情報セキュリティとリテラシー 2

久木田水生

名古屋大学情報学部

====== 第 5 講 ======

1　繰り返し処理を使ったアルゴリズム

前回の授業では `while` や `for` などを使った繰り返し処理を学びました．今回はそれを応用したアルゴリズムを作ってみましょう．その前に `while` についておさらいをしておきましょう．

1.1　while を用いるプログラム

`while` は次のような形で使います．

```
while xxx
    yyy
end
```

xxx は `true` または `false` を値に取る式です．xxx が `true` である限りインタープリターは yyy を繰り返します．より正確に言うと次のような流れで処理を行います．

1. xxx を評価する．`true` ならば 2 へ進む，`false` ならば終了．
2. yyy を実行する．1 に戻る．

以前の授業で見たように，注意しなければならないのは，`while` は適切に使わないと，処理がループして終了しないことがあるということです．ループした時は，[Ctrl] キーと C を同時に押して，処理を強制終了してください．

ではいくつか `while` を使ったプログラムの例を見ていきましょう．

1.1.1　最大公約数を求めるプログラム

二つの正整数 x と y の最大公約数（x と y を割り切ることのできる最も大きな整数）を求めることを考えてみましょう．まず x と y のうち小さい方を n とします．次に n で x と y の両方が割り切れるかをチェックして，少なくともどちらかが割り切れなければ n の値を 1 減らします．n が x と y の両方を割り切れるまで同じ手続きを繰り返します．

次は 113652 と 18368 という二つの整数の最大公約数を求めるプログラムです．

```
x = 113652
y = 18368
if x > y
    n = y
else
    n = x
end
while (x % n != 0 || y % n != 0)
    n -= 1
end
puts n
```

上のプログラムが行っていることを詳しく説明しましょう．まず `x` と `y` に最大公約数を求めたい二つの整

数を代入します．次に x と y のうちの小さい方の値を n に代入します．x と y の少なくともどちらかが n で割り切れない時は n の値を 1 だけ減らし，それを n が x と y の両方を割り切るまで繰り返します．n が x と y の両方を割り切ったとき，その n の値を出力します．

このプログラムの x と y に代入する値を変えればいろいろな二つの整数の最大公約数を求めることができます．試してみてください．

問題 1.1. (1) 二つの正整数の最小公倍数を求めるプログラムを作りなさい．

(2) 三つの正整数の最大公約数を求めるプログラムを作りなさい．

1.1.2 シラキュース問題

1 より大きい任意の正整数から選んで，偶数ならば 2 で割り，奇数ならば 3 倍して 1 加えるという手続きを繰り返します．どんな正整数でもこの手続きを繰り返せばいずれは 1 になると予想されています（ただし証明はされていない）．この予想が正しいのかどうかという問題はシラキュース問題と呼ばれています．例えば 11 から始めると，次のようなプロセスで 1 に到達します．

$$11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

以下はユーザーが入力した 1 より大きな正整数に対してこの手続きを適用して，上のような数字の並びを表示するプログラムです．

```
puts('Input an integer greater than one.')
x = gets().to_i
str = ''
while x != 1
    str = str + x.to_s + ' -> '
    if x % 2 == 0
        x /= 2
    else
        x = 3 * x + 1
    end
end
str = str + x.to_s
puts(str)
```

問題 1.2. 上のシラキュース問題のプログラムでは，ユーザーが 0 以下の整数を入力した場合には処理がループしてしまう．そこでユーザーが 1 以下の整数を入力した場合には計算を開始せず，「Input an integer greater than one.」と表示して，再び入力を受け取るようにプログラムを改良しなさい．

1.2 for を用いるプログラム

次に for の使い方を見ていきましょう．for は例えば次のような形で使います．

```
for 変数 in 範囲
    xxx
end
```

範囲は例えば「1..10」などと書いて，1 から 10 の間のすべての整数を表します（なお「1...10」というよ

うにピリオドを 3 個書くと，10 は範囲に含まれません）．for の後ろに置かれた変数は in の後で指定された範囲の値の一つ一つを値として取る変数です．範囲の中のそれぞれの値について「for 変数 in 範囲」と「end」に挟まれた部分に記述された処理が繰り返されます．

では for を使って 1 から 100 までの整数の和を求めるプログラムを書きます．

```
sum = 0
for x in 1..100
    sum += x
end
puts sum
```

「1..100」は 1 から 100 までの整数すべてを含む範囲を指します．for の後ろに置かれた変数（上では x）は in の後で指定された範囲の値の一つ一つを値として取る変数です．範囲の中のそれぞれの値について「for 変数 in 範囲」と「end」に挟まれた部分に記述された処理が繰り返されます．したがって sum += x は sum に 1, 2, 3, ..., 100 を順々に足していくことになります．

for 文とともに用いられる範囲は配列でも構いません．したがって each を使った繰り返し処理（第 3 回演習を参照）と同じことを for を使ってもできます．例えば array は文字列からなる配列であるとします．array に含まれる文字列を順番に表示するプログラムは for を使って次のように書けます．

```
for str in array
    puts str
end
```

ちなみにこれは each を使うと次のように書けます．

```
array.each do |str|
    puts str
end
```

問題 1.3. for を使って，100 から 0 までカウントダウンする（100，99，98... を順番に表示する）プログラムを書きなさい．

1.2.1 平方根を求めるニュートン法

for を用いたプログラムの例として，ニュートン法を使って平方根を求めるプログラムを見てみましょう．y の平方根が x であるとき，

$$y = x^2$$

が成り立ちます．したがって

$$y/x = x$$

です．いま x_0 を

$$y < x_0^2$$

となるような正の数だとします．このとき

$$y/x_0 < x_0$$

がなりたち，さらに

$$y/x_0 < y/x = x < x_0$$

が成り立ちます．よって y の平方根は y/x_0 と x_0 の間にあることが分かります．そこで

$$x_1 = (x_0 + y/x_0)/2$$

と置きます．$x_1^2 - y$ を計算すると 0 より大であることが分かるので，

$$y < x_1^2$$

が成り立ちます．このようにして，$y < x_0^2$ となるような適当な x_0 から始めて

$$x_{n+1} = (x_n + y/x_n)/2$$

と置くと，x_n は n の値が大きくなるにつれて y の平方根に近づきます．このようにして平方根の近似を求める方法を**ニュートン法**といいます．

次はニュートン法によって 2 の平方根の近似を x_{10} までを求めるプログラムです．

```
n = 2
x = 0.0
while (x**2 < n)
    x += 1
end
for i in 0..10 do
    x = (x + n/x)/2
end
puts x
```

