第4章　関数から計算へ

これまでの章では、画像の作成を除けば、単純な関数の組み合わによって複雑な計算を行う方法だけを見てきた。コンピュータの能力を引き出すためには、単純な関数を**何度も行って**答えを得る**反復**が必要になる。

この章では反復を行わせる 2 つの方法と、その応用を紹介する。第 4.1 節では**繰り返し**という、コンピュータに直接的に反復動作を行わせる方法を紹介する。第 4.2 節では**再帰**という、関数の定義に自分自身を使う方法によって反復を表わす方法を紹介する。第 4.3 節では配列や文字列を反復によってに扱う例をいくつか紹介する。

この章は繰り返しを先に再帰を後に紹介をしているが、第 4.1 節と第 4.2 節のどちらから先に読んでも構わないように書かれている。繰り返しによる定義はコンピュータ内部動作を考えやすい、再帰による定義はその計算が意味することを把握しやすいという特徴があるので両方を知っておくことが重要である。

4.1　繰り返しによる反復計算の定義

コンピュータの特徴の 1 つは、同じ計算を高速に反復して行う能力である。すでに第 3.4 節では、明度の計算を画像の各点に対して行うという簡単な反復処理の方法を見ている。

この反復処理は画像の作成だけでなく、例えば「1 から 100 までの数の和」や「100 番目の素数」のような値を求めるためにも使われる。

そのためには、反復して何かを計算しつつ、その結果を集めることが必要になる。まずは変数を使って結果を集めてゆく方法を見て、それを反復の中で使う方法を見てゆこう。

4.1.1　代入による変数の更新

第 1.3.2 節では、`weight=104.0` として BMI を計算した後、`weight=104.0*0.9` を実行して体重が 1 割減った場合の BMI を計算する例を見た。

このような代入で、右辺に同じ変数を使うこともできる。

```
1 irb(main):003:0> weight=104.0
2 => 104.0
```

```
3 irb(main):004:0> weight=weight-10
4 => 94.0
5 irb(main):005:0> weight
6 => 94.0
7 irb(main):006:0> weight=weight-10
8 => 84.0
```

3 行目の意味は、「現在の weight から 10 を引き、その結果を weight に代入する」というものである。別の言い方をすると、この命令は「weight の値を 10 減らす」ものだと言える。

当然、同じ代入命令をもう一度実行すると (7 行目)、そのときの weight の値から 10 引いた値に変更されるので、新しい weight の値はさらに小さなものになる。

4.1.2　繰り返しによる和の定義

まず、1 から n までの和を求める関数を考える。この計算は合計を表わす変数 s を用意して、「s を i だけ増やす」ことを $i \in 1 \ldots n$ について行えばよい。第 3.4.2 節で見たように for 命令を使うことで、次のように定義することができる。

```
1 # sum of the numbers from 1 to n
2 def sum_loop(n)
3     s = 0
4     for i in 1..n
5         s = s+i
6     end
7     s
8 end
```

ファイル 4.1: sum_loop

なおこの定義では、繰り返しによって書かれた関数を、再帰によって定義したものと区別するため、関数名に_loop を加えている。

(章末の練習問題: 4.6)

4.1.3　条件を満たす値を探す繰り返し

「1 から n までの数のうち、最大の k の約数 $\mathrm{gd}(k, n)$」を求める計算を繰り返しによって行うにはどうすればよいだろうか。再帰的関数を使った計算

(第 4.2.3 節) では、$n,\ n-1,\ n-2,\ \ldots$ の順に k の約数かどうかを調べ、最初に見つけた約数を答えとしていた。

これと同じ計算を繰り返しによって行うためには、新しい命令が必要になる。いままで使っていた for は「1 から n まで」のように回数を先に決めて繰り返す命令であるため、条件を満たす値が見つけるような場合は、何回繰り返せばよいかは分からないためである。

Ruby に用意されている、もう 1 つの繰り返しの命令は「ある条件が成り立っている間は計算を続ける」というものである。これを使って $\mathrm{gd}(k,n)$ を定義すると次のようになる。

```
1 # the greatest divisor of k in between 1 to n
2 load("./divisible.rb")
3
4 def gd_loop(k,n)
5     while !divisible(k,n)
6         n = n-1
7     end
8     n
9 end
```

