここでは十進で説明しましたが、2 進であれば「箱」は 2 つでよいわけです。

次に 2 進による基数ソートのコードを示します。整列する値のビット数を指定します (10000 までの値だと 14 ビットで済むので 14 を指定)。各周回ごとに当該ビットの値に応じてデータを配列 b と c に振り分け、終わったらこの順で a にコピーし戻しています。

```
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
```

### 1.5 演習 2〜5 — 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 $N$ を変えて手元のマシンで計測した結果を 1 に示します。

表 1: さまざまな整列アルゴリズムの時間計測

| データ数 | 1,000 | 2,000 | 3,000 | 5,000 | 10,000 | 20,000 | 30,000 | 50,000 |
|---|---|---|---|---|---|---|---|---|
| バブルソート | 1,219 | 4,945 | 11,117 | - | - | - | - | - |
| 単純選択法 | 305 | 1,242 | 2,766 | - | - | - | - | - |
| 単純挿入法 | 375 | 1,531 | 3,445 | - | - | - | - | - |
| マージソート | 23 | 62 | 94 | 164 | 351 | 758 | 1,172 | 2,055 |
| クイックソート | 15 | 31 | 55 | 109 | 219 | 508 | 789 | 1,492 |
| ビンソート | 8 | 8 | 11 | 14 | 20 | 34 | 47 | 74 |
| 基数ソート | 27 | 57 | 86 | 141 | 280 | 566 | 838 | 1,402 |
| $N+E$ | 11,000 | 12,000 | 13,000 | 15,000 | 20,000 | 30,000 | 40,000 | 60,000 |

$O(N^2)$ のアルゴリズムであるバブルソート、単純選択法、単純挿入法は、$N$ が大きくなると急激に遅くなって役に立たなくなります。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と言えます。

ビンソートは極めて速いことは分かりますね。では、このアルゴリズムの時間計算量はどうでしょう。まず、すべてのデータを順に走査するという点では $O(N)$ だと言えますが、それだけではありません。値の数を $E$ とすると、最後に大きさ $E$ 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ります。なので、時間計算量は $O(N+E)$ になるわけです。今回は $E$ が 10,000 なので、途中まではこちらの方が主に問題になります。表 1 の一番下に $N+E$ を示しましたが、所要時間がだいたいこれに比例していることが読み取れます。

そして、ビンソートは $E$ 個ぶんの配列を必要とすることも忘れてはいけません。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ますが、ビンソートはまさにその例です。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」のように言います。つまり、値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です。

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 $O(N)$ ですが、マージソートと基数ソートは「別の場所に移してから戻す」ので $O(2N)$ になって います。[2]

最後に基数ソートの時間計算量ですが、キーのビット数ぶんだけ振り分け処理をおこなうので、時間計算量は $O(N \log E)$ ということになります。これを $\log E$ が今回は定数 (14) と考えれば、時間計算量は**線形時間**ということになります。実際、表 1 をチェックすると所要時間が $N$ にほぼ比例していることが分かります。

ただし、時間そのものは半分くらいのところまで、クイックソートよりも劣っています。これはつまり、データが非常に多くなると、先に説明したようにオーダーの差がすべてを支配しますが、それ

[2]基数ソートでは b と c を a と同じサイズで取るので 3 倍と思うかもしれませんが、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」データを詰めてゆけばよいのです。

ほどでもない場合には定数項の差が無視できず、オーダーの大きいアルゴリズムでも処理時間が短くて済む場合がある、ということを意味しています。たとえば、データが数個しかないのであれば、クイックソートを使うよりも単純選択方を使うほうが適切なわけです。

### 1.6 演習 3 — クイックソートの弱点

先に掲げた 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すには、次のように 2 回ずつ整列してみればよいでしょう。

```
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、結果は次のようになりました。

| データ数 | 1,000 | 2,000 | 3,000 |
|---|---|---|---|
| 1 回目 | 16 | 39 | 63 |
| 2 回目 | 984 | 3960 | 8859 |

1 回目は $O(N)$ に近い (実際には $O(N \log N)$ のはず) けれど、2 回目はずっと遅く $O(N^2)$ に近いようです。これを改良するにはどうしたらいいでしょう? それには、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、代わりにランダムに取るようにすればよいでしょう。[3]

```
def quicksort(a, i, j)
  if j <= i then
    # do nothing
  else
    p = i + rand(j-i+1); swap(a, p, j) # ***
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

これの計測は上の表と異なり、1 回目も 2 回目もほぼ同じ時間で整列が終わります。

## 2 時間計算量ふたたび exam

ここまでで整列アルゴリズムを題材に時間計算量を考えて来ましたが、他のアルゴリズムについても考えましょう。時間計算量の求め方を簡単にまとめると、次のようになります。