最初の `n = 2` を変えるといろいろな数の平方根が得られます．

問題 1.4. 上のプログラムで 4 の平方根を求めようとすると，最初のステップで 2 という解が得られているにも関わらず，10 回計算を繰り返す．そこでこのプログラムを改良して，$x^2 - n$ が 0.0001 以下になったら計算を終了するようにしなさい．

2 関数の定義

プログラミングにおいては同じ処理を何度も繰り返すことがあります．そのときその処理に名前を付けて定義しておくと，同じことを何度も繰り返して書く手間が省けます．

例えばファイルに次のように書いて実行してみましょう．

```
def abs(x) # ここから関数 abs の定義．x は入力される値を表す変数
    if x >= 0
        x
    else
        -x
    end
end # abs の定義終わり

puts abs(-5)
puts abs(3)
puts abs(0)
```

abs は入力された値の絶対値を出力する関数です．関数はその定義の中に書かれた最後の式の値を出力としてもちます．上の例の場合は if 文の値が関数の返す出力になります．if 文の値は条件によって分岐している処

理の，実際に実行された部分の最後の式の値です．したがって上の例では x または-x が if 文の返り値，従って関数全体の返り値になります．

上で作った平方根を求めるプログラムを関数として定義すると次のようになります．

```
def newton_method_sqrt(n)
    x = 0.0
    while (x**2 < n)
        x += 1
    end
    for i in 0..10 do
        x = (x + n/x)/2
    end
    x
end
```

二つの整数の最小公倍数を求める関数は次のようにして定義できます．

```
def gcd(x, y)
    if x > y
        n = y
    else
        n = x
    end

    while (x % n != 0 || y % n != 0)
        n -= 1
    end

    n
end
```

2.1 インタラクティブモードで関数を使う

コマンドプロンプトで

```
irb
```

と打ち込むと（Interactive Ruby がインストールされていれば）Ruby のインタラクティブモードが開始されます．これはキーボードから入力された Ruby の式を，Ruby のインタプリタが即時に（Enter キーが押された瞬間に）評価・実行して結果（出力，返り値）を表示してくれるというものです．

コマンドプロンプトで関数が定義してあるファイルの置かれたフォルダ（ディレクトリ）まで移動して，

```
load ’ファイル名’
```

と入力してみましょう．正常にファイルが読み込めたらインタラクティブモードで定義した関数が使えますので，使ってみましょう．例えば上のニュートン法のプログラムを書いたファイルを読み込めば，コマンドプロンプトで

```
newton_method_sqrt(5)
```

と打ち込むことで 5 の平方根を計算することができます．他の数の平方根も試してみてください．また

```
newton_method_sqrt(5)**2
```

などがどのような値になるかを確かめてみてください．

2.2 配列について補足

第三回の演習で配列について扱いましたが，ここで配列を有効に利用するために便利なメソッドをいくつか紹介しておきましょう．以下で array は配列を表す変数とします．

- 配列の長さを求めるメソッド：`array.length`
- 配列の最後に新しい要素を加えるメソッド：`array.push(要素)`
- 配列の最後の要素を削除するメソッド：`array.pop`

  このメソッドは返り値として削除された要素を持ちます．よって例えば

  ```
  array = [0, 1, 2]
  x = array.pop
  ```

  を実行した後では `array` の値は配列 `[0, 1]` となり，`x` の値は `2` になります．

次のプログラムは長さの等しい二つの配列から，要素を互い違いに並べた新しい配列を作る関数です．

```
def interweave(array1, array2)
    new_array = []
    for index in 0...array1.length
        new_array.push(array1[index])
        new_array.push(array2[index])
    end
    new_array
end
```

3 課題

任意の数を与えられたときに，それを各桁の数（0 から 9）からなる配列を生成するプログラムを考えよう．このプログラムは例えば 7403 という数を与えられたときに配列 `[3, 0, 4, 7]` を生成します．これは次のようにして実行できます．まず空の配列を用意します．与えられた数，7403 を 10 で割ります．商は 740，余りは 3．この余り 3 を配列に入れます．次に商として出てきた 740 を 10 で割ります．商は 74，余りは 0．余り 0 を配列に入れます．再び商の 74 を 10 で割ります．商は 7，余りが 4．4 を配列に入れます．7 を 10 で割ります．商は 0，余りは 7．7 を配列に入れます．商に 0 が出たところで手続きを終了します．

この手続きをもう少しプログラムに近い形であらわすと次のようになります．

1. 空の配列を作り，適当な変数名（ここでは `array` とする）を付ける．
2. 変数 `q` を与えられた数で初期化する．
3. `q` を 10 で割った余りを配列 `array` に追加する．
4. `q` を 10 で割った商を新たな `q` の値とする．
5. `q` が 0 になるまで 3-4 を繰り返す．

ヒント：繰り返しの部分で `while` を使います．ループを継続するための条件は `q` が 0 でないこと（0 より大でもよい）です．整数 `x` と `y` に対して，`x` を `y` で割った余りは `x % y` で求めることができます．

4　練習問題

1. 配列を入力として受け取り，その逆順の配列を返す関数を定義しなさい．
2. 数値からなる配列（要素が少なくとも 1 個はある）を受け取り，その配列に含まれる最小の整数を返す関数を定義しなさい．
3. 数値からなる配列と数値を受け取り，その配列からその数値を取り除いた配列を返す関数を定義しなさい．
4. 数値からなる配列（同じ数値の重複はない）を受け取り，数値の小さい順に並べ替えた配列を返す関数を定義しなさい．（このような並べ替えのメソッドは Ruby にあらかじめ用意されているが，それを使わないで自分で考えてみよう）
5. 数値からなる配列（要素が少なくとも 1 個はある）を受け取り，その平均値を求める関数を定義しなさい．
6. 長さが等しい二つの数値からなる配列を受け取り，要素ごとの和の配列を返す関数を定義しなさい．
7. 与えられた正整数（1 より大）が素数であるかどうかを判定する関数（素数であれば true を返し，素数でなければ false を返す）を定義しなさい．
8. 与えられた正整数（1 より大）を素因数に分解する関数（素因数からなる配列を返す）を定義しなさい．この関数は例えば 84 を入力として受け取ると配列 [2, 2, 3, 7] を返す．
9. 与えられた非負整数を二進法表現に変換する関数（二進法で表した文字列を返す）を定義しなさい．この関数は例えば 13 を入力として受け取ると文字列”1101”を返す．（ヒント：上の課題が参考になります．）