ファイル 4.2: gd_loop

5 行目から 7 行目の while から end までが繰り返しの命令であり、while の後ろの式が繰り返しを続ける条件、次の行から end までが繰り返される命令である。従ってこの場合は「k が n で割り切れない間、n を 1 減らす計算を続ける」という意味である。例えば gd_loop(6,5) であれば

- k = 6, n = 5 のときに k が n で割り切れないので、
- n を 1 減らし、
- k = 6, n = 4 のときに k が n で割り切れないので、
- n を 1 減らし、
- k = 6, n = 3 のときに k が n で割り切れるので繰り返しを終える。

とい繰り返しになる。その後、8 行目でその時の—つまり k の約数になっていた—n の値を答えとするので、最初に見つけた約数を求めたことになる。

(章末の練習問題: 4.8)

4.2　再帰による反復計算の定義

コンピュータの特徴の 1 つは、同じ計算を高速に反復して行う能力である。すでに第 3.4 節では、明度の計算を画像の各点に対して行うという簡単な反復処理の方法を見ている。

この反復処理は画像の作成だけでなく、例えば「1 から 100 までの数の和」や「100 番目の素数」のような値を求めるためにも使われる。

このような計算を定義するための一つの方法は**再帰**である。以下では簡単な数学関数を例にとって、再帰的な関数の定義の方法を説明する。

4.2.1　再帰による和の定義

例として「1 から n までの和を求める関数 $\mathrm{sum}(n)$」を定義することを考えよう。式で書くと $\mathrm{sum}(n)$ は次のようになる。

$$\mathrm{sum}(n) = 1 + 2 + \cdots + n \tag{4.1}$$

この式は、そのままでは Ruby の関数にはならない。なぜなら、与えられた n によって式が変わってしまうからである。

ではどうすればよいだろうか。ここで $\mathrm{sum}(n-1)$ がどのような式かを考えてみよう。$n \geq 2$ の場合は次のようになる。

$$\mathrm{sum}(n-1) = 1 + 2 + \cdots + (n-1) \quad (\text{ただし } n \geq 2) \tag{4.2}$$

また、

$$\mathrm{sum}(n) = 1 \quad (n = 1 \text{ の場合}) \tag{4.3}$$

である。

式 4.1 と式 4.2 から、式 4.1 の右辺は $\mathrm{sum}(n-1) + n$ と表わせる (ただし $n \geq 2$ の場合) ことに気付く。$n = 1$ の場合とあわせると $\mathrm{sum}(n)$ には次のような関係があると言える。

$$\mathrm{sum}(n) = \begin{cases} \mathrm{sum}(n-1) + n & (n \geq 2) \\ 1 & (n = 1) \end{cases} \tag{4.4}$$

式 4.4 は、sum という関数を表わすために再び sum を使っている。この関係を左辺から右辺に展開してゆくことで和が求められる。実際に $\mathrm{sum}(3)$ のような式を展開することで確認してみよう。式の展開を $\mathrm{sum}(3) \Rightarrow \mathrm{sum}(2) + 3$ のように書くことにすると、

$$\mathrm{sum}(3) \Rightarrow \mathrm{sum}(2) + 3 \Rightarrow (\mathrm{sum}(1) + 2) + 3 \Rightarrow (1 + 2) + 3$$

となり、確かに 1 から 3 の和になっていることが分かる。

```
1 # sum of the numbers from 1 to n
2 def sum(n)
3     if n >= 2
4         sum(n-1) + n
5     else
6         1
7     end
8 end
```

ファイル 4.3: sum.rb

式 4.4 のように書かれた関係は、そのまま Ruby の関数として定義することができる。場合分けは条件分岐を使えばよいので、次のような定義になる。

これまでの関数定義と唯一違うのは、関数 sum の定義の中 (4 行目) に、関数 sum 自身を使った式が表われる点である。このような定義のことを、**再帰的関数**という。

実際に計算させてみると、確かに和を求めていることが分かる。

```
1 irb(main):004:0> sum(3)
2 => 6
3 irb(main):005:0> sum(10)
4 => 55
5 irb(main):006:0> 1+2+3+4+5+6+7+8+9+10
6 => 55
```

ここまでで見たことをまとめると次のように言える。

> **n 回の反復によって値を求めるような計算 $f(n)$ は、$f(n)$ と $f(n-1)$ の関係を見つければ、それを Ruby の再帰的関数として定義することができる。**