[3] プログラムの修正は***の 1 行のみ。ここで i〜j の範囲の整数 p を 1 つランダムに選び、a[j] と a[p] を交換してからこれまでと同様に処理します。

入力の値 $n$ に対して、プログラム中の「最も多く実行される箇所」の実行回数を求め、$n$ の式で表し、$O(f(n))$ の形で記す。

極端な例ですが、$n$ が出て来なければ $O(1)$(定数計算量) つまり $n$ の値に関わらず一定時間で終わることを意味します。速い方から順に典型的なものを挙げておきます。

- $O(1)$ — 定数時間
- $O(\log n)$ — 対数
- $O(\sqrt{n})$ — 平方根
- $O(n)$ — 線形計算量、$n$ に比例
- $O(n \log n)$ — よい整列アルゴリズム
- $O(n^2)$、$O(n^3)$ — 一般に多項式計算量と呼ぶ
- $O(2^n)$、$O(n!)$ — 指数計算量

実際に動かす時の感覚としては、$O(n)$ までは「すごく速い」、$O(n \log n)$ は「まあまあ速い」、$O(n^2)$ は「遅い」、$O(2^n)$ は「ひどく遅くて小さい $n$ しか役立たない」という感じです。

**演習 1** 以下に示すメソッドにさまざまな $n$ を与えた際の時間計算量を見積もりなさい。また、実際に bench で掛かる時間を計測し確認しなさい。bench のソースコードは次に再掲する。

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

**注意!** bench の計測値はあまり時間が短いと誤差が大きいので、少なくとも **0.1** 秒よりは大きくなるように回数を増やして計測すること。たとえば下の **(a)** であれば「bench(1000000) do sqare1(1000) end」(回数として百万を指定した場合) などとする。実際に **1** 回あたりの所要時間は表示された時間を回数で割って求めること。

**a.** $n^2$ を計算するメソッドその **1**

```
def square1(n)
  return n*n
end
```

**b.** $n^2$ を計算するメソッドその **2**

```
def square2(n)
  result = 0
  n.times do result = result + n end
  return result
end
```

**c.** $n^2$ を計算するメソッドその **3**

```
def square3(n)
  result = 0
  n.times do n.times do result = result + 1 end end
  return result
end
```

**d.** $1.0000000001^n$ を計算するメソッドその **1**

```
def near1pow1(n)
  result = 1.0
  n.times do result = result * 1.0000000001 end
  return result
end
```

**e.** $1.0000000001^n$ を計算するメソッドその **2**

```
def near1pow2(n)
  if n == 0
    return 1.0
  elsif n == 1
    return 1.0000000001
  elsif n % 2 > 0
    return near1pow2(n-1) * 1.0000000001
  else
    return near1pow2(n/2)**2
  end
end
```

**f.** $1.0000000001^n$ を計算するメソッドその **3** [4]

```
def near1pow3(n)
  return Math.exp(n*Math.log(1.0000000001))
end
```

**g.** $1$〜$3$ の値が **n** 個並んだ全組み合わせを生成する (印刷は省略)

```
def nest3n(n) nest3(n, "") end
def nest3(n, s)
  if n <= 0 then
    # puts(s)
  else
    1.step(3) do |i| nest3(n-1, s + i.to_s) end
  end
end
```

**h.** $1$〜$n$ の値のすべての順列を生成する (印刷は省略)

```
def perm(n)
  a = Array.new(n) do |i| i+1 end
  perm1(a, [])
end
def perm1(a, b)
  if a.length == b.length
    # p(b)
  else
    a.each_index do |i|
      if a[i] != nil
```

[4] Math.log は $\ln x$、Math.exp は $e^x$ を計算するメソッドである。

```
          x = a[i]; a[i] = nil; b.push(x)
          perm1(a, b)
          a[i] = x; b.pop
        end
      end
    end
  end
```

# 3 既出アルゴリズムの別バージョン

## 3.1 最大公約数

以下では、これまでに出てきたアルゴリズムについて、計算量の違う別バージョンをお見せして、計測の題材にしていただきましょう。まず、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ましたが、代わりに剰余演算を使えば演算回数はずっと少なくなります (こちらが一般にユークリッドの互除法 (Euclid's algorithm) として知られているものです。もっとも、最初にユークリッドが考案したのは引き算を使う方だったそうですが)。

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ます。

- gcd3(x, y) — x と y の最大公約数を求める
- 　i を $min(x, y)$ 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、i を返す。
- 　繰り返し終わり。

## 3.2 フィボナッチ数

やってみればすぐ分かりますが、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です。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ます (図 5)。