練習問題は提出を求めません．よって成績には反映されません．けっこう難しい問題も含まれています．やる気のある人は挑戦してみてください．答えはいずれ公開します．

====== 第 6 講 ======

1　乱数を使う

ゲームを作る時やシミュレーションを行う時などには，ランダムな仕方でコンピュータがアクションを起こす必要がある場合が多々あります．そのようなときには乱数（ランダムな数）を使いましょう．

Ruby では乱数は「rand」というメソッドで生成させます．厳密に言うとこれはランダムであるかのように数を生成するメソッドで，「メルセンヌ・ツイスター」という疑似乱数列生成器を利用しています．

rand は 0 以上の整数を引数に取ります．引数が 0 の場合は 0 以上 1 以下の実数をランダムに返します．引数が 1 以上の場合は 0 以上引数未満の整数をランダムに返します．例えば rand(5) は 0 から 4 までの整数のどれかを返します．

試しに次のプログラムを実行してみてください．

```
for i in 0..10
    puts "rand(#{i}): #{rand(i)}"
end
```

rand は範囲（第 5 回演習を参照）を引数に取ることもできます．例えば rand(5..10) は 5 から 10 までの整数をランダムに生成します．

問題 1.1. 以下の問いに答えなさい．

1. 5 から 10 までの実数をランダムに生成するプログラムを書きなさい．
2. 次はユーザーとインタラクティブにじゃんけんをするプログラムです．???の部分を補って完成させなさい．

```
# 1 から 3 の整数を"グー", "チョキ", "パー"の文字列に変換するメソッド
def hand(x)
    if x == 1
        "グー"
    elsif x == 2
       "チョキ"
    else
        "パー"
    end
end

puts "じゃんけん・・・"
puts "1. グー  2. チョキ  3. パー"

# ユーザーの手を取得
choice = gets().to_i
while choice != 1 && choice != 2 && choice != 3
    puts "1 から 3 の整数を入力してください"
    puts "1. グー  2. チョキ  3. パー"
    choice = gets().to_i
end
puts "ぽん！"
```

```
#コンピュータの手を 1 から 3 の整数としてランダムに生成.
x = ???
puts "コンピュータの手：" + hand(x)
puts "あなたの手：" + hand(choice)
# 勝敗の判定
if ???
    puts "あなたの勝ちです．"
elsif x == choice
    puts "あいこです．"
else
    puts "あなたの負けです．"
end
```

3. 上のじゃんけんプログラムではあいこでもプログラムが終了する．あいこの時はもう一度繰り返す，ただし掛け声は「あいこで・・・」,「しょ！」と表示するように改良しなさい．

1.1 データの偏りを見る

メルセンヌ・ツイスターを利用した乱数生成は，乱数を偏りなく発生させるという特徴を持っています．このことを確かめてみましょう．そのために分散，標準偏差という尺度を利用します．

分散（variance）: 数値の集まりが与えられたとき，全体の平均値から各々の要素がどのくらい離れているかを平均値との差の 2 乗によって表し，それを平均したものが分散と呼ばれます．なぜ 2 乗するかといえば，例えば [10, 10, 20, 20] という数値の集まりと，[10, 10, 15, 25] という数値の集まりでは，どちらも平均は 25，そして平均との差の平均は 5 となり，同じになります．しかし後者の集まりの方が平均から大きく離れている値があるため，ばらつきは大きいと考えたほうがよいのです．そのため平均値から遠く離れている値ほど分散に大きく貢献するように，平均との差を 2 乗します．

標準偏差（standard deviation）: 分散は平均値との差を 2 乗しているため，もとの数値と次元が異なり，比較しにくくなります．そこで分散の平方根が偏りの尺度としてよく使われます．これを標準偏差といいます．

前回の演習で平方根を求めるプログラムを作りましたが，Ruby にはあらかじめ平方根を求めるためのメソッド `Math.sqrt` が用意されています．

```
Math.sqrt(n)
```

によって n の平方根を求めることができます．`Math` は数値計算のための定数やメソッドが色々定義されたモジュールです．モジュールについては後述します．

問題 1.2. 以下の問いに答えなさい．

1. 次のプログラムは分散と標準偏差をもとめる関数を定義したものです．???の部分を補ってプログラムを完成させなさい．

```
# 配列の合計を求める関数
def sum(array)
    s = 0.0
    ???
    s
end

# 配列の平均値を求める関数（上の sum を利用する）
```

```
def mean(array)
    ???
end

# 配列の分散を求める関数
def variance(array)
    m = mean(array)
    diff_array = []
    ???
    mean(diff_array)
end

# 配列の標準偏差を求める関数
def standard_deviation(array)
    Math.sqrt(variance(array))
end
```

2. 0 以上 1000 未満のランダムな整数，500 個からなる配列を生成し，その標準偏差を求めなさい．またその値を完全に偏りのない配列（すなわち 0 から 999 までのすべての整数が一個ずつ含まれた配列）の標準偏差と比較しなさい．

1.2 ランダムな交換

この教室の中の一人一人に 100 円ずつ与えられたとします．各人はランダムに相手を選んで自分の手持ちのお金から 1 円渡すとします．これを繰り返していくと各自の所持金はどのように分布すると思いますか？これをシミュレーションで確かめてみたいと思います．次のような手続きで行います．

1. 長さ 50，各要素が 100 であるような配列を作る．
2. 各要素について，それが 0 でなければそこから 1 を引き，ランダムに選んだ別の要素に 1 を足す．
3. 2 を一定回数繰り返したら配列を小さい順に並べ替える．
4. 出力として配列を返す．

配列を並べ替えるには `sort!`というメソッドを使います．`array` が数値からなる配列であるとき，

```
array.sort!
```

を実行すると `array` は小さい順に並べ替えられます．

問題 1.3. 上の手続きを実行するプログラムを書き，ランダムな交換によって所持金の分布がどのようになるかを確かめなさい．その際，配列の長さと繰り返しの回数を入力として受け取る関数として定義しなさい．また繰り返しの回数を変えて，所持金の分布について，繰り返しの回数を変えた時に，標準偏差がどのように変化するかを確かめなさい．

