# ショートプログラムでパワーアップ — Ruby 編

2009 年 3 月 4 日

# 第 1 章

# いろいろな取り決め

## 1.1 Ruby プログラムの実行

Ruby のプログラムを hogehoge.rb としておきましょう。これを走らせる基本的な方法は，シェルのコマンドラインから次のように入力することです。

```
> ruby hogehoge.rb
```

上の先頭の > はシェルのプロンプトですから入力しないでください。

毎回 ruby と打つのは面倒だと思ったら，プログラムの先頭に次の行を入れて，実行プログラムである ruby のパスを指定してやります。

```
#!/usr/local/bin/ruby
```

こうすると，hogehoge.rb は次のように単体で実行できるようになります。

```
> hogehoge.rb
```

ただし，Cygwin などでは /usr/bin/ ruby のパスが /usr/local/bin/ruby ではなく，/usr/bin/ruby になっていることもあるので，それに合わせます。 ruby のパスがどうなっているかを知るには，which コマンドを使います。

## 1.2 重要な用語

### 1.2.1 一般的なもの

**リテラル** プログラム中に記述できる数値や文字列などのデータのこと。数値リテラル，文字列リテラル，正規表現リテラルなどといった使い方をします。

**定数** MyName = ’えみちゃん’ の MyName のように最初が英字の大文字で始まる識別子は定数です。一度定義された定数に対する代入はできません。

**変数**　変数にはさまざまな値を自由に代入できます。Ruby の変数にはスコープの異なるいくつかの種類があります。

**ローカル変数**　cx1, _foo のように英字の小文字またはアンダースコア _ で始まる識別子はローカル変数。それが定義されたメソッド，ブロック内にスコープをもっています。

**グローバル変数**　$scale のようにドル記号 $ 1 つが先頭にある変数はプログラム全体にスコープを持ちます。グローバル変数の多用は致命的なバグの原因になりやすいので，使用は極力避けてください。

**メソッド**　オブジェクトに働きかけて何らかの仕事をするもの。下の例では配列 [1,7,5,2] というオブジェクトにメソッド max が働きかけて，最大の要素という情報を受け取っています。

```
puts [1,7,5,2].max  #=> 7
```

**レシーバ**　メソッドからの働きかけを受ける側のこと。上の例では，.max の前にある配列がレシーバになっています。

### 1.2.2　オブジェクト指向にかかわる用語

**オブジェクト**　「もの」

**クラス**　オブジェクト（つまりプログラムで扱う「もの」）の分類を定めているのがクラス。Ruby ではすべてのものが何らかのクラスに属しています。しょっちゅう使っているクラスとしては，String, Array, Fixnum, Float, Regexp, Hash などがあります。

**モジュール**　クラスとよく似ていますが，インスタンスを持たない。Math はよく使われるモジュール。

**インスタンス**　あるクラスに属するオブジェクト。

**継承**　上位のクラスから下位のクラスに性質を受け継ぐこと。

**スーパークラス**　上位のクラス。動物は哺乳類のスーパークラスであり，哺乳類はウシのスーパークラス。

**クラス変数**　先頭が '@@' で始まる変数はクラス変数。そのクラスに属するインスタンス全体で共有されます。あまり使われません。

**インスタンス変数**　先頭が '@' 1 個で始まる変数はインスタンス変数。あるクラスに属する複数のインスタンスがそれぞれ独自の値を持つことができます。

## 1.3　よい名前の付け方

プログラミング言語には，**予約語**というあらかじめ決められた語彙があります。これらは言語仕様に含まれていたもので変更不可能です。一方，それ以外の変数，定数，メソッド，クラスの名称は，プログラムを書く人が一定に制約内で自分の好みに従って付けることができます。

しかし，プログラムのソースは他の開発者と共有されるものと考えるべきであり，慣例に従った書き方をすることが望ましいのです。そのような慣例を**コーディング規約 (coding convention)** といいます。

### 1.3.1　名前の付け方の原則

**原則として英単語で**　しばしば変数名に `heikin`, `kakaku` などというローマ字を使った入門書がありますが，ソースプログラムは国際的に共有されるものという意識をもって英語を使いましょう。これらは `average`, `price` などとすべきです。

**スコープが広い名前は長く**　プログラム全体にかかわる定数は，どこでも意味が分かるように丁寧な名前を使いましょう。

**スコープが狭い名前は短く**　10 行程度の短いプログラムでなければ，変数名を `x`, `t` のように 1 文字にすることは推奨できません。あとでソースを見直すときに検索できなくなってしまいます。 ただし，次の例の `x`, `y` のようにきわめて寿命の短い変数は 1 文字でもかまいません。

```
coordinates.each do |x,y|
  dist  = sqrt(x * x + y * y)
  grad  = y / x
end
```

