プログラム

変数

- オブジェクトを「格納する」「指し示す」
- 変数名は英小文字(a-z)で始まる英数字と '_' からなる文字列です。

```
a
x
y
abc
aLPHA
```

予約語

変数名や関数名(後述)に使えない文字列です。

```
BEGIN    class    ensure   nil      self     when
END      def      false    not      super    while
alias    defined? for      or       then     yield
and      do       if       redo     true
begin    else     in       rescue   undef
break    elsif    module   retry    unless
case     end      next     return   until
```

定数

- 真偽値

`true ....` **真を表す値**
`false ...` **偽を表す値**
`nil .....` **初期化されていない（条件判断の際には偽と解釈されます）**

代入

- 変数にオブジェクトを割当てます。

```
x = 1
y = 2
str = "Hello"
```

演算子

- () で括れば演算順序を指定できます。
- a, b は実際には変数や定数やオブジェクトです。

```
-a ......... 符号の逆転
a + b  ..... 加算
a - b  ..... 減算
a * b  ..... 乗算
a / b  ..... 除算(整除)
a % b  ..... 剰余
a ** b ..... 冪乗

a && b ..... 論理積
a and b .... 論理積
a || b ..... 論理和
a or b ..... 論理和
!a ......... 論理否定
not a ...... 論理否定

a < b ...... 比較(a は b より小さい)
a <= b ..... 比較(a は b 以下)
a >= b ..... 比較(a は b 以上)
a > b ...... 比較(a は b より大きい)
a == b ..... 比較(a と b が等しければ true 、異なれば false)  ※ "=" ではない！
a <=> b .... 比較(a > b ならば 1 ,a と b が等しければ 0 、a < b ならば -1)

a += b ..... 変数 a の値を a + b の値にする
a -= b ..... 変数 a の値を a - b の値にする
a *= b ..... 変数 a の値を a * b の値にする
a /= b ..... 変数 a の値を a / b の値にする
a %= b ..... 変数 a の値を a % b の値にする
a **= b .... 変数 a の値を a ** b の値にする
```

範囲指定演算子

```
a .. b      両端を含む範囲
a ... b     終端(b)を含まない範囲
```

式

- プログラムの最小要素。
- 全ての文/式は値を持ちます。

```
x = 1
x = ((x + 1) * (x - 1))**2
```

```
true && false
true || false
```

コメント

'#' から行末までの文はすべて無視されます

制御構造

プログラムの実行の流れを制御します。

文 (逐次実行)

改行もしくは ';' で区切られた文(式)は順番に実行されます。

```
a = 1
b = 1; c = 4
print a, "=",
   b, "<", c, "\n"
```

if 文 (条件分岐)

```
if a1 then
  b1
elsif a2 then
  b2
else
  b3
end
```

もし a1 が真であれば b1 を実行、そうでなければ、もし a2 が真であれば b2 を実行、そうでなければ、b3 を実行します。

- then は省略してもよいです。
- elsif 節は複数あってもよいし、なくてもよいです。
- else 節はあってもなくてもよいです。

真とはなにか？ false と nil は「偽」、それ以外のすべての値は「真」です。

```
n = gets.to_i    # 文字列を読み込んで整数を返します。
if n > 0 then    # n が正だったら
 puts n         # n を印字
else             # さもなければ
 puts -n        # -n を印字
end
```

注: unless, case 文については省略します(if 文で代替可能)。

while 文 (ループ)

```
while a
 b
end
```

条件部 a が真である限り、b を実行し続けます。

```
a = 5          # a に 5 を代入
while a > 0    # a が正である限り
 puts a**3     # a の 3 乗を計算して印字
 a -= 1        # a の値を 1 減らす
end            # ループの最初(while ...)に戻る
```

プログラムの書き方によっては止まらなくなります。

```
a = 5          # a に 5 を代入
while a > 0    # a が正である限り
 puts a**3     # a の 3 乗を計算して印字
 a += 1        # a の値を 1 増やす
end            # ループの最初(while ...)に戻る
```

こういうときは RDE のメニューで "Run" -> "Terminate Process" を選択すると 強制的に実行を止めてくれます。それでも駄目だったら RDE 自体を強制終了させます。

注: until 文については省略します(while 文で代替可能)。

for 文 (ループ)

```
for var in obj
 c
end
```

obj 内の各要素を順番に var へ代入し、c を実行します。

```
for x in 0 .. 5
 puts x*x
end

for x in 0 ... 5
 puts x*x
end
```

break, next, redo (ループ脱出)

while / for ループ中では break, next, redo という命令が利用できます。

- break : そこでループを終了します。
- next : ループの残りを実行せずに、次のループへ進みます。
- redo : 現在のループを再度実行します、条件部の再評価はしません。

例: 1+2+...n > 1000 となる最小の n は？ 1+2+...n = n(n+1)/2 ですから n = 45 が解です。

```
s = 0              # 1+2+...n を保存しておくための変数 s の初期化
n = 0
while s <= 1000    # 1+2+...n が 1000 以下である限り
 n += 1           # n を 1 増やす
 s += n           # 1+2+...n を計算・保存
end
puts n             # n を印字

s = 0              # 1+2+...n を保存しておくための変数 s の初期化
n = 0
while true         # 無限ループ
 n += 1           # n を 1 増やす
 s += n           # 1+2+...n を計算・保存
 if s > 1000      # 1+2+...n が 1000 以上になったらループを終了
  break
 end
end
puts n             # n を印字
```