(章末の練習問題: 4.9 4.10)

4.2.2　約数の和

次に「1 から n までの数のうち、k の約数になっているものの和 $\mathrm{sod}(k,n)$」を定義することを考えてみよう。前に定義した sum との違いは下線部分、つまり k の約数だけを選んで和を求めるという点だけであるので、$\mathrm{sod}(k,n)$ と

関数名 sod は sum of divisors (約数の和) の略。

$\mathrm{sod}(k, n-1)$ の関係は次のように表わせる。

$$\mathrm{sod}(k,n) = \begin{cases} \mathrm{sod}(k,n-1)+n & (n \geq 2 \text{ かつ } n \text{ が } k \text{ の約数}) \\ \mathrm{sod}(k,n-1)+0 & (n \geq 2 \text{ かつ } n \text{ が } k \text{ の約数でない}) \\ 1 & (n = 1) \end{cases} \tag{4.5}$$

式 4.4 と見比べると、n が k の約数でない場合には 0 を足している点だけが違う。これを Ruby の関数として表わすと以下のようになる。

```
1  # sum of divisors of k in between 1 and n
2  load("./divisible.rb")
3
4  def sod(k,n)
5      if n >= 2
6          if divisible(k,n)
7              sod(k,n-1)+n
8          else
9              sod(k,n-1)
10         end
11     else
12         1
13     end
14 end
```

ファイル 4.4: `sod.rb` (ここでは 練習 3.7a で定義した `divisible(k,n)` を使って `n` が `k` の約数かどうかを判定している。)

```
1  irb(main):004:0> sod(10,9)
2  => 8
3  irb(main):005:0> 5+2+1
4  => 8
5  irb(main):006:0> sod(28,27)
6  => 28
7  irb(main):007:0> 14+7+4+2+1
8  => 28
```

練習 4.1 (素数の判定) 素数とは、1 と自分自身しか約数がないような数である。上で定義した関数 `sod` を使って 2 以上の整数 `n` が素数のときにのみ `true`, そうでないときに `false` となるような関数 `prime(n)` を定義せよ。(`n` が 1 の場合は考えなくてよい。つまり、`prime(1)` の答は `true` でも `false` でもよいとする。)

```
irb(main):007:0> prime(6)
=> false
irb(main):008:0> prime(7)
=> true
irb(main):009:0> prime(107)
=> true
irb(main):010:0> prime(961)
=> false
```

練習 4.2 (組み合わせ数) n 個から k 個を選ぶ組み合わせ数 ${}_nC_k$ を求める `combination(n,k)` を定義せよ。ただし ${}_nC_k$ は、次のような関係を満たしている。

$$ {}_nC_k = \begin{cases} 0 & (k > n \text{ のとき}) \\ 1 & (k = 0 \text{ のとき}) \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & (\text{それ以外}) \end{cases} \tag{4.6} $$

この関係は「n 個から k 個を選ぶ」方法は、n 個の中の最初の 1 個に注目すると「それを選び、残りの $n-1$ 個から $k-1$ 個を選ぶ」か「それを選ばず、残りの $n-1$ 個から k 個を選ぶ」に限られるからだと理解できる。

(章末の練習問題: 4.11)

4.2.3　条件を満たす値を探す

「1 から n までの数のうち、最大の k の約数」を求める関数 $\mathrm{gd}(k,n)$ を考えてみよう。前に定義した sod からの類推で、n が k の約数の場合と、そうない場合について考えてみる。もし n が k の約数だったとすると、明らかに $\mathrm{gd}(k,n) = n$ である。また n が k の約数でないとすると、最大の約数は $n-1$ 以下なので $\mathrm{gd}(k,n) = \mathrm{gd}(k,n-1)$ となるはずである。まとめると次のような関係になる。

関数名 gd は greatest divisor(最大の約数) の略。

$$ \mathrm{gd}(k,n) = \begin{cases} n & (n \text{ が } k \text{ の約数}) \\ \mathrm{gd}(k,n-1) & (n \text{ が } k \text{ の約数でない}) \end{cases} \tag{4.7} $$

(この関係には $n = 1$ の場合がないように見えるが、1 は常に k の約数になるので 1 つ目の場合に含まれている。)