**ループ変数は 1 文字でも**　for ループなどのループ変数として `i`, `j`, `k` を使うことはかまいません。これらは使用頻度が高く，計算式に組み込まれることも多いために，長い名前にするとかえって煩雑に見えます。

**メソッド名は機能がわかるように**　平均を求めるメソッドであれば `calc_average`, 条件の初期化のメソッドなら `init_conditions` のような名前を付けましょう。

**複合した名前の慣例**　プログラムの中の名前は機能や内容を表すものなので，ときどき複数の単語の組み合わせを使いたくなります。そのときの原則は次の通りです。

**クラスや定数にはキャメルケース**　これらはスコープが広く重要な意味をもつので，目立つように次のようにします。

WonderLand，ActionController，FrameWidth

このように複合した単語の頭だけを大文字にするような記法を**キャメルケース** **(camel case)** と呼びます。

**変数やメソッドにはアンダースコア区切り**　一方，英小文字で始める変数名，メソッド名の場合には，複合した名前では単語をアンダースコア '_' で区切ります。

user_id，font_name，serial_number，path_length

**しかし変数は短縮形も使うべき**　変数はソースの中で最もよく使われるので，長いとかえって煩雑になることがあります。 次のような短縮形にすることもよくあります。

uid，fntname，snum，pathlen

**配列には複数形を使うとわかりやすい**　配列の名前として名詞の複数形を使うと，意味が分かりやすくなります。

```
fruits = ["Melon","Banana","Orange","Kiwi"]
points = [[0.0,0.0],[150.0,0.0],[200.0,100.0],[0.0,100.0]]
```

上の 2 つ目は点の集合を $x, y$ 座標のペアからなる配列の配列で表したものです。

## 1.4　知っておきたい Ruby の仕様

Ruby は，「こういうふうになっていると便利だ」という仕様を多数用意しています[*1]。それもたいてい分かりやすい。他の言語でも似たような機能はありますが，Ruby にはこの種の便利な道具がきわだってたくさん用意されています。

### 1.4.1　式展開

変数 x の値を出力するには**式展開**を使うと便利なことがよくあります。下の例を比較してみましょう。

```
x = 10.0
y = 1.5
print "x = ", x, "y = ", y, "\n"
print "x = #{x}, y = #{y\n}"
puts "x = #{x}, y = #{y}"
```

ただし，下の例のように小数点以下が冗長になるような場合には，printf を使って出力の桁数を制御するべきです。printf は C, Java などにもある汎用の関数ですから使いこなせないといけません。

```
x = Math::PI
puts x
printf("%6.3f\n",x)
```

### 1.4.2　出力の整形

Ruby ではオブジェクトを分かりやすい形で出力してくれる p というメソッドがあって，数値，文字列，配列などのデータをうまく出力し分けてくれます。しかし配列の配列などは p で出力してもまだ見やすいとはいえません。

そこで，pp.rb というライブラリが用意されていて，さらに丁寧で分かりやすい出力を実現してくれています。次のソースを走らせてみましょう。

```
require 'pp'
ar = [[1,2,3],[3,4,5],[6,7,8]]
```

[*1] 便利そうな仕様を新しく思いついたら，ぜひ開発者に提案してください。役立ちそうだったら採用されるはずです。

```
p ar
pp ar
```

### 1.4.3 日本語を正しく表示させる

p や pp を使って日本語の文字列データを含む配列を表示させると，文字化けが起きることがあります。たとえば次のソースで試してみましょう。

```
# inuneko.rb
ar = ['イヌ','ネコ','ハト']
p ar
ar.each{|d| puts d}
```

これを次のように走らせます。

```
> ruby  inuneko.rb
```

出力をみると配列の中が化けてしまっています。これはおもしろくありません。

```
["\343\202\244\343\203\214","\343\203\215\343\202\263",
 "\343\203\217\343\203\210"]
イヌ
ネコ
ハト
```

文字化けを避けるには次のように ruby にコマンドラインオプションを付けて，文字コードを指定すればよろしい。

```
> ruby  -Ks inuneko.rb
```

結果は次のようになります。

```
["イヌ", "ネコ", "ハト"]
イヌ
ネコ
ハト
```

-Ks は漢字コードに Shift-JIS を使うという意味。漢字コードはプラットフォームや設定によって異なりますが，Windows であれば Shift-JIS が基本です。 MacOSX, UNIX 系ならば UTF-8 が多いので -Ku とすることになります。