例: コラッツ-角谷の予想

任意の正数(初期値)に対して

- 偶数ならば 2 で割る
- 奇数ならば 3 倍して 1 を加える(必ず偶数となる)

という操作を繰り返すと必ず 1 になります。いまだに証明されていない(はずです)。

```
x = gets.to_i      # 入力
while x > 1        # x が 1 より大きい限りループ
 if x % 2 == 0    #
  x /= 2          # x が偶数(2 の剰余が 0)ならば x を 2 で割った値を x とする
 else              #
  x = 3*x+1       # さもなければ(x が奇数ならば) 3x+1 を x とする
 end               #
 puts x            # x を出力する
end
```

問 1: 途中経過の代りに、ループ終了時にループをどれだけ回ったかを出力するようにしてみましょ

う。

```
x = gets.to_i      # 初期値の入力
count = 0          # ループの回数を記録しておく変数
while x > 1        # x が 1 より大きい限りループ
 if x % 2 == 0    #
  x /= 2          # x が偶数(2 の剰余が 0)ならば x を 2 で割った値を x とする
 else              #
  x = 3*x+1       # さもなければ(x が奇数ならば) 3x+1 を x とする
 end               #
 count += 1        # count を 1 増やす
end                #
puts count         #
```

問 2: 1 から 100 まで動かしたときに、一番たくさんループを回る初期値を見付けましょう。

問 3: 調べる範囲の上限を 100 ではなく 1000, 10000 とした場合は？

```
e = 100              # 調べる範囲の上限
n = 1                # 一番たくさんループを回った初期値を保存する変数
c = 0                # 一番たくさん回ったループの回数を保存する変数
for i in (1 .. e)
 x = i
 count = 0           # ループの回数を記録しておく変数
 while x > 1         # x が 1 より大きい限りループ
  if x % 2 == 0
   x /= 2            # x が偶数(2 の剰余が 0)ならば x を 2 で割った値を x とする
  else
   x = 3*x+1         # さもなければ(x が奇数ならば) 3x+1 を x とする
  end
  count += 1         # count を 1 増やす
 end
 if count > c        # いままでで一番たくさんループを回ったならば
  n = i              # そのときの初期値を保存
  c = count          # そのときのループ回数を保存
 end
end
puts n, c            # 結果を出力する
```

論理演算

if や while などの条件部に書く論理式を組み立てるときに使われる論理演算に and / or / not があります。それぞれ、and (論理積)は「かつ」、or (論理和)は「または」、not (論理否定)は「でなければ」に相当します。

and / or　　not

A	*B*	*A* and *B*	*A* or*B*
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

A	true	false
not*A*	false	true

また and は && 、or は || 、not は！と書くこともできます(基本的には好みの問題、文法的には若干の違いがありますが)。

複雑な論理式では演算の順序を明確にするために括弧を使いましょう。たとえば (*A* and *B*) or *C* と *A* and (*B* or *C*) は異なる論理式です。

例 1: x は y 以上 → x => y あるいは not (x < y)

例 2: x が y より小さく、かつ、x^2 が y より大きい。→ x < y and x**2 > y

例 3: x と y が等しいか、x が負値かつ x^2 と y^2 が等しい → x == y or (x < 0 and x**2 == y**2)

関数(メソッド)定義

ある操作について名前を付けます。

```
def funname(var1, var2, ...)
 commands
end
```

funname が関数(メソッド)名、var1 を仮引数(カリヒキスウ)といいます。仮引数はゼロ個でも 1 個でも複数個でもかまいません。また関数名は英小文字(a-z)で始まる英数字と '_' からなる文字列です(このあたり厳密には嘘なので 正確なところはリファレンスマニュアルを参照してください)。

関数は funname(e1, e2, ...) という形で呼出されます(関数呼出し)。このとき e1, e2 を引数(ヒキスウ)と呼びます。引数の数は関数定義の仮引数の数と一致します。

```
def square(n)      # 2 乗を計算する関数の定義
 return n**2
end

puts square(10)    # 関数呼出し
puts square(20)    # 関数呼出し
```

return n でその関数を終了して値(n)を返します。関数内の最後(end の直前)まで実行された場合は 最後の文を実行して、その値を返します。

```
def sum_of_squares(x,y)  # x^2 + y^2 を計算する関数の定義
 return x*x + y*y
end

puts sum_of_squares(3,4)
```