1.3 モンテカルロ法による円周率の近似

モンテカルロ法は乱数によってシミュレーションを行う手法の総称です．ここではモンテカルロ法を使って，円周率を近似してみましょう．

円周率 π は，半径 1 の円の面積です．従って下図の正方形の面積が 1 であるのに対して，グレーの 4 分の 1 円の面積は $\pi/4$ です．従って下図の正方形の中にランダムに点を打った時，それがグレーの領域の中にある確率は $\pi/4$ です．

逆に言うとこの正方形の中にランダムに N 個の点を打った時，灰色の領域に入っている点が n 個だったとすると，$4(n/N)$ の値は π に近似しているはずだということになります．N を大きくすればするほど近似は正確になっていくでしょう．

この方法を次のような手続きで実装します．

1. 二つの変数 `interior` と `total` を 0.0 で初期化する
2. 0 以上 1 未満の実数を二つランダムに生成し，変数 `x` と `y` の値として代入する．
3. 点 (`x`, `y`) と原点 (0, 0) との距離（`x` の 2 乗と `y` の 2 乗の和の平方根）が 1 未満だったときは変数 `interior` および `total` の値を 1 増やし，そうでないときは変数 `total` の値だけを 1 増やす[*1]．
4. 2-3 を一定の回数を繰り返す．
5. `interior` を `total` で割り，それに 4 をかけた値を返す．

問題 1.4. 上の手続きをプログラムで書きなさい．反復の回数を色々変えて結果を確認しなさい．

2　プログラムのモジュール化

前節で平方根を求める Math.sqrt というメソッドを紹介しました．例えば

```
Math.sqrt(2)
```

を実行すると，1.4142135623730951 という値が返されます．しかし `sqrt(2)` と書いても「そのようなメソッドは定義されていない」という趣旨の（英語の）メッセージが表示されて，エラーになります．

平方根を求めるプログラムは `Math` という「モジュール」の中で定義されています．Ruby におけるモジュールとは，特定の目的のために使われるメソッドや定数をひとまとまりにしたものです．モジュールで定義されたメソッドをプログラムの中で使うには

[*1] 実際はある正の実数 a の平方根が 1 未満かどうかという問題は，a 自体が 1 未満かどうかという問題に帰着するので，`x` の 2 乗と `y` の 2 乗の和が 1 未満であるかどうかをチェックすれば良い．

Module.method

というように書きます．Module がモジュールの名前，method がメソッドの名前です．

モジュールの中で定義された定数を呼び出すには

Module::CONSTANT

というようにモジュール名の後ろにコロンを二つ，そのあとに定数名（Ruby では定数名を大文字で書くのが一般的）を書きます．

プログラムを作るときには，異なる種類の機能は別の部分に分割して，使うときに呼び出して組み合わせるというやり方が推奨されます．これをプログラムのカプセル化あるいはモジュール化といいます．モジュール化の利点は以下のようなものがあります．

- よく使う機能をモジュールにまとめて置くと，モジュールを再利用することで一つ一つのプログラムの中に書かなくて済む．
- ある機能の実装を変更したいときに，他の部分に影響を与えない．また一つのモジュールの中だけ変更すれば済む．
- 一度に一つの機能に集中して作業ができ，プログラムの他の部分を気にしなくてよい．
- プログラム全体の構造が把握しやすい．

モジュール化の発想は，プログラミングに限らず，工学全般においても（さらに言えば数学や哲学などの抽象的な概念を扱う学問においても）重要です．

プログラミングにおけるモジュール化の手法としてプログラムを複数のファイルに分ける（そして別のファイルに読み込む）という直接的な方法の他，前回扱った関数の定義，モジュールの作成，クラスの作成（これがいわゆる「オブジェクト指向」のアイディアの核になります），ライブラリの作成などがあります．ただしこの授業では関数の定義以外は扱いません．

Math モジュールにはメソッドとして三角関数（sin, cos, tan など），逆三角関数（asin, acos, atan など），対数関数（log, log2, log10），指数関数（exp）など，定数として円周率（PI），ネイピア数（E）が定義されています．

問題 2.1. 以下の問題に答えなさい．

1. 上で作成したモンテカルロ法による円周率近似プログラムの結果を Math::PI の結果と比較しなさい．
2. Math.log(x, b) は b を底とした x の対数を返す．b を省略した場合は自然対数（ネイピア数 e を底とした対数）を返す．Math.log(Math.exp(x)) が x に等しくなることを確かめなさい．（ただし x の値が大きすぎると計算できない）

3 ファイルの操作

Ruby で，コンピュータの中にあるファイルを操作する方法について少しだけ説明しましょう．

Ruby でファイルを扱うためには，File というクラスのオブジェクトを作ります．クラスやオブジェクトについて説明するにはオブジェクト指向について説明しないといけませんが，ここでは割愛します．

まず実際にコードを書いて試してみましょう．次のように書いたプログラムを実行してください．\は環境

によっては ¥ になっていると思います．

```
File.open('example.txt', 'w') do |file|
 file.write("Ruby\nJava\nPython\nLisp")
end
```

すると同じフォルダに example.txt という名前のファイルが作成されると思います．開いてみると中には

Ruby
Java
Python
Lisp

書かれています．このプログラムが行っている処理について説明しましょう．

- `File.open`
  `File` というクラスに対して定義されている `open` というメソッドを実行しています．このメソッドは指定されたファイルを開き，そのファイルに基づいて File クラスのオブジェクトを生成します[*2]．
- `('example.txt', 'w')`
  `File.open` メソッドに対する入力[*3]です．なおこの括弧は省略できますこれまでに使ったメソッド，例えば `puts` に対する引数も，本来は括弧を付けて表すところを，省略して表記しています．一番目の引数`'example.txt'` はファイルへのパスを指定します．ファイル名だけが指定されている場合はプログラムファイルが置かれているのと同じフォルダを参照します．
  二番目の引数`'w'` は，ファイルを開くモードを指定します．モードには次のようなものがあります．
  - `'r'`：読み取り専用モード．ファイルの中身を読み取ることはできますが，ファイルを変更することはできません．
  - `'w'`：書き込み専用モード．`'r'` の逆です．開かれたファイルは，このモードで開いた時点で中身が消去されます．ファイルが存在していない場合は新規に作成されます．
  - `'a'`：追記専用モード．`'w'` とは違って元のファイルの中身を維持した上で，その後ろから新しく書き込みを行います．
  - `'r+'`：読み込みと書き込みの両方ができるモードです．
  - `'w+'`：上と同様ですが，ファイルを開いた時点でファイルの中身は消去されます．
  - `'a+'`：読み込みと書き込みの両方ができますが，元のファイルの中身は維持され，書き込み開始位置は元のファイルの最後になります．
- `do...end`
  `File.open` メソッドで生成されたファイルオブジェクトに `do` と `end` で囲まれた部分（これを Ruby ではブロックと呼びます）に記述された処理を施します．`|file|`は渡されたファイルオブジェクトを指示するための変数を指定しています．なおこのブロックが終了すると，自動的に開かれたファイルが閉じられます．ファイルを開きっぱなしにするとメモリリーク[*4]が生じます．