Ruby のプログラムをスクリプトとして走らせるときには，次のように ruby を指定するヘッダ行に -Ks オプションを付けます。

```
#! /usr/local/bin/ruby -Ks
# inuneko.rb
ar = ['イヌ','ネコ','ハト']
p ar
ar.each{|d| puts d}
```

### 1.4.4　多重代入

複数の変数に対して，同時に複数の値を代入できます。

```
x,y = 10, 20
```

これを使うと簡単にスワップできます。

```
x = 10
y = 20
x, y = y, x
puts "#{x} #{y}"
```

配列からの多重代入も可能です。次を走らせてみましょう。

```
x, y = [5,10,15]
puts "#{x} #{y}"
```

なお，ある値をいくつかの変数にいっぺんに代入するには，次のように書いてかまいません。このような代入の仕方は C や　Java でも有効です。

```
x = y = z = 100.0
```

### 1.4.5　後置 if

else　節を伴わない単純な if 構文では， if を実行文の後に置くことができます。次の 2 つのソースを比較してみましょう。後置 if によって 3 行が 1 行にまとまっています。

```
sum = 0
1000.times do
```

```
  if rand(10) % 3 == 0 then
    sum += 1
  end
end
puts sum

sum = 0
1000.times do
  sum += 1 if rand(10) % 3 == 0
end
puts sum
```

## 1.5　イテレータ (Iterator)

データの集合を扱うデータ型としては，配列，連想配列 (ハッシュテーブル) などがあります。これらはいずれも含まれる要素を列挙していくことができます。この性質を Enumerable といいます[*2]。この種のデータの処理には，他の言語では `for` ループなどがよく使われますが，Ruby では**イテレータ**と言う便利なメソッドを用意してあります。

### 1.5.1　イテレータ each

イテレータにはさまざまのものがありますが，代表的なものは `each` というメソッドです。配列にこのメソッドを使う例としては次を見てください。次の 2 つはブロックを区切るのに `do` と `end` を使うか，あるいは中括弧の対を使うかという違いがありますが，いずれも同じように働きます。

```
ar = [3,5,7,2,9]

ar.each do |x|
  puts x
end

ar.each { |x|
  puts x
```

[*2] 数え上げたり列挙したりすることを enumerate といいます。

```
}
```

これらのプログラムは次のものと同じ働きをしますが，配列の添字を意識しなくてもすむので，スマートに処理できます。

```
ar = [3,5,7,2,9]
for i in 0 ... ar.size
  puts i
end
```

次の例は，x,y 座標のペアの配列の配列から 2 つの座標値を 2 つの変数に代入させる仕掛けです。とても分かりやすい書き方だと思いませんか？

```
  points = [[0.0,0.0],[150.0,0.0],[200.0,100.0],[0.0,100.0]]
  points.each do |x,y|   # ここに注目！
     puts "(#{x},#{y})"
  end
```

each を作用させることができるクラスには，配列以外に Range(範囲), Hash(ハッシュテーブル) などがあります。

**練習問題 1–1** 次のように定義された Range オブジェクト range と Hash オブジェクト hash がある。それぞれに each を使って，1 から 1000 までの和，および hash の値 (32, 17, . . . ) の和を求める処理を書きなさい。

```
range = 1..1000
hash  = {"a" => 32, "b" => 17, "c" => 21, "d" => 18}
```

### 1.5.2 その他のイテレータ

Ruby のイテレータには，配列の処理によく用いられる map, collect, inject など，またファイル読み込みで多用される each_line など，多種多様なものがあります。なんらかの繰り返しが関係する作業をしたいときには，リファレンスを見て使えそうな道具がないか調べてみましょう。次の練習問題では map というイテレータの使いかを学びます。これはとても便利な道具です。

文字列の扱いに関してもイテレータは強力な武器になります。ただし， Ruby 1.8 までと Ruby 1.9 以降では文字列まわりの仕様がかなり異なるので，気をつけないといけません。

**練習問題 1–2** 配列 nums が次のように定義されているものとする。

```
nums = ["12","15","21","10","30"]
```

この配列の要素は数字による文字列なので，そのままでは計算には使えない。map メソッドの使い方をリファレンスで調べて，nums を次のように整数を要素とする配列に変化させなさい。

```
[12,15,21,10,30]
```

## 1.6 メソッドチェイン

次の文字列があります。

```
str = "orange lemon mango grape apple kiwi"
```

この果物の名前をアルファベットの逆順に並べ替えて，次のようにするにはどうしたらよいでしょうか。

```
"orange mango lemon kiwi grape apple"
```