同じ値を返す関数でも定義の方法は複数あることが普通です。

- 階乗 n! の定義 1: 1 から n までの整数を乗算した値
- 階乗 n! の定義 2: n が 0 ならば 1 、そうでなければ n! = n * (n-1)!

```
def fact1(n) # 階乗の計算(1)
 s = 1
 for i in 1 .. n
  s *= i
 end
 return s
end

def fact2(n) # 階乗の計算(2) ... 再帰的定義
 if (n > 0)
  return n * fact2(n-1)
 else
  return 1
 end
end

puts fact1(10)       # 10!
puts fact2(10)       # 10!
puts fact1(fact2(5)) # (5!)! = 120!

---------------------------------------------------------------------------
fact1(5)
 s = 1
 i = 1
 s *= i    # s は 1
 i = 2
 s *= i    # s は 1*2 = 2
 i = 3
 s *= i    # s は 1*2*3 = 6
 i = 4
 s *= i    # s は 1*2*3*4 = 24
 i = 5
 s *= i    # s は 1*2*3*4*5 = 120
 return s  # fact1(5) の値は 120 となる

fact2(5)
 5 * fact2(4)                    # n = 5 > 0 だから
 5 * 4 * fact2(3)                # n = 4 > 0 だから
 5 * 4 * 3 * fact2(2)            # n = 3 > 0 だから
 5 * 4 * 3 * 2 * fact1(1)        # n = 2 > 0 だから
 5 * 4 * 3 * 2 * 1 * fact1(0)    # n = 1 > 0 だから
 5 * 4 * 3 * 2 * 1 * 1           # n = 0 なので fact1(n) = 1
 5 * 4 * 3 * 2 * 1
 5 * 4 * 3 * 2
 5 * 4 * 6
 5 * 24
```

```
 120                                # fact2(5) の値は 120 となる
--------------------------------------------------------------------------
```

変数のスコープ: 仮引数や関数内で定義された変数は、関数の外からは見えません。関数外や別関数内に同名の変数があっても、それは無関係です。

```
def f(x)
 y = x+1
 return y
end

x=5
y=10
puts x + f(y)
```

問 2: 2 つの数を引数として大きい方の値を返す関数 max と、小さい方の値を返す関数 min を定義してください。

```
def max(x,y)
 if x > y
  return x
 else
  return y
 end
end

def min(x,y)
 if x < y
  return x
 else
  return y
 end
end
```

問 3: 引数 n (> 0)に対して $m^2 < n$ となる最大の m を返す関数 isquare を定義してみましょう。

```
def isquare(n)
 i=0
 while true
  if (i+1)**2 > n
   return i
  end
  i += 1
 end
end

puts isquare(99)
```

暇な人向けの問: Ackermann 関数

```
def a(x,y)
 if x==0    then return y+1
 elsif y==0 then return a(x-1,1)
 else            return a(x-1,a(x, y-1))
 end
end

def f0(n) return a(0,n) end
def f1(n) return a(1,n) end
def f2(n) return a(2,n) end
def f3(n) return a(3,n) end
def f4(n) return a(4,n) end
```

1. 計算結果を元に関数 f1, f2, f3, f4 の簡潔な数学的定義を予測してみましょう。
2. その予測が正しいことを証明してみましょう。
3. 自分の PC 上で f3(n), f4(n) は n がどれくらいの大きさになるまで計算できるでしょうか？

問 4: (この授業では) Fibonacci 数列の定義は以下のとおりです。

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ $(n > 1)$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

n を引数として F_n を返す関数を定義してみましょう。

```
def fib1(n)
 if n==0
  return 0
 elsif n==1
  return 1
 else
  return fib1(n-1) + fib1(n-2)
 end
end

for n in 0 .. 5
 puts fib1(n)
end
```

```
------------------------------------------------------------------------
fib1(5)
fib1(4) + fib1(3)
fib1(3) + fib1(2) + fib1(2) + fib1(1)
fib1(2) + fib1(1) + fib1(1) + fib1(0) + fib1(1) + fib1(0) + fib1(1)
fib1(1) + fib1(0) + fib1(1) + fib1(1) + fib1(0) + fib1(1) + fib1(0) + fib1(1)
1 + 0 + 1 + 1 + 0 + 1 + 0 +1
5
------------------------------------------------------------------------
```

問 5(発展課題): n を大きくしていくと計算時間はどうなるでしょうか？

まず F_n の一般形を求める。二次方程式 $z^2-z-1 = 0$ の(異なる)2 つの解を各々

- $\alpha = (1+\sqrt{5})/2 = 1.6180$ ... ← いわゆる黄金比
- $\beta = (1-\sqrt{5})/2 = 1-\alpha = -0.6180$ ...

とおくと

- $(\alpha^0 - \beta^0)/\sqrt{5} = (1-1)/\sqrt{5} = 0$
- $(\alpha^1 - \beta^1)/\sqrt{5} = (\alpha-\beta)/\sqrt{5} = \sqrt{5} / \sqrt{5} = 1$
- $(\alpha^n - \beta^n)/\sqrt{5} -(\alpha^{n-1} - \beta^{n-1})/\sqrt{5} -(\alpha^{n-2} - \beta^{n-2})/\sqrt{5}$
  $= (\alpha^{n-2}(\alpha^2-\alpha-1) - \beta^{n-2}(\beta^2-\beta-1))/\sqrt{5}$
  $= (\alpha^{n-2}\ 0 - \beta^{n-2}\ 0) / \sqrt{5}$
  $= 0$