これを Ruby の関数にするとファイル 4.5 のようになる。このようにして定義された `gd` の計算は n を 1 つずつ減らしていって最初に k の約数となった時に終わる。例えば `gd(6,5)` の場合であれば

$$ \mathtt{gd}(6,5) \overset{5 \text{ は } 6 \text{ の約数でない}}{\Longrightarrow} \mathtt{gd}(6,4) \overset{4 \text{ は } 6 \text{ の約数でない}}{\Longrightarrow} \mathtt{gd}(6,3) \overset{3 \text{ は } 6 \text{ の約数}}{\Longrightarrow} \mathtt{3} $$

のような順で展開され、答えの 3 を得る。

```
# the greatest divisor of k in between 1 to n
load("./divisible.rb")

def gd(k,n)
    if divisible(k,n)
        n
    else
        gd(k,n-1)
    end
end
```

ファイル 4.5: gd.rb

練習 4.3 (素数の判定の改良) 素数とは、自身を除く最大の約数が 1 であるような数だとも言える。関数 gd を使って素数の判定をする関数 prime2(n) を定義せよ。練習 4.1 で定義した prime と計算回数の違いがあるかを考えよ。

(章末の練習問題: 4.12)

4.3　配列・文字列と繰り返し

繰り返しを使った計算は、配列や文字列のように、似たような値が沢山並んでいるデータを扱う場合に威力を発揮する。すでに前章では画像を作成する例を見たが、ここでは違う種類の例として、組み合わせ数の計算と、文字列の探索を紹介する。

4.3.1　配列と繰り返しを使った組み合わせ数の計算

練習 4.2 に出てきた組み合わせ数 ${}_nC_k$ の計算は、そのまま繰り返しにすることは難しい。これは、組み合わせ数が単純に値を積み上げて決まるような性質がないからである。

このような場合でも、配列と繰り返しを使うと上手く計算ができることがある。まずは組み合わせ数 ${}_nC_k$ を表にしてみよう (表 4.1)。

この表の右上半分は $k > n$ なのですべて 0 であるが、組み合わせ数としては意味がないので空欄にしている。

この表は上の行から順に埋めてゆくことができる。各行について、まず左端 ($k = 0$) と対角線上 ($k = n$ となる所) はすべて 1 である。それ以外の部分は、${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k$ なので、真上と左上にある 2 つの数の和を求めてやればよい。

${}_nC_n = 1$ であることは式 4.6(p. 52) の定義から簡単に証明できる。

この表を、配列と繰り返しによって作ってみよう。以下は表を作り、そこから ${}_nC_k$ の値をとり出すような関数 combination_loop(n,k) の定義であ

n \ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

表 4.1: 組み合わせ数 ${}_nC_k$ の値

る。この関数はまず表をあらわす配列として大きさ (n+1)×(n+1) の配列を作る (4 行目)。そして配列の 0 行目から n 行目について、左端 (6 行目) と対角線上 (10 行目) を 1 にして、その間を内側の繰り返しによって埋めている。8 行目が真上と左上にある数を足した値をしまっているところである。このようにして表が完成したら、表の n 行 k 列目が求める答である (12 行目)。

```
1  load("./make2d.rb")
2
3  def combination_loop(n,k)
4    c=make2d(n+1,n+1)
5    for i in 0..n
6      c[i][0] = 1
7      for j in 1..(i-1)
8        c[i][j] = c[i-1][j-1] + c[i-1][j]
9      end
10     c[i][i] = 1
11   end
12   c[n][k]
13 end
```

ファイル 4.6: combination_loop

練習 4.4 (再帰と繰り返しの比較) 練習 4.2 で定義した combination とここで定義した combination_loop が同じ結果を返すことを、いくつかの n, k について確めよ。また、n, k を大きくした場合にはどちらが速いか、実際に計算させて比べてみよ。

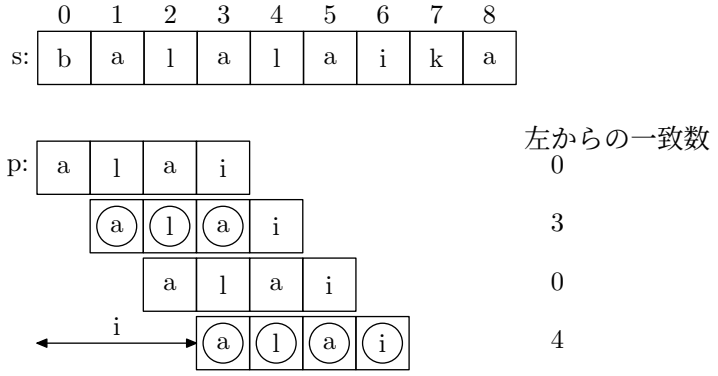
(章末の練習問題: 4.13 4.14)