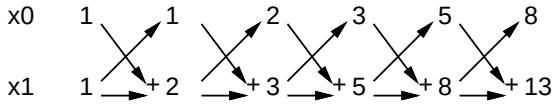


図 5: ループによるフィボナッチ数

もう 1 つ、こういうのはどうでしょうか。

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

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ります。漸化式といっても次々に同じ行列を掛けるだけですから、次の $Q$、$v$ について $Q^n v$ を求めればよいのです。

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

そして、$Q^n$ を求めるときに次の漸化式を活用すると、まじめに $n$ 回行列の掛け算をやるよりも速く結果を求められます。[5]

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

### 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です。別の方法として、以前にやった掛け算を使う方法がまずあります。また、**パスカルの三角形** (Pascal's triangle) を作る方法があります (図 6)。

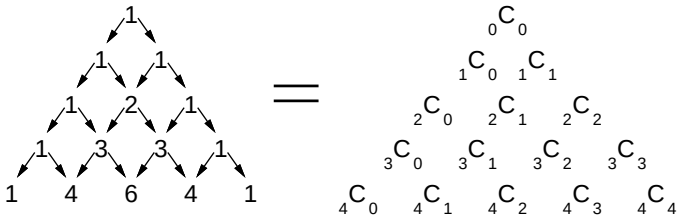


図 6: パスカルの三角形

**演習 2** ここに挙げたものまたはそれ以外のものについて、計算量の異なる複数 (少なくとも 2 つ) のアルゴリズムを用いたプログラムを作成し、それらの答えが一致することを確認した上で、実行時間を比較しなさい。

## 4 乱数とランダムアルゴリズム

### 4.1 乱数とは

**乱数** (random number) とは、簡単に言えば「ランダムな数」です。より正確に言うと、「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」のことを乱数列 (randam sequence) と言い、その中の個々の値が乱数です。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します。

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ものです (図 7)。たとえば、区間 $[0,1)$ の一様分布 (uniform distribution) 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します。これを**一様乱数** (uniform random number) と言います。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 以下の整数値ですが、どの数も同じ確率で出現するため、これも一様乱数です。この他によく使われる乱数としては、正規分布 (normal distribution) に従う**正規乱数** (normal random numbers) があります。[6]

### 4.2 擬似乱数 exam

**擬似乱数** (pseudorandom number) とは、プログラムで順次計算を行うことで生成される数値の列で、乱数列のように思えるものを言います。さんざん学んできたように、プログラムの動作は完全に決定的なものなので、「でたらめな」数を生成するのは思いのほか難しいものです。

擬似乱数のアルゴリズムの代表的なものとして、次のものがあります。

[5] 先の $1.0000000001^n$ の計算もこれと同じ方法のものがあります。

[6] 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** (standard score) などでおなじみです。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になったりします。

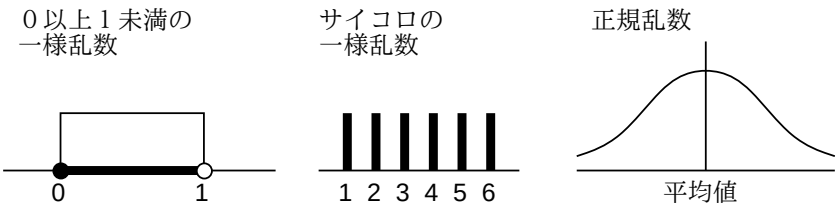


図 7: 乱数と分布

- **自乗採中法** (middle-square method) — $w$ 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。[7]
- **線形合同法** (linear congruential method)— $x_{i+1} = (x_i \times a + c)\ mod\ m$ により次々に値を生成していく。[8]
- **メルセンヌツイスター** (Mersenne twister)、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き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ります。これを**周期** (period) といい、もちろん周期が長いものが望まれます。MT は 32 ビットで $2^{19937}-1$ という長い周期を保証できるという性質を持つという点で画期的でした。さらに周期の他に統計的独立性の検定などもクリアしています。

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ます。[9]

このほか、最近では**オペレーティングシステム** (operating system, OS) が乱数機能を提供してくれている場合があります。OS の乱数機能は、ユーザのキーボード入力やディスクの動作などの外部割り込みに基づいた「ランダムさ」を活用するので、擬似乱数のような周期の問題がありませんが、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ります。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ります。

### 4.3　ランダムアルゴリズム exam

**ランダムアルゴリズム** (randomized algorithm) とは、(擬似) 乱数を活用して、ランダムな振舞いを持たせたアルゴリズムを言います。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) と呼ばれます。

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ます。たとえば、1 億要素の配列があるとして、「その半分の 5 千万要素には値 a が入っているが、それがどことどこか所は分からない」場合と「まったく値 a が入っていない」場合とがあり、そのどちらであるかを判断する必要があるものとしましょう。