という関係が成立する。したがって任意の n について $F_n = (\alpha^n - \beta^n)/\sqrt{5}$ と表現できる。

より一般的な解法はたぶん線形代数の講義あたりで学習している(あるいはする) と思いますが、たとえば番号を 1 個ずらす線型写像を考えてそれを表現する行列の 冪乗の一般形を求める問題に帰着させるという方法があります。いま考えている Fibonacci 数列の場合はその行列が

$$R = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

ですから R の特性方程式 $z^2-z-1 = 0$ が現れてくると。また初期値 F_0 と F_1 を変更した場合も α^n と β^n の線形結合で F_n が表現されることも わかります。

β の絶対値は 1 未満で α は 1 より大きな正数であるから、n が大きくなると F_n は指数関数的に増大することになる。また F_n と F_{n-1} の比は α に「近づく」。

問 4 のプログラム

```
def fib1(n)
 if n==0
  return 0
 elsif n==1
  return 1
 else
  return fib1(n-1) + fib1(n-2)
 end
end
```

について fib1(n) を計算するために必要な時間を T_n とする。大雑把に考えて $T_n = T_{n-1} + T_{n-2}$ であるから、T_n は Fibinacci 数列 F_n と同様の増加傾向を辿ることになり、ほぼ α^n に比例すること。

問 6: もっと「効率良く」Fibonacci 数列を計算する関数は定義できるでしょうか？

```
def fib2(n)
 a = 1        # F(1)
 b = 0        # F(0)
```

```
 while n > 0  # n 回ループする
  t = a
  a += b
  b = t
  n -= 1
 end
 return b
end

for n in 0 .. 5
 puts fib2(n)
end

# コメント: Ruby では a, b = a+b, a という多重代入も可能だがここでは触れない
```

問 7: for n in 0 .. 25 くらいで fib1 と fib2 の計算時間を比較してみましょう。

問 8: fib2(n) で F_n が計算できることの証明をしてみましょう。

$$V_n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

$$R = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

と定義したとき

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

すなわち $V_n = R\,V_{n-1}$ となる。よって

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

すなわち $V_n = R^n V_0$ が成立する。ここで

$$V_0 = \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

したがって、まず

```
a = 1
b = 0
```

と変数 a, b の値を代入後に a, b を(同時に) a+b, a で置き換える 計算を n 回くりかえすと $a = F_{n+1}$, $b = F_n$ となっている(証明終り)。

このように「アルゴリズム」が異なると計算時間などが大幅に違ってくることがあります。したがって、ある問題を解くときにより効率的なアルゴリズムを適用するのが賢い方法です。

問 9(発展課題): fib2(n) の計算には n 回ループを廻す必要がありますが、もうちょっと 頑張ると log(n) 回程度のループで済ますことが可能です。そのような関数 fib3(n) を定義してみましょう。

$$V_n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

$$R = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

と定義したとき

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

すなわち $V_n = R\, V_{n-1}$ となる。よって

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

すなわち $V_n = R^n V_0$ が成立する。ここで

$$V_0 = \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

さて R は対称行列であるから R^n も対称行列となる。そこで

$$R^n = \begin{pmatrix} a_n & b_n \\ b_n & c_n \end{pmatrix}$$

と a_n, b_n, c_n $(n > 0)$ を定義すると $a_1 = 1, b_1 = 1, c_1 = 0$ である。さて次の式

$$\begin{pmatrix} a_{2n} & b_{2n} \\ b_{2n} & c_{2n} \end{pmatrix} = R^{2n} = R^n R^n = \begin{pmatrix} a_n & b_n \\ b_n & c_n \end{pmatrix}\begin{pmatrix} a_n & b_n \\ b_n & c_n \end{pmatrix} = \begin{pmatrix} a_n a_n + b_n b_n & b_n(a_n + c_n) \\ b_n(a_n + c_n) & b_n b_n + c_n c_n \end{pmatrix}$$

を考慮すると R, R^2, R^4, R^8, ... という R^{2i} (i=0,1,2...)の列が次々と計算できる。したがって n の 2 進数表記を $n_k\, n_{k-1} \ldots n_0$ (各 n_i は 0 か 1 の値をとる)とすれば $R^n = R^{n_k 2^k + n_{k-1} 2^{k-1} + \ldots + n_0 2^0} = R^{n_k 2^k} R^{n_{k-1} 2^{k-1}} \cdots R^{n_0 2^0}$