4.3.2　言葉探し

文字列に関する処理には、長い文字列の中に特定の文字列が含まれている場所を見つけるというものがある。この処理は、ワードプロセッサの検索機能や、DNA の塩基配列の解析など多くの場面で使われる。

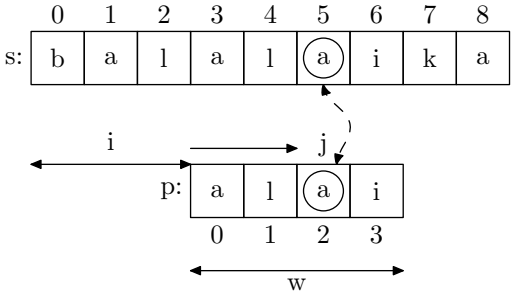
例として「balalaika」という文字列 (s とする) の中に「alai」という文字列 (p とする) が何文字目に現われるかを探す問題を考えよう。話を簡単にするために、s のどこかに必ず p が現われることは分かっているものとする。また 2ヶ所以上に現われる場合には、一番左側の場所を答えることにする。

下の図を使って探し方を説明する。



文字列 s と p を縦に並べて、左から何文字目まで一致しているかを調べる。図では丸の付いた文字が一致を表わし、右端にその個数を書いている。一致数が p の長さに満たない場合は p を 1 文字ずつ右にずらして繰り返す。そうすると一致数と p の長さが等しくなったときのずらした文字数が求める値になる。

このやり方では、s の i 文字目からと p が何文字一致しているかを調べる必要がある。それを関数 `submatch(s,i,p,w)` として定義してみよう。ただし、w は p の長さである。



関数 submatch は、上図のように j を $0, 1, 2, \ldots$ と変えながら s の i + j 文字目と p の j 文字目を比べてゆく。両者が一致しなくなったときは 0 から j − 1 文字目までは一致していたので一致数は j である。また、j が w になっ

たときは 0 から w − 1 文字目まですべて一致していたことになるので一致数はやはり j = w である。

従って j が w より小さくかつ、s の i + j 文字目と p の j 文字目が等しいあいだは繰り返しを続けて、どちらかの条件が成り立たなくなったときの j の値が一致数となる。Ruby の関数としては次のように定義できる。

```
1  def submatch(s,i,p,w)
2    j = 0
3    while j < w && s[(i+j)..(i+j)] == p[j..j]
4      j = j + 1
5    end
6    j
7  end
```

ファイル 4.7: match.rb (前半)

3 行目の後半の条件は、s の i+j 文字目と p の j 文字目が等しいかどうかを調べている。第 3.3 節で説明したように、Ruby では文字列 t の k 文字目は t[k..k] という式によって取り出すことに注意せよ。

例えば submatch("balalaika", 1, "alai", 4) の値は 3, submatch("balalaika", 3, "alai", 4) の値は 4 になる。

関数 match は、i を $0, 1, 2, \ldots$ と変えながら、submatch(s, i, p, w) が w に等しくなるような最小の i を求めればよいので、次のようになる。

```
8  def match(s,p)
9    i=0
10   w=p.length()
11   while submatch(s,i,p,w) < w
12     i = i + 1
13   end
14   i
15 end
```

ファイル 4.8: match.rb (後半)

なお、10 行目の p.length() は文字列の p の長さを求める式である (第 3.3 節参照)。

練習 4.5 (RNA 塩基配列の探索) 配布プログラム rna_sample.rb にある関数 seq0(), seq1() は RNA 塩基配列の例である。これらの中に AUG という文字列が表われる場所を探せ。

(章末の練習問題: 4.15)

4.4　定義のまとめ

条件を満たすまでの繰り返し: 式 が成り立つ間 命令$_1$ から 命令$_n$ を毎回実行する**繰り返し** `while` は次のように書く。

```
while 式
  命令1
  ⋮
  命令n
end
```