[*2] クラスとは，簡単に言えば，プログラミング言語で扱う対象の種類のことです．それぞれのクラスには固有の属性や処理が定義されています．例えば Array クラスには長さなどの属性があり，また配列同士をつなげるといった処理が定義されています．

[*3] 数学では関数に対する入力を「引数 argument」と呼びますが，プログラミングでも同じ言葉を使います．

[*4] メモリはコンピュータの中で処理を行う “場所” のようなものです．一つ一つの処理に一定のメモリが割り当てられるので，ファイルを開きっぱなしにしておくと無駄にメモリを使用することになります．これをメモリリークと言います．

- `file.write("Ruby\nJava\nPython\nLisp")`
  File クラスオブジェクトに対しては，読み込みや書き込みなどの処理が定義されていますが，ここでは書き込みが行われており，`file` という変数で名指されている File オブジェクトに「Ruby\nJava\nPython\nList」という文字列が書き込まれています．`\n` は改行記号を表します．従ってテキストファイルを開くと`\n` が書かれている箇所で改行されている訳です．ただし `write` の後ろの括弧の中でシングルクオート (') を使った場合には`\n` は改行マークとして認識されず，その通りの文字列として書き込まれます．

次にファイルを読み込むプログラムの例を見てみましょう．作成された exmaple.txt はそのままにしておいて，次のプログラムを実行してみてください．

```
File.open('example.txt', 'r') do |file|
 puts file.read
end
```

`'r'` は読み込みモードでファイルを開くことを指定しています．このプログラムを実行すると example.txt の中身が表示されたと思います．`file.read` は `file` で指示されたファイルオブジェクトの中身を読み込んで文字列として返しています．指定された名前のファイルが存在しないときはエラーが返されます．

最後に，ファイルから一行ごとに文字列を読み込む方法を紹介します．

```
languages = []
File.open('example.txt') do |file|
 file.each_line do |line|
 languages.push(line.chomp)
 end
end
p languages
```

このプログラムは，ファイルのオブジェクトに対して定義された `each_line` というメソッドを使って，読み込んだファイルから一行ずつ文字列を取り出して配列を作っています．`chomp` というメソッドは文字列から最後の改行マークを取り除きます．

問題 3.1. 指定された名前のテキストファイルを読み込み，各行をランダムに並べ替えた別なテキストファイルを作成する関数を定義しなさい．引数は読み込むファイルの名前と作成するファイルの名前の二つを取るものとする．

4 課題

モンテカルロ法によって π の近似を実行するプログラムを書きなさい。

====== 第 7 講 ======

1　関数の再帰的定義

数学で用いられる階乗（factorial）という関数は，任意の非負整数 n に対して次のように定義されます．

$$f(n) = \begin{cases} 1 & (n = 0) \\ n \cdot f(n-1) & (n > 0) \end{cases}$$

この関数は n が 0 の時には 1 であると定義され，0 より大のときには x と $f(x-1)$ の積として定義されています．例えば n が 5 のとき，$f(n)$ の値は

$$\begin{aligned} f(5) &= 5 * f(4) \\ &= 5 * (4 * f(3)) \\ &= 5 * (4 * (3 * f(2))) \\ &= 5 * (4 * (3 * (2 * f(1)))) \\ &= 5 * (4 * (3 * (2 * (1 * f(0))))) \\ &= 5 * (4 * (3 * (2 * (1 * 1)))) \\ &= 120 \end{aligned}$$

のように求めることができます．

この定義においては，定義式の中に，定義されている関数それ自体が現れています．これは一見，循環的な定義に見えますが，しかし正常な入力（非負整数）であれば必ず正常な値を返す，well-defined な関数です．このような関数の定義を再帰的定義と呼びます．

プログラミングにおいても関数を再帰的に定義することができます．次は階乗の再帰的な定義です．

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

上で定義した関数に引数として-1 を入れて実行すると，処理が終了しなくなります．試してみてください．（強制終了は [Ctrl] + C です．）

再帰的定義は複雑な手続きを非常にシンプルに記述できる便利なものですが，気を付けないと処理がループしてしまう可能性があります．この点で再帰的関数と while 構文には共通点があります．実は一般的に while を使って書いたプログラムは再帰的関数を使って書き換えることができ，またその逆も成り立つということが数学的に証明できます（参考：高橋正子『計算論』，定理 1.4.2 および定理 1.4.6，近代科学社，1991）．

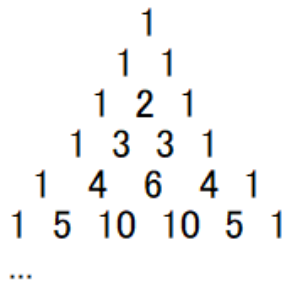
例えば階乗を計算する関数は次のように再帰を使わずに定義できます．

```
def factorial(n)
    if n == 0
        1
    else
```

```
        n * factorial(n - 1)
    end
end
```

問題 1.1. 以下の問いに答えなさい。

1. 最大公約数を求める手続きとして，ユークリッドの互除法と呼ばれるアルゴリズムがある．これは正整数 x, y が与えられたとき，x を y で割った余りを r とすると，x と y の最大公約数は y と r の最大公約数に等しいという性質を利用している．具体的には次のような手順である．
   (a) x を y で割った商を q，余りを r とする．
   (b) q が 0 ならば y を出力する．そうでなければ x に y の値を代入，y に r の値を代入し，(a) に戻る．
   この手続きを再帰的関数として実装しなさい．
2. ユークリッドの互除法を，再帰を使わず while を用いて実装しなさい．
3. 次の数の配列はパスカルの三角形と呼ばれるものである．



   n 段目の列は n 個の数からなり，その 1 番目と n 番目は 1 で，それ以外の数はそれがその列の m 番目とすると，$n-1$ 列目の $m-1$ 個目の数と m 個目の数の輪になっている．正整数 n を引数に取り，パスカルの三角形の n 列目の数の配列を出力する関数を再帰的に定義しなさい．例えばこの関数は 5 という引数に対して配列 `[1, 4, 6, 4, 1]` を出力する．