この答えは次のようになります。irb で試してみてください。

```
str.split(" ").sort.reverse.join(" ")
```

この意味を理解するには，下のように str の後のメソッドをひとつずつ増やしながら結果を見てください。

```
irb(main):001:0> str = "orange lemon mango grape apple kiwi"
=> "orange lemon mango grape apple kiwi"
irb(main):002:0> str.split(" ")
=> ["orange", "lemon", "mango", "grape", "apple", "kiwi"]
irb(main):003:0> str.split(" ").sort
=> ["apple", "grape", "kiwi", "lemon", "mango", "orange"]
irb(main):004:0> str.split(" ").sort.reverse
=> ["orange", "mango", "lemon", "kiwi", "grape", "apple"]
irb(main):005:0> str.split(" ").sort.reverse.join(" ")
=> "orange mango lemon kiwi grape apple"
```

結局，やっていることは次のような操作を連続して行っていることになります。

1. split(" ") を文字列に働かせることで，スペースで区切られた単語を配列に変更する

2. sort メソッドで配列要素を昇順に並べる
3. reverse メソッドで配列要素を逆転させる
4. join(" ") メソッドを使って，配列要素をスペースで区切って並べた文字列を生成する

このように，何らかのオブジェクトに次々にメソッドをつないでいって，望みの結果を得ることは Ruby ではしばしば使われる手法です。このようなメソッドの連鎖を**メソッドチェイン**といいます。

**練習問題 1–3** 次の文字列がある。

```
str = "Japan,CHINA,Brasil,argentine"
```

これを次のように変形させるメソッドチェインを考えなさい

```
"Argentine,Brasil,China,Japan"
```

# 第 2 章

# 自力でソースを書くために

この章ではよくあるプログラムの例を取り上げて，ソースを書いていくための基本的な作業について説明していきます。

## 2.1　クラスライブラリを活用する—文字列と配列のメソッド

### 2.1.1　クラスライブラリとは

Ruby に限らずほとんどのプログラミング言語は，さまざまの機能をライブラリとして備えています。初心者が学ぶのは言語の基本的な仕様をどう使うかという部分ですが，実際のコーディングにおいては，すでに用意されたライブラリの資源をどう活用するかが仕事の効率の決め手になってきます。すでに存在する機能を知らないで新たに自分でソースを書いていたのでは，時間の無駄遣いになってしまうし，たいていは既存のライブラリの方が高性能だったりするからです[*1]。

ライブラリの使い方は，たいていはウェブで公開されたリファレンスマニュアルに記載されていて，それを参照しながらプログラムを書いていくことになります。

ここではプログラミングにおいて最も用途の多い文字列処理と配列の扱いを題材にして，リファレンスに記載されたライブラリをどう使うかを実践的に学ぶことにしましょう。

### 2.1.2　リファレンスをよく見よう

Ruby のリファレンスの中でもっともよく参照されるのは，**組み込みライブラリ**と**標準添付ライブラリ**と呼ばれるクラスライブラリに関するドキュメントです。

組み込みライブラリには最もよく使われるクラスのメソッドが含まれています。その主なクラスを下に示しました。

---

[*1] ロシアのことわざには，「自転車を二度発明するのは愚か者だ」というのがあるそうです。とはいえ，勉強のために既存の機能を自分で実装するのは悪いことではありません。

**Array，Dir，Enumerable，File，Hash，IO，Range，Regexp，String，Time**

添付ライブラリは，ソースの中で require 'hoge' という形で読み込んで使われるライブラリです。ネットワーク機能や文字コード変換，あるいは数学関係の処理といった，特化した目的のために使われるものが多く含まれています。

### 2.1.3　文字列処理スクリプトを書く

次の問題を考えます。この種のプログラムはさまざまな分野で活用されるものです。

**200 語程度の英文をネット (たとえば http://www.cnn.com ) からコピーしてテキストファイルにしなさい。ファイル名は news.txt とします。その後，その英文の語数，重複を取り除いた単語の数を数えるプログラムを書きなさい。This と this のようにケース[*2]が異なるつづりも同じ語とみなし，またピリオド，カンマ，引用符などの記号類は数えず，空文字列 ""とみなします。**

### 2.1.4　方針

まず，必要な段取りを分析しましょう。

1. テキストファイルを読み込む
2. 不要な記号類を取り除き，大文字は小文字に変換する
3. すべての単語をひとつの配列に取り込む
4. この時点で語数を求める
5. 重複する単語を一つずつにまとめる
6. 使われている単語数を求める