この命令は、まず (☆) 式 の値を求め、それが真であれば 命令$_1$ から 命令$_n$ までを順に実行し、再び☆からの手順を繰り返す。式 の値が偽になったら `end` 以降の命令に移る。

4.5　章末問題

練習 4.6 (繰り返しによって値を集める) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) `n` の階乗、即ち 1 から `n` までの積 `n!` を求める `factorial_loop(n)`.

b) $2^{\mathtt{n}}$ を求める `power2_loop(n)`. 記号`**`を使わずに定義せよ。

c) $\mathtt{x}^{\mathtt{n}}$ を求める `power_loop(x,n)`. 同じく記号`**`を使わずに定義せよ。

d) 次の式に示されるような級数を求める `taylor_e_loop(x,n)`.

$$\sum_{k=0}^{\mathtt{n}} \frac{\mathtt{x}^k}{k!}$$

($\mathtt{n} \to \infty$ のとき、この式は $e^{\mathtt{x}}$ の Taylor 級数になっている。)

練習 4.7 (繰り返しによって条件を満たす値を集める) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) 1 から `n` までの数のうち、`k` の約数になっているものの 個数(number of divisors) を求める `nod_loop(k,n)`. (ヒント: 式 4.5 の足す数を変える。)

b) 1 から n までの数のうち、素数になっているものの個数 (number of primes) を求める nop_loop(n).

c) 1 から n までの数について、それぞれ自身を含むような約数の和を求めたとき、最大のもの (maximum sum of divisors) を求める msod_loop(n).

練習 4.8 (繰り返しによって条件を満たす値を探す) 繰り返しによって以下のような値を求める Ruby の関数を定義せよ。

a) n より大きくかつ最小の素数—つまり次の素数 (next prime number) —を求める np_loop(n).

b) 素数 p よりも大きな素数のうち n 番目の素数を求める nth_prime_loop(p,n).

c) 第 3.2 節の式 3.1(p. 34) で紹介した関数 tnpo(n) は n が偶数なら 1/2, 奇数なら 3 倍して 1 加えた数を求めるものだった。数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ 1 になると予想した。例えば 3 から始めた場合は $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ といった具合に予想通りになっていることが確められる。

この予想は、反例は見つかっていないが証明もされていない、数学の未解決問題の 1 つである。

そこで n から上の手順で数を変化させて 1 になるまでの回数を collatz(n) とする。例えば collatz(5) = 5, collatz(16) = 4 である。

collatz(n) を求める Ruby の関数 collatz_loop(n) を定義せよ。

d) n 以上の整数で最小の完全数を見つける next_perfect_loop(n). ただし完全数 k とは k を除く約数の和が k と等しいような数のことである。例えば 6 の約数は (6 を除くと) 3, 2, 1 だけであり、その和が 6 になるので 6 は完全数である。最初に n が完全数であるかを判定する関数 perfect_loop(n) を定義しておくとよい。

練習 4.9 (再帰的に値を集める) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) n の階乗、即ち 1 から n までの積 $n!$ を求める factorial(n).

b) 2^n を求める power2(n). 記号**を使わずに定義せよ。(ヒント: 2^n は、$n \geq 1$ の場合に $2^n = 2 \times 2^{n-1}$ を満たしている。)

c) x^n を求める power(x,n). 同じく記号**を使わずに定義せよ。

d) 次の式に示されるような級数を求める taylor_e(x,n).

$$\sum_{k=0}^{n} \frac{x^k}{k!}$$

($n \to \infty$ のとき、この式は e^x の Taylor 級数になっている。)

練習 4.10 (文字列を作る再帰) ファイル 4.3 の 6 行目の数値 `1` を文字列 `"1"` に置きかえた場合、`sum(n)` はどのような答えを返すか?

練習 4.11 (再帰的に条件を満たす値を集める) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) 1 から `n` までの数のうち、`k` の約数になっているものの 個数(number of divisors) を求める `nod(k,n)`. (ヒント: 式 4.5 の足す数を変える。)

b) 1 から `n` までの数のうち、素数になっているものの個数 (number of primes) を求める `nop(n)`.

c) 1 から `n` までの数について、それぞれ自身を含むような約数の和を求めたとき、最大のもの (maximum sum of divisors) を求める `msod(n)`.