1.1 フィボナッチ数列

フィボナッチ数列は数学的に興味深い性質を持つ数列で，その最初のいくつかの項を並べると次のようになります．

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

この数列は最初の二つは 1，そして三つ目以降は先行する二つの項の和になっているという性質を持っています．したがってその定義は次のようになります．

$$f(n) = \begin{cases} 1 & (n = 0, 1) \\ f(n-1) + f(n-2) & (n > 1) \end{cases}$$

これをそのままプログラムで書くと次のようになります．

```
def fibonacci(n)
    if n == 0 || n == 1
        1
```

```
    else
        fibonacci(n-2) + fibonacci(n-1)
    end
end
```

これはシンプルでわかりやすい定義ですが，そのままでは問題があります．というのは fibonacci の定義の中で fibonacci を 2 回呼び出しているため，n が増加するごとに呼び出し回数が倍々で増え，計算時間が指数関数的に増大します．試しに上で定義した fibonacci に 40 という引数を与えてみてください．結構な時間がかかると思います．

再帰はプログラミングをシンプルにするという利点がありますが，このように計算時間を増やしてしまうという危険性もあります．これに対してとれる対策はいくつかあります．一つは再帰呼び出しをせずに，while で小さな項から数え上げていく方法です．このプログラムは練習問題とします．

もう一つは「メモリ」を使う方法です．上の fibonacci の定義では再帰呼び出しによって同じ引数に対して何度も同じ計算をすることになります．例えば fibonacci(5) の計算をするときには fibonacci(3) は 2 回，fibonacci(2) は 4 回，fibonacci(1) は 8 回計算させられます．そこで一度計算した値については配列に記録しておき，そこにアクセスすることで後から同じ計算をする手間を省くことができます．具体的には次のようなプログラムで実装できます．これだと n の増加に対して計算時間は線形にしか増加しません．

```
def fibonacci_with_memory(n, memory)
    if memory[n] == nil
        memory[n] = fibonacci_with_memory(n - 2, memory)
                        + fibonacci_with_memory(n - 1, memory)
    end
    memory[n]
end

def effective_fibonacci(n)
    memory = [1, 1]
    index = 2
    while index <= n
        memory.push(nil)
        index += 1
    end
    fibonacci_with_memory(n, memory)
end
```

1.2 プログラムの処理速度を計測する

フィボナッチ数列を計算する二つのプログラムが実際にどれだけ時間がかかっているかを計測してみましょう．そのためにプログラムの実行開始時の時刻と終了時の時刻を記録しておき，その差を求めます．

Ruby には時刻を表現するオブジェクトのクラスが用意されています．これは「Time」という名前が付けられています．Time クラスを使ってみましょう．

```
Time.now
```

で現在の時刻が取得できます．

```
puts Time.now
```

を実行してみると現在の時刻が出力されます．例えば

2017-07-31 07:33:12 +0900

などと表示されるでしょう．これは現在の時刻が 2017 年 7 月 31 日の 7 時 33 分 12 秒であり，GMT（グリニッジ標準時刻）に 9 時間先立っているということを表しています．

Time クラスのオブジェクトを使うと，次のように書くことで処理の時間を計測することができます．

```
# 開始時の時刻を取得する
start = Time.now

# 時間を計測したい処理をここに書く
fibonacci(35)

# 終了時の時刻を取得する
finish = Time.now

# 終了時の時刻と開始時の時刻の差を計算する
time_taken = finish - start

puts "この処理には#{time_taken}秒かかりました．"
```

ここでの「-」（マイナス記号）は Time クラスのオブジェクトの間に定義されたメソッドで，二つの時刻の間の差を秒数（実数）で返します．

問題 1.2. 以下の問いに答えなさい。

1. 再帰を使わずフィボナッチ数列を生成するプログラムを書きなさい．
2. フィボナッチ数列を生成するプログラムの三種類（再帰を使いメモリを使わないプログラム，再帰を使わないプログラム，再帰とメモリーを使うプログラム）の処理速度を比較しなさい．

1.3 クイックソート

再帰の非常に有効な使用例としてクイックソートがあります．これは 1960 年に計算機科学者のチャールズ・アントニー・リチャード・ホーアによって考案された並べ替えのアルゴリズムで，特に大量のデータを並べ替える際に効率的な方法です．ざっくりと説明すると次のようなものです．データの配列の中からランダムにいくつかのサンプルを選び出し，それらの中央値を計算します．この値をピボットと呼びます．次にデータの全体を，(1) ピボットより小さい値の配列，(2) ピボットに等しい値の配列，(3) ピボットより大きい値の配列に分けます．(1)(3) に対しては同じ処理を繰り返します（ここで再帰を使います）．配列の長さが 1 または 0 になったときには処理を終了します．(1)(2)(3) を結合すると，小さい順に並べ替えることができます．

クイックソートの実装は次のようになります．???の部分を補ってプログラムを完成させましょう．

```
# 3 つの値の中央値を求める
def median(x, y, z)
 if x > y
  if y > z
   y
  elsif x > z
   z
```

```
  else
   x
  end
 elsif z < x
  x
 elsif z < y
  z
 else
  y
 end
end


# ピボットを求める．配列の長さが 10 より大きい時は，
# 3 つの要素をランダムに選んでその中央値を返す．
# それ以外はランダムに選んだ 1 つの要素を返す．
def select_pivot(array)
 n = array.length
 if n > 10
  x = array[rand(n)]
  y = array[rand(n)]
  z = array[rand(n)]
  median(x, y, z)
 else
  array[rand(n)]
 end
end


def quick_sort(array)
 if array.length <= 1
  new_array = []
  array.each do |item|
   new_array.push(item)
  end
  new_array
 else
  smaller = []
  larger = []
  pivots = []
  pivot = select_pivot(array)
  index = 0
  len = array.length
  while index < len
   # 配列の要素をピボットを基準に仕分ける
   ???
   index += 1
  end
  # 仕分けた配列に再帰的にこの関数を適用し，
  # その結果として得られる配列を合わせて返す．
  # 配列を合わせるには「+」を使う．
  ???
 end
end
```