$V_n = R^{n_k 2^k} R^{n_{k-1} 2^{k-1}} \cdots R^{n_0 2^0} V_0$

となる。したがって R^{2i} (i=0,1,2...k)を次々と計算し n_i が 1 だったら R^{2i} を V_0 に対して乗ずる、という操作を k 回おこなうと V_n (すなわち F_n)が求まることになる。この「アルゴリズム」を ruby でプログラムしたものが fib3 である。

```
# n 回ループを廻るアルゴリズム
def fib2(n)
 a = 1; b = 0
 while n > 0
  t = a; a += b; b = t; n -= 1
 end
 return b
end
```

```
# 約 log(n) 回ループを廻るアルゴリズム
def fib3(n)
 m = n              # n をそのまま使ってよいが説明の便宜上 m という新しい変数を使う
 i = 0              # 不要な変数だが説明の便宜上用意してある
 x=1; y=0           # 初期値は V(0) の要素 F(1) と F(0)
 a=1; b=1; c=0      # R^(2^i) の要素 a,b,c が記録される変数
 while m > 0        # for i in 1 .. k と同じ
  if m % 2 == 1  # m の 2 進表記の 0 桁目 (最下位ビット) が 1 である
          # つまり n の 2 進表記の i 桁目が 1 である
    x2 = x; y2 = y
    x = a*x2 + b*y2
    y = b*x2 + c*y2
  end
  i += 1  # 着目する桁を 1 つ進める
  m /= 2  # 余りを無視し 2 で割った値で m を置き換える (右シフトする)
  # R^(2^i) の計算
  aa = a*a; bb = b*b; cc = c*c; a_c = a+c
  a = aa + bb; b *= a_c; c = bb + cc
 end
 # while ループ終了時には V(n) が計算できている
 return y
end

for i in 0 .. 15
 # i に対して幅 2 桁で 10 進表記 (幅 4 桁で 2 進表記)
 # fib2(i) の値を幅 4 桁で 10 進表記
 # fib3(i) の値を幅 4 桁で 10 進表記
 printf("%2d(%4b) %4d %4d\n", i, i, fib2(i), fib3(i))
end
```

配列

- 配列とはオブジェクトを順番に並べたもので、これもまたひとつのオブジェクトです。
- 配列の各要素には整数のインデックスを使ってアクセスできます。インデックスは 0 から始まります。
- 任意のオブジェクトを配列の要素にできます。

配列の作り方

配列の各要素を ',' で区切り、'[' と ']' で括ると配列が生成できます。

```
[]
```

要素がゼロ個の空な配列です。

```
a = [3, 1, 4]
puts a[0]
```

```
puts a[1]
puts a[2]
```

配列要素がすべて整数である配列 a を定義しています。

```
b = ['apple', 'banana', 'orange']
puts b[0]
puts b[1]
puts b[2]
```

配列要素がすべて文字列である配列 b を定義しています。

```
c = ['apple', 12345, 2.71828]
puts c[0]
puts c[1]
puts c[2]
```

配列要素に文字列・整数・浮動小数点数が混ざっている配列 c を定義しています。

```
d = ["apple", [["banana", "orange"], "melon"], "kiwi"]

puts d[0]          # "apple"
puts d[1][0][0]    # "banana"
puts d[1][0][1]    # "orange"
puts d[1][1]       # "melon"
puts d[2]          # "kiwi"

puts d[-1]     # 負値は末尾から数えたインデックスと解釈されます
puts d[1024]   # 定義されていないインデックスに対応する配列要素は nil
```

配列メソッド

オンラインマニュアル には配列についての様々なメソッドが記載されていますが、ここでは最小限に 留めます(最初から便利なメソッドを使ってしまうと勉強にならない、という 観点も含んでいます)。

a[i]	i 番目の要素
a[i..j]	i 番目から j 番目までの要素(からなる配列)
a[i, n]	i 番目から n 個の要素(からなる配列)、つまり a[i, i+n-1]
a[i] = x	i 番目に x を代入
a[i..j] = b	i 番目から j 番目までに配列 b を代入、つまり a[i] = b[0], … a[j] = b[j-i]
a[i, n] = b	i 番目から n 個の要素に配列 b を代入、つまり a[i] = b[0], … a[i+n-1] = b[n-1]
a.length	配列 a のサイズ(インデックスの最大値 + 1)
a.size	配列 a のサイズ、a.length と同等
a.pop	末尾の要素の取り出し
a.push(x)	x を末尾に追加

a.shift	先頭の要素の取り出し
a.unshift(x)	x を先頭に追加

```
a = ["apple", "orange"]
puts a[0]                   # apple
puts a.length               # 2
a[0] = "banana"
puts a[0]                   # banana
a.push("melon")
puts a[-1]                  # melon
puts a.length               # 3
puts a.pop                  # melon
puts a.length               # 2
puts a.pop                  # orange
puts a.length               # 1
puts a.pop                  # banana
puts a.length               # 0
```

問 10: 月(1 から 12)を入力すると、その月の日数を表示するプログラムを作ってみましょう(うるう年は考えないことにします)。

```
#      0   1   2   3   4   5   6   7   8   9  10  11  12
a = [nil, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
m = gets.to_i
puts a[m]
```

問 11: 与えられた(要素が整数の)配列から 2 乗の最大値を探す関数を定義しましょう。

```
def m(a)
 ...       # ここを考える
end

puts m([3, 7, 8, 1, -5, 9, 0])    #  81 が表示される

# 配列のインデックスを使って計算する
def m1(a)
 m = 0                    # 最大値を記録しておく変数
 for i in 0 ... a.size    # i を 0 から a.size - 1 まで動かす
  v = a[i]*a[i]
  if v > m
   m = v                # a[i]**2 がいままでの最大値より大きければ m の値を更新
  end
 end
 return m
end

puts m1([3, 7, 8, 1, -5, 9, 0])

# 配列の要素を使って計算する
def m2(a)
 m = 0                    # 最大値を記録しておく変数
 for v in a               # 配列 a の各要素を変数 v に代入していく
```

```
  vv = v*v
  if vv > m
   m = vv              # vv がいままでの最大値より大きければ m の値を更新
  end
 end
 return m
end

puts m2([3, 7, 8, 1, -5, 9, 0])
```