最初にファイル入力があり，その後は文字列と配列の扱いが中心です。

ファイル入出力，文字列，配列は組み込みライブラリで

**ファイル入力**　関連するクラス：**IO(入出力)，File(ファイル操作)**

主な処理：ファイル読み込み

注意：IO は I/O とも書かれ，Input/Output の省略形です。ファイル入出力に関する情報は File クラスを調べるとよさそうに思えますが，その親クラスである IO クラスの方に基本的なメソッドは用意されています。File クラスのメソッドは，ファイルそのものの操作や属性に関するものが含まれます。

[*2] 大文字は uppercase，小文字は lowercase ということから，ケースが異なるというのです。

**文字列関連**　関連するクラス： **String(文字列)**, Regexp(**正規表現**)
　　主な処理：単語の切り出し，大文字と小文字の変換，文字列の置換
**配列関連**　関連するクラス：**Array(配列)**
　　主な処理：要素を数える，ソートする，重複する要素を取り除く

### 2.1.5　ファイル読み込み

ファイルをオープンして読み書きの処理を行うには，いくつかのやり方が用意されています。ここでは次のようにブロックを使う形を使いましょう。ブロックを使うと処理後のクローズを自動的に行ってくれるので，気持ちがすっきりします[*3]。

```
File.open(filename,mode) do |fo|
  # ファイルオブジェクト fo に対する処理
end
```

上の手続きによって，ファイルオブジェクト fo が生成し，これをレシーバとしてさまざまなメソッドを働かせることができるようになります。ここでやりたいことはファイルから情報を読み込むことです。ただし，気をつけなければならないのは，ファイル入出力に関するメソッドは，File クラスではなく，その親クラスである IO クラスのメソッドとして定義されているということです。

というわけで， Ruby のリファレンスマニュアルから IO クラスのページを開いてみましょう。すると，ファイルからの読み込みのために，次のように沢山のメソッドが用意されていることが分かります。

　　**read，readbyte，readchar，readline，readlines，each_line，each_char, each_byte, getbyte, getc, gets,**

あまりに多くて迷いそうですね。

が，ここで作ろうとしているプログラムでは，文字をひとつひとつ検査しながら，しかも一気にファイル全体を読み込みたいので，each_char を選ぶことにしましょう。これはファイルから文字を 1 つずつ読み込むという繰り返しを実現するイテレータです。

そこで上のソースの # **ファイルオブジェクト** fo **に対する処理** のところは次のように書くことにしましょう。

```
  fo.each_char do |c|
    # 1 個ずつの文字を使って仕事をする
```

[*3] ソースを書くときのすっきり感はとても大切です。

```
  end
```

### 2.1.6 拾い出した文字から文字列を作るには

しかし，ファイルから文字を 1 個ずつ読んでいったとして，それをどうしたらよいのでしょうか？イメージをもって考えましょう。まず，文字列というのは足し算ができるものであることを思い出してください。たとえば,

```
str = ""
str += 'A'
str += 'BC'
```

とすると， str の中身は 'ABC' となります。

このことを念頭に置いて，たとえば次の文章がファイルの最初の方にあったとしましょう。

```
"Hi, Yuki!" he said to her.
```

この情報を each_char は'"', 'H', 'i', ',', ' ', 'Y', 'u', 'k', 'i', '!', '"', ... というふうに読み込みます。これから**必要な文字だけを選んで，**何かの文字列に足し込んでいけばいいはずです。

そのために，ループよりも前のところで空文字列 text を作っておくことにしましょう。するとこんなふうになります。

```
text = ""
File.open() do |fo|
  fo.each_char do |c|
     if  必要な文字であれば
         text += c
     end
  end
...
```

### 2.1.7 正規表現で必要な文字を選び出す

イテレータ each_char のループによって 1 個ずつ読み出された文字のうち，必要なものだけを選ぶにはどうしたらよいのでしょうか。そこで登場する協力な武器が**正規表現**です。

正規表現を使うと，文字や文字列を「賢く」選ぶことができます。リファレンスの言語仕様には正規表現の項目がありますから，使えそうなものを探します。今ほしいのは，カンマやピリオド，引用符などの記号以外の文字とスペースです。それに合致しそうな正規表現を探すと次のようなものがあります。

`\w`　**英数字**
`\s`　**空白文字：スペース，タブ，改行**

ほしいのは，「英数字またはスペース」ですから，それを表す正規表現は `[\w\s]` です。これに合致する文字だけを拾いだすようにすれば，よけいな記号類はふるい落とされることになります。リファレンスでは`\s` がスペースだけでなく，改行文字も含むことに注意しておいてください。