問題 1.3. 以下の問いに答えなさい。

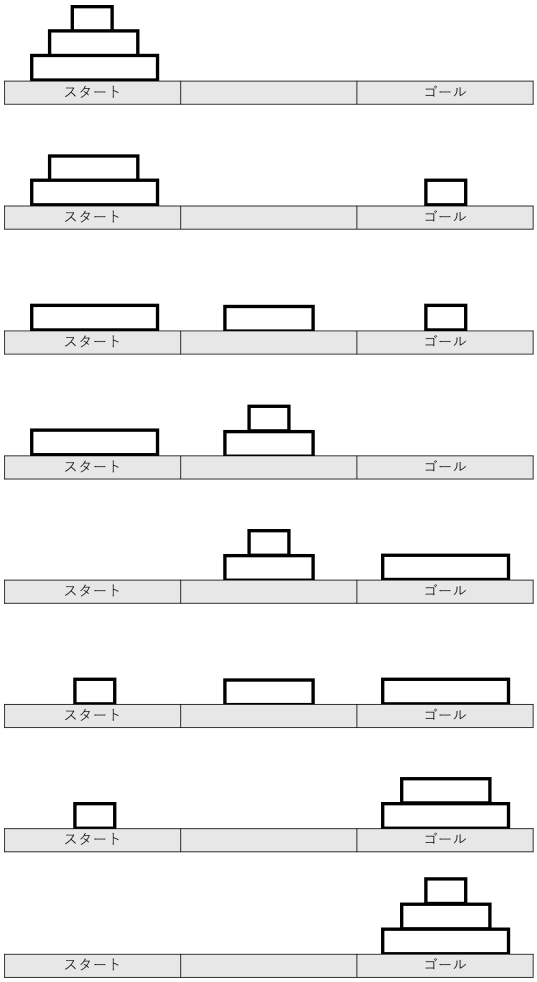
1. ランダムに選んだ 1000 未満の非負整数からなる配列を生成し，クイックソートによる並べ替えにかかる時間を測定しなさい．要素の数を 10000，20000，30000 と増やして時間がどのように増大するか調べなさい．
2. 以下のように定義される slow_sort と上のクイックソートの速さを比べなさい．

```
# 配列の最小値のインデックスを返す
def min_index(array)
    min_ind = 0
    m = array[0]
    for index in 0...array.length
        if array[index] < m
            m = array[index]
            min_ind = index
        end
    end
    min_ind
end

def copy(array)
    new_array = []
    array.each do |item|
        new_array.push(item)
    end
    new_array
end

def slow_sort(array)
    copied_array = copy(array)
    new_array = []
    while copied_array != []
        new_array.push(copied_array.delete_at(min_index(copied_array)))
    end
    new_array
end
```

ただしこのプログラムの中で使われている「delete_at」というメソッドは，配列に適用され，指定したインデックスの要素を返すとともに，その配列からそのインデックスの要素を削除するメソッドである．

1.4 ハノイの塔

ハノイの塔という大きさの異なる複数の円盤を一定のルールに従って移動させるゲームがあります．円盤を置ける場所は三か所あり，最初はすべての円盤が大きいものから順に一か所に重ねてあります．これを別の一か所にすべて移動させればゲームは終了です．ただし円盤を移動させるときは次のルールに従わなければなりません．(1) 一度に移動できる円盤は一枚だけ．(2) 移動させる先に移動する円盤より小さい円盤があってはいけない．例えば，三個の円盤を使う場合，下図のように移動させます．



ハノイの塔を解くプログラムを考えてみましょう．使用する円盤が n 個だとして，小さい円盤から順に $D_1, D_2, \ldots, D_n$ と呼ぶことにします．ハノイの塔を解くには，必ず円盤 D_1 から D_{n-1} がスタートでもゴールでもない場所（上の図なら真ん中．以後，この場所をバッファと呼びます）に置かれている状態で，D_n をスタート地点からゴール地点に移すというプロセスがなければなりません．

$n-1$ 個の円盤を使うハノイの塔が解ければ，その方法を応用して次のように n 個の円盤をつかうハノイの塔を解くことができます．

1. D_1 から D_{n-1} をスタートからバッファに移動する．
2. D_n をスタートからゴールに移動する．
3. D_1 から D_{n-1} をバッファからゴールに移動する．

このアイディアをプログラムで実装してみましょう．

まずハノイの塔のゲームの状態をコンピュータの中で表現する方法を考えます．ハノイの塔では三つの場所に大きさの異なる円盤が置かれています．そこで円盤が n 個使われている時，それぞれの円盤を 1 から n までの整数で表すことにします．円盤が置かれる場所のそれぞれは整数の配列で表します．それらを三つ集めて，配列の配列を作ります．これでハノイの塔のゲームの状態を表現することにします．上の図で示した三つの円盤を使うハノイの塔の場合は，ゲーム開始時の状態（初期状態）は次の配列で表現されます．

```
[[3, 2, 1], [], []]
```

ゲーム終了時の状態（停止状態）は次の配列です．

```
[[], [], [3, 2, 1]]
```

ゲーム全体の状態の推移は次のようになります．

```
[[3, 2, 1], [], []]
[[3, 2], [], [1]]
[[3], [2], [1]]
[[3], [2, 1], []]
[[], [2, 1], [3]]
[[1], [2], [3]]
[[1], [], [3, 2]]
[[], [], [3, 2, 1]]
```

一度の円盤の移動は，移動元の配列の最後の要素を取り出し，移動先の配列の最後にその要素を付け加えるという操作になっていることが分かります．この操作は pop と push を使って，次のように定義できます．

```
def move(towers, from, to)
    towers[to].push(towers[from].pop)
end
```

変数 towers は，円盤の置かれる三つの場所を表します（つまり 3 個の配列からなる配列です）．変数 from と to は 0 から 2 の整数で，それぞれ円盤の移動元，移動先を表します．メソッド pop は配列の最後の要素を返すと同時にその配列から最後の要素を削除して，配列を変化させるということに注意しましょう．例えば towers が

```
[[1], [], [3, 2]]
```