問 12: 与えられた(要素が整数でサイズが等しい)配列 a, b に対して、各要素の積の和(Σa[i]b[i])を計算する関数を定義しましょう。

```
def s(a,b)
 ...       # ここを考える
end

a = [1,2,3,4,5]
b = [5,4,3,2,1]
puts s(a,b)      # 35 が表示される

def s1(a,b)
 sum = 0                    # 総和を記録しておく変数
 for i in 0 ... a.size      # インデックス i を配列 a,b の全体に動かす
  sum += a[i]*b[i]          # 配列 a,b の i 番目の要素の積を計算して sum に加える
 end
 return sum
end

def s2(a,b)
 if a.size == 0             # a,b が空の配列であれば s2(a,b) は 0 となる
  return 0
 else
  v = a.shift * b.shift     # 配列 a,b から先頭要素(つまり a[0] と b[0])を取り出して積を計算
  return v + s2(a,b)        # (先頭要素を取り除いた) a,b について s2(a,b) を計算して加える
 end
end

a = [1,2,3,4,5]
b = [5,4,3,2,1]
puts s1(a,b)
puts s2(a,b)
```

入出力

コンソール入出力:

関数名	意味	備考
gets	コンソールからの文字列の入力	

gets.to_i	コンソールからの整数の入力	本当は gets が返す文字列へ to_i メソッドを送っている
gets.to_f	コンソールからの浮動小数点数の入力	本当は gets が返す文字列へ to_f メソッドを送っている
puts(*obj*)	コンソールへの出力	()は省略可能、最後に改行を出力
p(*obj1*,...)	コンソールへの出力(どちらかというとデバッグ用？)	()は省略可能
print(*arg1*,...)	コンソールへの出力	()は省略可能
printf(*format*,*arg*,...)	コンソールへのフォーマット出力	()は省略可能、フォーマットの詳細は オンラインマニュアルを参照

演習:

```
s = "いろはにほへと"
i = 12345
f = 3.1415
printf("s は %s で i は %d で f は %f です。\n", s, i, f)
```

問 13: 1 から n までの間にある素数を列挙するプログラムを作ってみましょう。n はプログラム中に書き込むことにして、とりあえず 100 とします。

```
# いわゆる「エラトステネス(Eratosthenes)のふるい」による計算
n = 100

a = [false, false]      # 0,1 は素数ではない(合成数である)
for i in 2 .. n do      # 2 から n まで順にチェックする
 if a[i] == nil         # a[i] が素数か合成数か未チェックである
  a[i] = true           # i は素数である、と a[i] に記録する
  k = 2*i               # ↑
  while k <= n do       # ↑
   a[k] = false         # 素数 i の倍数 k は合成数である、と a[k] に記録する
   k += i               # ↓
  end                   # ↓
 end
end

for i in 0 .. n do      # i を 0 から n まで動かしたとき
 if a[i]                # もし a[i] が true である(つまり i が素数である)
  puts i                # ならば i を表示する
 end
end

--------------------------------------------------------------------------------
n = 10 の場合 (N -> nil, T -> true, F -> false)

      0  1  2  3  4  5  6  7  8  9  10
```

```
a -> [F, F, N, N, N, N, N, N, N, N, N, ...]

i = 2, a[i] == nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, N, N, N, N, N, N, N, N, ...]
a -> [F, F, T, N, F, N, F, N, F, N, F, ...]

i = 3, a[i] == nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, N, F, N, F, N, F, ...]
a -> [F, F, T, T, F, N, F, N, F, F, F, ...]

i = 4, a[i] == false != nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, N, F, N, F, F, F, ...]

i = 5, a[i] == nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, N, F, F, F, ...]

i = 6, a[i] == false != nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, N, F, F, F, ...]

i = 7, a[i] == nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, T, F, F, F, ...]

i = 8, a[i] == false != nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, T, F, F, F, ...]

i = 9, a[i] == false != nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, T, F, F, F, ...]

i = 10, a[i] == false != nil
      0  1  2  3  4  5  6  7  8  9  10
a -> [F, F, T, T, F, T, F, T, F, F, F, ...]
```

問 14: 1 から n までの間にある素数の個数を列挙するプログラムを作ってみましょう。n はプログラム中に書き込むこととし、100,1000,10000,10000 と変更して実行してみましょう。

```
n = 100                 # 100,1000,10000,10000 と変更してみる
a = [false, false]      # 0,1 は素数ではない(合成数である)
for i in 2 .. n do      # 2 から n まで順にチェックする
 if a[i] == nil        # a[i] が素数か合成数か未チェックである
  a[i] = true          # i は素数である、と a[i] に記録する
  k = 2*i              # ↑
  while k <= n do      # ↑
   a[k] = false       # 素数 i の倍数 k は合成数である、と a[k] に記録する
   k += i             # ↓
  end                  # ↓
 end
```

```
end

counter = 0
for i in 0 .. n do      # i を 0 から n まで動かしたとき
 if a[i]               # もし a[i] が true である(つまり i が素数である)
  counter += 1        # ならば counter を 1 増やす
 end
end

puts counter            # 0 から n までの間に存在する素数の数
```

問 15(発展課題): 問 13,14 のプログラムをより効率的にしてみましょう。
例:

- 2 以外の偶数は、必ず合成数です。
- ab = n であれば a と b のどちらかは √n より小さいはずです。
- ...