さて，文字列（ここでは 1 文字だけ）`c` がこの正規表現とマッチするかどうかは次の式が true になるかどうかで判断できます。

```
/[\w\s] =~ c
```

これを `if` で判定してやればいいのです。したがって次のようになります。

```
if /[\w\s] =~ c
    text += c
end
```

この判定は後置 if 構文 (1.4.5 節参照) を使って次のように処理すれは 1 行で済みます。

```
text += c if /[\w\s] =~ c
```

### 2.1.8　配列を使って単語ごとに処理

ここまでのプロセスで，テキストファイルの単語はすべてスペースで区切られて `text` という長い文字列に取り込まれました[*4]。

この後はすべての文字を小文字に変換して，単語ごとに切り離さないといけません。リファレンスの String クラスの項を見ると小文字への変換には `donwcase` が用意されています。使い方は次のように簡単です。

```
text.donwcase
```

[*4] この手法は C では使えません。C の場合には文字列の長さはあらかじめ決めた制限を超えることができず，それに違反するとエラーになります。一方 Ruby ではコンピュータのメモリが許す限り長い文字列（おそらく 100 万字以上）を扱うことが出来ます。

また，すでに 1.6 節で見たように，単語を切り離して配列にするには split が利用できます。ただしこの場合にはちょっと注意が必要です。text という文字列ではスペースだけではなく，改行も単語の区切りとして使われているはずです。単に split(' ') としたのでは，改行による区切りが意味をなさなくなってしまいます。

そこでもう一度 split のリファレンスを見直すと，区切りとして正規表現が使えることになっています。今の課題では，スペースや改行などが 1 個以上連続した文字列を区切りが単語の間に入っているはずなので，/\s+/ という正規表現を区切りの指定に使えばよさそうです。結局，次の形のメソッドチェインですべて小文字化した単語の配列が得られます。結果は words という配列に格納しましょう。

```
words = text.downcase.split(/\s+/)
```

この配列の要素の数が，使われている語数です。それを出力するには単に次のようにします。

```
puts  words.size
```

次に，words という配列から重複した要素を取り除くにはどうしたらよいのでしょうか。実はこれもすでに uniq というメソッドが用意されています。uniq というのは「一義の」という意味で，重複をなくすことを意味しています[*5]。リファレンスで調べて，irb で動作をチェックしてみてください。したがって， uniq を働かせると同一の単語はすべて 1 つにまとめられるので，異なる単語の数を次のようにして数えることができます。

```
puts words.uniq.size
```

最後に，完成形のソースをまとめて下に示しておきます。

```
# word_count.rb
textfile = "news.txt"
words = []
text = ""
File.open(textfile) do |fo|
  fo.each_char do |c|
    text += c if /[\w\s]/ =~ c
  end
end
words = text.downcase.split(/\s+/)
```

[*5] 日本語では「ユニーク」＝「独特の」という意味ですが，それだとかなり偏った意味になってしまいます。

```
puts  words.size
puts  words.uniq.size
```

**練習問題 2–1**　上で作成したプログラムを少し書き換えると，ある単語が何回出現したかを集計することができます。実際にプログラムを作成してください。
ヒント：「単語ごとの出現回数」を表すデータ表現としては，`'a'` → 12,`'about'` → 7 というふうに単語に整数が対応づけられた形を利用すればいいはずです。

## 2.2　処理をメソッド化する

ソースを分かりやすくするためには，まとまった処理はメソッド化しておいて，メインから呼び出すようにするのが常道です。ここではそのやり方を学びましょう。例として次のプログラムを取り上げます。

```
# calc_average.rb
datafile = 'seiseki.csv'
sex = "F"
data = []
File.open(datafile) do |f|
  f.each_line do |line|
    c = line.chomp.split(',')
    if c.size > 1 then
      if c[1] == sex then
        data << c[0].to_i
      end
    end
  end
end
upperLimit = 90
lowerLimit = 60
sum = 0
count = 0
data.each do |v|
  if v >= lowerLimit && v <= upperLimit then
    sum += v
    count += 1
  end
end
average = sum / count.to_f
printf("average = %4.1f\n",average)
```

このプログラムは，次のようなフォーマットの CSV ファイルがあるものとして，与えられた性別 'F' か 'M' のいずれか，および与えられた上限と下限の間にある整数のデータを選び出し，平均を出力します。

```
80,F
76,M
52,F
65,F
90,M
...
```