決定的アルゴリズムでは、どのような調べ方をしてもその「裏をかかれる」可能性があって半分は調べなければ確実な判断ができません。たとえば先頭から順番に値 a があるかどうか見ていくとすると、値 a が全部後半に詰まっているかもしれないので、最悪で 5 千万要素を見る必要があります。では後ろから順に見ればいいかというと、値 a が全部前半に詰まっているかもしれないので同じことです。1 つおきでも何でも同様です。

[7]自乗採中法は古くからあるアルゴリズムですが、あまりよい乱数列は生成できないことが知られています。

[8]線形合同法は MT の発明以前は主流のアルゴリズムでした。ただし、よい擬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です。

[9]Ruby の `rand` の乱数アルゴリズムは最近の処理系 (バージョン 1.8 以降) では MT が使われています。

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 万回繰り返したとしましょう。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ありません。というのは、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からです。

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 algorithm) と言います。[10] 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 algorithm) と言います。[11]

たとえば、クイックソート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ります。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ものの、整列そのものはやはり正しく行えるからです。

### 4.4 モンテカルロ法 exam

**モンテカルロ法** (Monte Carlo method) 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います。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いくかを見ることで、さまざまな方法で交通信号を制御した場合の状況を簡単に試すことができ、それに基づいてよい制御方式を出すことができます。

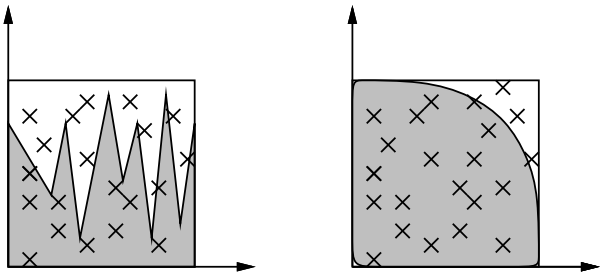

図 8: モンテカルロ法による数値積分

また、モンテカルロ法は数値積分にも使うことができます (図 8)。具体的には、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ます。積分とは「その関数の下側の面積を求める」ことですから、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) として使えます。

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よいのでは? 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) でないと使えません。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 つとなるのです。

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) $\pi$ の近似値を計算してみましょう。[12]

```
def pirandom(n)
  count = 0
  n.times do
    x = rand(); y = rand()
    if x**2 + y**2 < 1.0 then count = count + 1 end
```

[10] 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です。
[11] ラスベガスは米国にあるカジノで有名な都市の名前です。
[12] もちろん円周は十分連続かつ微分可能ですが、それは置いておいて。

```
    end
    return 4.0 * count / n
  end
```

実行させてみると次のとおり。

```
irb> pirandom 10000
=> 3.13
irb> pirandom 100000
=> 3.14444
irb> pirandom 1000000
=> 3.141604
```

有効数字 3〜4 桁では使えない、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場合は結構あります。たとえば、来年の GDP の成長率が 5.11 だろうと 5.12 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。

### 4.5 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きましょう。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りますが、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ます。これは、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からです。

簡単なゲームとして「数当て」を作ってみます。そのルールは次のとおり。

- プログラムは内部で 4 桁の数を「思い浮かべ」る (4 桁の中に重複はない。また 0 もない)。
- 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。
- プログラムは 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します。

```
def kazuate
  a = ""; b = "123456789"; count = 0
  4.times do i = rand(b.length); a += b[i..i]; b[i] = "" end
  while true do
    print("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 +=  1 else blow += 1 end
        end
      end
    end
    if hit == 4 then puts "you win!"; return end
    count += 1
    if count > 9 then puts "you lose! answer = #{a}."; return end
    puts "hit = #{hit}, blow = #{blow}."
  end
end
```

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 を使っています。また、puts は改行してしまうので、プロンプトを出力するのに print を使いました。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ます。

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しれませんが、「まず a に空文字列、b に"123456789"を入れ、b の中からランダムに 1 文字選んでそれを a に連結し、b のその文字は空文字列に置き換える」ことを 4 回やっています。

本体ループは、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るわけです。

**演習 5** 上の例題プログラムを打ち込んで遊んでみよ。納得したら、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。

### 本日の課題 8A

「演習 1」または「演習 2」から計算量を見積もる課題を 1 つ以上選び、動かしたプログラムを含むレポートを提出しなさい。見積りの簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. 計算量が見積もれるようになったと思いますか。

Q2. 乱数を使ったアルゴリズムの利点を納得しましたか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

### 次回までの課題 8B

「演習 1」〜「演習 8」の (小) 課題から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. あなたなりの「計算量の見積もり方」を簡単に説明してみてください。

Q2. 乱数を使ったアルゴリズムを自分なりにどのように考えますか。

Q3. 課題に対する感想と今後の要望をお書きください。