問 16: 配列を利用して問 4 の fib1 を効率化してみましょう。
ヒント: fib1 では同じ計算を何度も繰り返しています。

```
memory = [0,1]    # 定義より fib4(0) = 0, fib4(1) = 1 である

def fib4(m, n)    # 一度計算した値は配列 m に記憶しておいて再利用する
 if m[n] == nil                              # まだ fib4(n) が未計算であれば
  m[n] = fib4(m, n-1) + fib4(m, n-2)    # 計算して配列 m に記憶する
 end
 return m[n]                                 # 計算済みの値を返す
end

for n in 0 .. 5         # 上限を 10, 15, 20, 25, 30 と大きくして
 puts fib4(memory,n)  # fib1 と計算時間を比較してみましょう
end
```

絶対値・整数への丸め

以前に触れておくべき項目でしたが、おそまきながら。以下で *y* は整数もしくは浮動小数点数です。

***y*.abs**	絶対値
***y*.ceil**	整数への丸め(天井)
***y*.floor**	整数への丸め(床)
***y*.round**	整数への丸め(四捨五入)
***y*.truncate**	整数への丸め(切り捨て)

```
p = -3.14
```

```
puts p.abs
puts p.ceil
puts p.floor
puts p.round
puts p.truncate

puts (-3.14).abs
puts (-3.14).ceil
puts (-3.14).floor
puts (-3.14).round
puts (-3.14).truncate

def out(i)
 printf("%f %f %f %f %f %f\n", i, i.abs, i.ceil, i.floor, i.round, i.truncate)
end

for n in [1.9, 1.1, -1.1, -1.9]
 out(n)
end
```

問 17: 浮動小数点数 x を入力したときにその abs, ceil, floor, round, truncate を表示するプログラムを考えてみましょう。

```
x = gets.to_f
printf("%f %f %f %f %f\n", x.abs, x.ceil, x.floor, x.round, x.truncate)
```

数学関数

いくつかの数学的な関数と定数が定義済みです。詳細については オンラインマニュアル を参照してください。

acos(*x*)	ラジアン単位での逆三角関数
asin(*x*)	ラジアン単位での逆三角関数
atan(*x*)	ラジアン単位での逆三角関数
cos(*x*)	ラジアン単位での三角関数
sin(*x*)	ラジアン単位での三角関数
tan(*x*)	ラジアン単位での三角関数
exp(*x*)	指数関数 e^x
log(*x*)	自然対数
log10(*x*)	常用対数
sqrt(*x*)	平方根 $\sqrt{x}$
E	自然対数の底 2.718281828...
PI	円周率 3.1415926...

```
include Math    # 数学関数を利用するときの「おまじない」

printf("PI=%f, E=%f\n", PI, E)
```

```
x = PI/4
printf("x=%f, sin(x)=%f, cos(x)=%f, tan(x)=%f\n", x, sin(x), cos(x), tan(x))
x = 10
printf("x=%f, exp(x)=%f, log(x)=%f, log10(x)=%f, sqrt(x)=%f\n",
    x, exp(x), log(x), log10(x), sqrt(x))
```

問 18: 浮動小数点数 x を入力すると sin(x) + cos(x) を表示するプログラムを考えてみましょう。

```
include Math

x = gets.to_f
printf("%f\n", sin(x)+cos(x))
```

問 19: 浮動小数点数 x を入力すると log(x) + exp(x) を表示するプログラムを考えてみましょう。x が負値のときはどうなるでしょうか？

```
include Math

x = gets.to_f
printf("%f\n", log(x)+exp(x))
```

問 20: 整数 n を入力すると tan(PI/n) を表示するプログラムを考えてみましょう。

```
include Math

n = gets.to_i
printf("%f\n", tan(PI/n))
```

問 21: 整数 n を入力すると tan(PI/n) と atan(tan(PI/n) を表示するプログラムを考えてみましょう。

```
include Math

n = gets.to_i
z = PI/n
x = tan(z)
y = atan(x)
printf("%f %f %f %f\n", z, x, y, z-y)
```

ユークリッド(Euclide)の互除法

整数 a, b に対して

- どちらの約数にもなっている最大の自然数を 最大公約数 gcd (greatest common divisor)
- どちらの倍数にもなっている最小の自然数を 最小公倍数 lcm (least common multiple)

といいます。もうすこし厳密に定義しますと

`a` と `b` の最大公約数 = `max { k | k` は `a` を割り切る かつ `k` は `b` を割り切る `}`
`a` と `b` の最小公倍数 = `min { k | a` は `k` を割り切る かつ `b` は `k` を割り切る かつ `k > 0 }`