という状態であるとき，

```
move(towers, 0, 2)
```

を実行すると，towers は

```
[[], [], [3, 2, 1]]
```

という状態になります．

次にある場所から別の場所に n 個の円盤を移動する操作を実装します．$n = 1$ のときは単にいま定義した move を使うだけです．n が 1 より大の時は上で示した 1-3 の手順に従います．少し手間なのが，例えば移動元と移動先を指定されたとき，残った一か所をバッファとして使うので，移動元と移動先がどこであるかという情報からバッファがどこであるかを求める必要があるということです．これを次の関数によって行います．

```
def buffer(from, to)
    for i in 0..2 do
       if (i != from && i != to)
           return i
```

```
        end
    end
end
```

この中で使われている return は関数の返り値を指定するメソッドで，繰り返し処理の途中でも return が実行されたときはその処理が中断されます．

次に定義する関数がハノイの塔の解法のキモになる部分で，else 以下の部分が上記の 1-3 で記述されている手順の実装です．

```
def move_n_discs(towers, n, from, to)
    if n == 1
        move(towers, from, to)
    else
        buffer = buffer(from, to)
        move_n_discs(towers, n - 1, from, buffer)
        move(towers, from, to)
        move_n_discs(towers, n - 1, buffer, to)
    end
end
```

最後に初期状態の設定とゲームの開始のための関数を定義します．

```
def initial_state(n)
    towers = [[],[],[]]
    for i in 1..n
        towers[0].push(n - (i - 1))
    end
    towers
end

def tower_of_hanoi(n)
    towers = initial_state(n)
    print towers
    puts
    move_n_discs(towers, n, 0, 2)
    print towers
end
```

ハノイの塔の解法の実装は以上です．ややこしそうな手続きが再帰を使ってエレガントに記述できることが実感できるでしょう．

ところで上の定義では初期状態と停止状態は表示されますが，途中の経過がどうなっているかがわかりません．そこで移動の回数を記録し，また移動の回数と途中の状態も表示するようにプログラムを改良しましょう．

```
def move_n_discs2(towers, n, from, to, iteration)
    if n == 1
        move(towers, from, to)
        iteration += 1
        print iteration.to_s + ": "
        print towers
        puts
    else
        buffer = buffer(from, to)
        iteration = move_n_discs2(towers, n - 1, from, buffer, iteration) + 1
```

```
            move(towers, from, to)
            print iteration.to_s + ": "
            print towers
            puts
            iteration = move_n_discs2(towers, n - 1, buffer, to, iteration)
        end
        iteration
    end

    def tower_of_hanoi2(n)
     towers = initial_state(n)
     print "0: "
     print towers
     puts
     move_n_discs2(towers, n, 0, 2, 0)
    end
```

引数の値をいろいろに変えて，tower_of_hanoi2 を実行してみてください．

ハノイの塔は円盤の数が増えれば移動の回数も増えるということはすぐにわかりますが，具体的にはどのくらいの回数が必要なのでしょうか．円盤の数が n 個の時に必要な移動回数を H_n によって表すと，$H_1 = 1$ は自明です．n が 1 より大きい時は D_1 から D_{n-1} をスタートからバッファに移動，D_n をスタートからゴールに移動，D_1 から D_{n-1} をバッファからゴールに移動するので $2H_{n-1} + 1$ 回の移動が必要となることが分かります．つまり数列 H_n の漸化式は

$$H_1 = 1$$
$$H_n = 2H_{n-1} + 1$$

一般項は

$$H_n = 2^n - 1$$

になります．したがってハノイの塔を解くために必要な移動の回数は円盤の数に対して指数関数的に増大することが分かります．

1.5 課題

正整数 n を入力として受け取り、パスカルの三角形の n 段目の配列を生成するプログラムを再帰を使って書きなさい。

====== 第 8 講 ======

1　タイピング練習プログラムを作ろう

これまでに学んだことを活かして，タイピング練習プログラムを作りましょう．これは次のような機能を持ちます．

- 出題される単語のリストを書いたテキストファイルを開き，単語の配列を生成する．ファイルの操作については第 6 回演習用のテキストを参照．
- 文字列の配列からランダムに単語を選び出して出題する．乱数については第 6 回演習用のテキストを参照．同じ単語を重複して出題しないように，配列からその単語を削除しておく．これには delete_at というメソッドを使う．例えば array.delete_at(n) は配列 array の n 番目の要素を返すと同時に，その要素を削除する．$n+1$ 番目以降の要素は一つ前に移動する．
- ユーザーがキーボードで入力した文字列を受け取り，出題された単語と比較する．同じだったら正解，異なっていたら不正解として記録する．
- 2，3 を一定の回数繰り返す．
- 正解・不正解それぞれの数，間違った単語，かかった時間を表示する．時間を測る方法については第 7 回のテキストを参照．

問題 1.1. 下記のプログラムの???の部分を補ってタイピング練習のプログラムを完成させなさい。

```
word_list = []

# 別のファイルにある単語リストを読み込んで、配列にする。ここから出題する。
File.open('word_list.txt') do |file|
 file.each_line do |line|
 ??? #word_list に単語を一行ずつ追加する。
 end
end

error_list = [] # 間違った単語をここに入れて置く。
error_count = 0 # 間違った数をカウントする。
letter_count = 0 # 総文字数をカウントする。

puts "タイピングの練習をしましょう．画面に表示される文字列をキーボードから入力してください．"
puts "文字列はすべてひらがなです．全部で 10 問出されます．"
puts "Enter キーを押すとスタートします．"
gets
start = ??? #開始した時間を記録。

for i in 1..10
    # 単語のリストからランダムに一個の単語を選び出し word に代入する。
    # その単語はリストから削除する。
 word = ???
 puts "                    #{word}"
 typed = ??? # キーボードからの入力を受け取る。
 letter_count += typed.length
```

```
 if word != typed # タイプ間違いの場合。
 error_list.push(word)
 puts "間違いです. "
 error_count += 1
 else
 puts "正解です. "
 end
 puts
end

finish = ??? # 終了時間を記録。

if error_count == 0
 puts "パーフェクトです！"
else
 puts "間違った問題は#{???}個でした. "
end

time_taken = ??? # かかった時間を計算。
time_per_letter = ??? # 一文字あたりにかかった時間を計算。
letters_per_minute = ??? # 一分あたりにタイプできる文字数を計算。
puts "かかった時間は#{???}秒でした. "
puts "一文字あたりにかかった時間はおよそ#{???}秒です. "
puts "一分間におよそ#{(60/time_per_letter).round}文字, タイプできます. "

if error_count > 0
 puts "間違った問題は次の通りです. "
 puts error_list
end
```