### 2.2.1　プログラムの機能を独立させる

このプログラムの働きは 2 つに分けることができます。

1. データをファイルから読み込んで，男女いずれかの点数の配列を作る。
2. 配列のデータのうち，下限以上，上限以下のデータを選んで，それらの平均を計算する。

これらを独立させてメソッド化することにします。

**ファイル入力と配列生成**

手続きをメソッド化するときには，そのメソッドが必要とする情報が何であるかを考えないといけません。最初の部分で必要な情報は何でしょうか。

こういう問題の答えは状況によって異なるのですが，ここではファイル名，男女のいずれであるかという情報をメソッドが要求するものとしましょう。すると，メソッド定義の骨格は次のようになります。

```
def read_file(filename,sex)
   # いろいろな処理
   # できた配列を返す。
end
```

このメソッドの中には，ソースのどこからどこまでを含めればよいでしょうか。ファイルのオープンからクローズまでの処理は当然で，その前に空の配列 data を用意するところも含めます。最後にそれを戻り値として返すところまでがこのメソッドの範囲です。

```
def read_file(filename,sex)
  data = []
  File.open(filename) do |f|
    f.each_line do |line|
      c = line.chomp.split(',')
```

```
      if c.size > 1 then
        if c[1] == sex then
          data << c[0].to_i
        end
      end
    end
  end
  return data
end
```

### データを計算する部分

こちらのメソッドに必要なのは，データの下限と上限です。また配列データも当然必要です。それ以外の sum とか count といった変数はメソッドの中で閉じて使われています。したがって，このメソッドは次のようになります。

```
def calc_data(ll,ul,data)
  count = sum = 0
  data.each do |v|
    if v >= ll && v <= ul then
      sum += v
      count += 1
    end
  end
  return sum / count.to_f
end
```

### 全体を組み立てる

これらのメソッドを組み込んだ全体はどうなるべきでしょうか。Ruby のソースではメソッドが先に置かれ，メインの実行部分はその後に置かれるという配置の仕方にしなくてはいけません[*6]。

したがって，結局次のようなソースになります。ソースが見やすくなるように，メソッドとメソッドの間，メソッドとメインの間には 1 行の空白を入れておきましょう。このソースの省略を補って完成して，実際に走らせてください。

[*6] C や Java では，メインの場所はどこにでも置けます。その代わり， main という名前のメソッド (C では関数) をあらわに指定して，どこが最初に実行されるべきメインなのかを明示する必要があります。

```
# calc_average.rb
def read_file(filename,sex)
  data = []
  ... 省略
  return data
end

def  calc_data(data,lowerLimit,upperLimit)
sum = count = 0
  ... 省略
return sum / count.to_f
end

datafile = 'seiseki.csv'
sex = "F"
data = read_file(datafile,sex)

upperLimit = 90
lowerLimit = 60
average = calc_data(data,lowerLimit,upperLimit)
printf("average = %4.1f\n",average)
```

# 第 3 章

# 再帰的方法

## 3.1 再帰的な記述のかたち

**例題 1【配列を追跡する】** データが格納された配列があります。この配列の要素をすべて書き出すプログラムを再帰的なメソッドを使って書きなさい。範囲外の添字で配列にアクセスすると nil が返る[*1]ことを利用すること。

配列の要素を書き出すために再帰的な方法を使うのは実用的な意味はありませんが，練習のためにやってみましょう。

```
# read_array.rb
def read_next(i,ar)
  puts ar[i]
  v = ar[i+1]
  if v then
    read_next(i+1,ar)
  else
    return
  end
end

ar = [2,4,6,8,10]
read_next(0,ar)
```

このプログラムの動作を調べましょう。

[*1] Java, C では範囲外の添字で配列にアクセスするとエラーになります。

1. 最後の 2 行を見ると，配列 `ar` が定義されて，`read_next` というメソッドに引数として 0 と `ar` が渡されています。
2. 次に，呼び出された `read_next` を追ってみる。最初の仮引数の `i` には 0 が渡されてきます。
3. `ar[0]` を出力
4. `ar[i+1]` = `ar[1]` の値が `v` に代入されます。
5. `v` が値をもっていたら `read_next` の引数に 1 を渡して呼び出す。

**例題 2【ユークリッドの互除法】**　2 つの整数の最大公約数をもとめるメソッドを作りなさい。

ユークリッドの互除法は 2 つの整数の最大公約数 LCD(Largest Common Divisor) を求める方法で，歴史上もっとも古くから知られているアルゴリズムです。

1. 2 つの整数を $a, b$ とする
2. $a > b$ となるように，必要ならスワップ
3. $r$ に $a$ を $b$ で割った余りを代入
4. もし $r = 0$ なら，$b$ を 返して終了
5. そうでなければ $a \leftarrow b$, $b \leftarrow r$
6. 3 に戻る。