※ `a,b` のどちらかが `0` のときは `gcd(a,0)=a, lcm(a,0)=0` と定義するのが便利なようです。

問 22: 最大公約数を計算する関数 gcd(a,b) を定義してみましょう。

最大公約数の定義から

```
gcd(a,0) = a  ... (1)
```

また r = a%b とおくと a = nb+r (n は適当な整数)で

- k が a, b を割り切る -> k は r も割り切る
- k が b, r を割り切る -> k は a = nb+r も割り切る

したがって集合として考えると

```
{ k | k は a を割り切る かつ k は b を割り切る } = { k | k は b を割り切る かつ k は r を割り切る }
```

よって

```
gcd(a,b) = gcd(b, a%b)  ... (2)
```

```
さて a < b ならば a%b = a なので (2) を一回適用すると a,b が入れ替わる。
したがって a >= b と仮定してよい。a >= b であれば a%b < b である。
したがって (2) を適用して a,b の値を小さくしていけば最終的には (1) に帰着する。
```

上記にしたがって作ったプログラムは以下のとおり。

```
def gcd1(a,b)
 if b == 0
  c = a
 else
  c = gcd1(b, a%b)
 end
 return c            # 変数 c を使わずに直接 return してもよい
end

def gcd2(a,b)
 while b > 0
  # a, b <- b, a%b
  t = a
  a = b
  b = t % b
 end
 return a
end

a = 1234567
b = 7654321
printf("gcd1(%d,%d)=%d\n", a, b, gcd1(a,b))
printf("gcd2(%d,%d)=%d\n", a, b, gcd2(a,b))
```

問 23: 最小公倍数を計算する関数 lcm(a,b) を定義してみましょう。

```
G = gcd(a,b)
L = lcm(a,b)
```

```
とおくと、適当な互いに素である正数 X,Y を用いて

a = X*G
b = Y*G
```

と書ける。このとき a*b/G = X*Y*G は a と b 両方の倍数なので
最小公倍数 L の倍数になっている。そこで適当な正数 U,V を用いて

```
L = U*a = U*X*G
L = V*b = V*Y*G
```

すなわち

```
U*X = V*Y
```

が成立する。ここで X と Y は互いに素なので U は Y の倍数、V は X の倍数で
ある。もし U > Y ならば L = U*X*G > Y*X*G = X*Y*G となり最小公倍数 L より小
さな X*Y*G が a と b の公倍数であることになり矛盾。V > X と仮定しても同様
である。
したがって V=X かつ U=Y となるので L = U*a = Y*X*G 、すなわち a と b の最小
公倍数 L は a*b/G に等しい。

式を整理すると gcd(a,b) * lcm(a,b) = a*b となる。

※ 素因数分解の一意性を使うともっと簡単に証明できます。

上記にしたがって作ったプログラムは以下のとおり。

```
def gcd(a,b)
 if b == 0
  return a
 else
  return gcd(b, a%b)
 end
end

def lcm(a,b)
 return a * b / gcd(a,b)
end

a = 1234567
b = 7654321
printf("lcm(%d,%d)=%d\n", a, b, lcm(a,b))
```

問 24: 整数 a, b に対して a m + b n = gcd(a,b) となる m,n が必ず存在します。この m,n と c=gcd(a,b) を計算する関数 egcd を定義してみましょう。

b = 0 であれば gcd(a,b) = a なので m = 1, n = 0, c=a とすればよい(*)。

さて b > 0 に対して b * m2 + (a%b) * n2 = gcd(b,a%b) となる整数 m2, n2 が存在したとすると

```
gcd(a,b) = gcd(b,a%b)
```

であるから

```
b * m2 + (a%b) * n2 = gcd(a,b) ... (1)
```

となる。/ と % の定義

```
a = (a/b)*b + (a%b)
```

を使って(1)を変形すると

```
b * m2 + (a - (a/b)*b) * n2 = gcd(a,b)
a * n2 + (m2 - (a/b)*n2) * b = gcd(a,b)
```

であるから egcd(b,a%b) = (m2, n2, c) が計算できれば

```
m = n2
n = m2 - (a/b)*n2
gcd(a,b) = c
```

として egcd(a,b) = (m,n,c) も求まることになる。a%b < b であるから a <- b, b <- a%b という置き換えを繰り返すと必ず b = 0 となり * に帰着する。したがって整数 a, b に対して m a + n b = gcd(a,b) となる m,n が必ず存在することが具体的な m,n,gcd(a,b) の計算方法を含めて証明できたことになる。

```
def egcd(a,b)
 if b==0
  return [1,0,a]     # a*m+b*n = a*0+0*0 = a = gcd(a,0)
 else
  v  = egcd(b,a%b)
  m2 = v[0]          # m2,n2,c に v[0], v[2], v[2] を
  n2 = v[1]          # わざわざ代入しなくてもよい
  c = v[2]           # ここでは説明の都合上そうしているだけ
  m = n2
  n = m2 - (a/b)*n2
  return [m,n,c]     # a*m+b*n=c=gcd(a,b) となっている
 end
end

a = 1234567
b = 7654321
v = egcd(a,b)
m = v[0]
n = v[1]
c = v[2]
check = a*m+b*n-c    # 0 になるはず
printf("%d*(%d)+%d*(%d)=%d | %d\n", a, m, b, n, c, check)
```