$\mathrm{sod}(k,k) = s_k$ と書くことにすると、この問題は s_1 から s_n の最大値を求めることに等しい。「s_1 から s_n の最大値」は「s_1 から s_{n-1} の最大値」より s_n が大きい場合とそうでない場合に分けて考えればよいので、次のような関係を満たしている。

$$\texttt{msod}(n) = \begin{cases} \texttt{msod}(n-1) & (\texttt{msod}(n-1) \geq s_n) \\ s_n & (\texttt{msod}(n-1) < s_n) \\ s_1 & (n = 1) \end{cases}$$

練習 4.12 (再帰的に条件を満たす値を探す) 以下のような値を求める再帰的な Ruby の関数を定義せよ。繰り返しは使わないこと。

a) `n` より大きくかつ最小の素数—つまり次の素数 (next prime number) —を求める `np(n)`. まず `n` が素数だった場合は `np(n)` = `n` である。そうでない場合は、`np(n)` は「`n`＋1 より大きくかつ最小の素数」に一致する。

b) 素数 `p` よりも大きな素数のうち n 番目の素数を求める `nth_prime(p,n)`. 例えば `nth_prime(5,3)` は 5 よりも大きな素数のうち 3 番目であり、5 の次の素数は 7 なので 7 よりも大きな素数のうち 2 番目つまり `nth_prime(7,2)` と等しい。これはさらに 7 の次の素数 11 よりも大きな素数のうち 1 番目と等しいので 11 の次の素数 13 が答になるはずである。

c) 第 3.2 節の式 3.1(p. 34) で紹介した関数 $\mathrm{tnpo}(n)$ は n が偶数なら 1/2, 奇数なら 3 倍して 1 加えた数を求めるものだった。数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ 1 になると予想した。例えば 3 から始めた場合は $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ といった具合に予想通りになっていることが確められる。

この予想は、反例は見つかっていないが証明もされていない、数学の未解決問題の 1 つである。

そこで n から上の手順で数を変化させて 1 になるまでの回数を $\mathrm{collatz}(n)$ とする。例えば $\mathrm{collatz}(5) = 5$, $\mathrm{collatz}(16) = 4$ である。

(a) collatz(n) と collatz(tnpo(n)) の関係を書け。

(b) collatz(n) を求める Ruby の関数 `collatz(n)` を定義せよ。

d) `n` 以上の整数で最小の完全数を見つける `next_perfect(n)`. ただし完全数 k とは k を除く約数の和が k と等しいような数のことである。例えば 6 の約数は (6 を除くと) 3, 2, 1 だけであり、その和が 6 になるので 6 は完全数である。最初に `n` が完全数であるかを判定する関数 `perfect(n)` を定義しておくとよい。

練習 4.13 (Eratosthenes の篩 *) 素数を求める方法の 1 つに Eratosthenes の篩(ふるい) というものがある。考え方はとしては「整数が素数かどうか」を記録する大きな表を用意して、素数を発見するたびにその倍数を表から消してゆくものである。

まず 2 から始まる整数の列を用意する。

2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,
52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,
77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,

列の先頭の数 2 はまだ消されていないので素数である。2 に丸を付け、2 の倍数を消す。

②, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9,~~10~~,11,~~12~~,13,~~14~~,15,~~16~~,17,~~18~~,19,~~20~~,21,~~22~~,23,~~24~~,25,~~26~~,
27,~~28~~,29,~~30~~,31,~~32~~,33,~~34~~,35,~~36~~,37,~~38~~,39,~~40~~,41,~~42~~,43,~~44~~,45,~~46~~,47,~~48~~,49,~~50~~,51,
~~52~~,53,~~54~~,55,~~56~~,57,~~58~~,59,~~60~~,61,~~62~~,63,~~64~~,65,~~66~~,67,~~68~~,69,~~70~~,71,~~72~~,73,~~74~~,75,~~76~~,
77,~~78~~,79,~~80~~,81,~~82~~,83,~~84~~,85,~~86~~,87,~~88~~,89,~~90~~,91,~~92~~,93,~~94~~,95,~~96~~,97,~~98~~,99,

次に消されていない数 3 も素数である。3 に丸を付け、3 の倍数を消す。

②,③, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,25,~~26~~,
~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~,~~34~~,35,~~36~~,37,~~38~~,~~39~~,~~40~~,41,~~42~~,43,~~44~~,~~45~~,~~46~~,47,~~48~~,49,~~50~~,~~51~~,
~~52~~,53,~~54~~,55,~~56~~,~~57~~,~~58~~,59,~~60~~,61,~~62~~,~~63~~,~~64~~,65,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
77,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,85,~~86~~,~~87~~,~~88~~,89,~~90~~,91,~~92~~,~~93~~,~~94~~,95,~~96~~,97,~~98~~,~~99~~,

次の数 4 は消されているので素数でない。その次の 5 が素数であるので同様に丸を付け倍数を消す。

②,③, ~~4~~,⑤, ~~6~~, 7, ~~8~~, ~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,~~25~~,~~26~~,
~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~,~~34~~,~~35~~,~~36~~,37,~~38~~,~~39~~,~~40~~,41,~~42~~,43,~~44~~,~~45~~,~~46~~,47,~~48~~,49,~~50~~,~~51~~,
~~52~~,53,~~54~~,~~55~~,~~56~~,~~57~~,~~58~~,59,~~60~~,61,~~62~~,~~63~~,~~64~~,~~65~~,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
77,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,~~85~~,~~86~~,~~87~~,~~88~~,89,~~90~~,91,~~92~~,~~93~~,~~94~~,~~95~~,~~96~~,97,~~98~~,~~99~~,

これを続けてゆくとやがて消されていない数が素数として残る。

②,③, ~~4~~,⑤, ~~6~~,⑦, ~~8~~, ~~9~~,~~10~~,⑪,~~12~~,⑬,~~14~~,~~15~~,~~16~~,⑰,~~18~~,⑲,~~20~~,~~21~~,~~22~~,㉓,~~24~~,~~25~~,~~26~~,
~~27~~,~~28~~,㉙,~~30~~,㉛,~~32~~,~~33~~,~~34~~,~~35~~,~~36~~,㊲,~~38~~,~~39~~,~~40~~,㊶,~~42~~,㊸,~~44~~,~~45~~,~~46~~,㊼,~~48~~,~~49~~,~~50~~,~~51~~,
~~52~~,㊿,~~54~~,~~55~~,~~56~~,~~57~~,~~58~~,~~59~~,~~60~~,61,~~62~~,~~63~~,~~64~~,~~65~~,~~66~~,67,~~68~~,~~69~~,~~70~~,71,~~72~~,73,~~74~~,~~75~~,~~76~~,
~~77~~,~~78~~,79,~~80~~,~~81~~,~~82~~,83,~~84~~,~~85~~,~~86~~,~~87~~,~~88~~,89,~~90~~,~~91~~,~~92~~,~~93~~,~~94~~,~~95~~,~~96~~,97,~~98~~,~~99~~,

この考え方をもとに、2 から `n` までの数が素数かどうかを示す配列を作る `primes(n)` を定義せよ。ただし、作られる配列の i 番目は、i が素数のとき 0, 素数でないとき 1 だとする。

練習 4.14 (Sierpinski の三角形) 大きさ `n` × `n` で、i 行 j 列目が ($_iC_j$ を 2 で割った余り) になっているような配列を作る関数 `sierpinski(n)` を定義せよ。この関数が作る配列を `show` を使って見ると図 4.1 のようになる (見易さ

のために白黒を逆にしている)。この図形は Sierpiński の三角形として知られているものである。

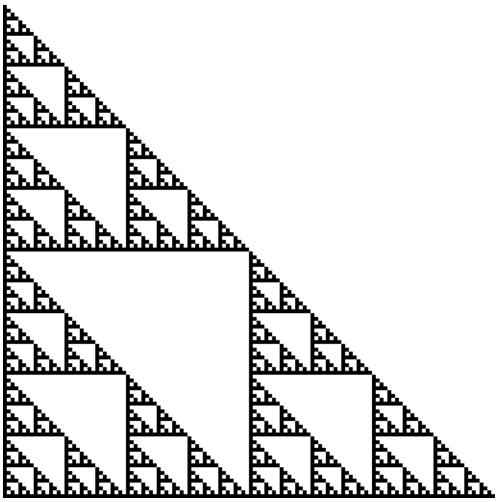


図 4.1: Sierpiński の三角形

練習 4.15 (見つからない場合) ファイル 4.8 の `match` の定義では `s` 中に `p` が必ず現われることを仮定していた。`p` が現われない場合に -1 と答える `match_safe(s,p)` を定義せよ。