具体的に 819 と 520 の LCD を求めてみましょう。

1. 819 を 520 で割った余りは 299
2. 520 を 299 で割った余りは 221
3. 299 を 221 で割った余りは 78
4. 221 を 78 で割った余りは 65
5. 65 を 13 で割った余りは 0

従って 13 が 819 と 520 の LCD です。

以上の手順を実装すると次のようになります。

```
# Euclid's algorithm
def get_lcd(m,n)
  m, n = n, m if m < n
  q = m % n
  if q == 0 then
    return n
  else
```

```
    get_lcd(n,q)
  end
end
m, n = ARGV[0].to_i,ARGV[1].to_i
if m <= 0 || n <= 0 then
  puts "*** USAGE:  #{$PROGRAM_NAME}  m  n"
else
  puts get_lcd(m,n)
end
```

**練習問題 3–1**　引数として整数を与えると，その階乗 (factorial) を返すメソッドをループと再帰的方法の両方を用いて作りなさい。

**練習問題 3–2**　フィボナッチ (fibonacci) 数列は下の漸化式で定義される数列で，始めの部分は $1, 1, 2, 3, 5, 8, 13, \ldots, n$ となります。いま，$n$ を引数として与えたときに，第 $n$ 項を返すメソッドをループと再帰的方法の両方を用いて作りなさい。

$$a_1 = 1$$
$$a_2 = 1$$
$$a_n = a_{n-1} + a_{n-2},\ (n = 3, 4, 5, \ldots)$$

# 第 4 章

# グラフ関連のプログラム

## 4.1 グラフとは何か

**グラフ** **graph** というのは，ものとものとの「つながり」のようすを図 4.1 のような点と線で表した図形です。ただしここでは，線の長さとか点の間の距離には特に意味はなく，単に**どの点とどの点がつながっているか**ということだけを考えます。このように連結の有無だけを考慮したグラフは**単純グラフ**といいます。

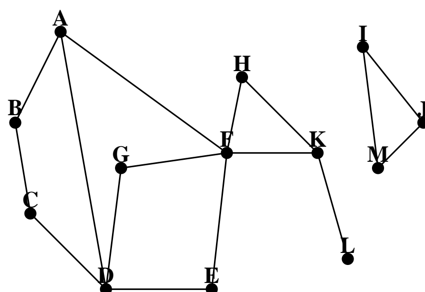


図 4.1　グラフの例

グラフを構成する要素は，**頂点** **(vertex)**[*1*2] と**辺** **(edge)** です。頂点は辺によって**連結**されて **(connected)** います。

ある頂点からある頂点へ，途中の辺をたどりながら訪問することができるとき，この道を**パス** **(path)** あるいは**経路**といいます。

---

[*1] 英語では node ということもあります。

[*2] 複数形は vertices

### 4.1.1　グラフで表現されるもの

知り合い同士がつくる人間関係はグラフで表すことができます。たとえば図 4.1 を 13 人の人たちの交友関係とみれば，だれが付き合いが広く，だれがそうでないか，どのようなグループが存在するかは一目瞭然です。他には，たとえば鉄道の路線図も代表的なグラフですし，インターネットも巨大なグラフと考えることができます。

## 4.2　グラフのデータ表現

### 4.2.1　隣接行列を使う

グラフを表現するやり方としては行列を使うのが分かりやすいやり方です。たとえば図 4.2 のグラフを表すには次の形の行列が使えます。

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \tag{4.1}$$

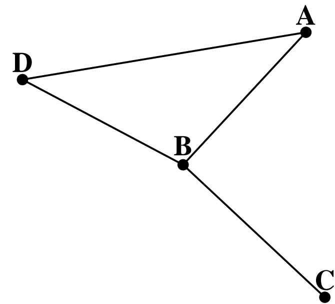


図 4.2　4 つの頂点をもつグラフ

ここでは，行列の要素は 0 から数えるものとし，0,1,2,3 に対してそれぞれ頂点 **A, B, C, D** が対応づけられていると考えてください。図では A と **B** の間が連結されていますが，行列では 1 行 2 列および 2 行 1 列の位置の要素が 1 となっていることでこの連結が表現されています。なお，自分自身との連結を意味する対角要素は，この式では 1 になっていますが， 0 とされることもあります。

このようにグラフを表す行列を**隣接行列 (adjacent matrix)** と呼びます。プログラム上で式 (4.1) の行列を表現するには，次のような配列の配列（二重配列）が使われます。

```
adj_matrix = [[1,1,0,1],
              [1,1,1,1],
              [0,1,1,0],
              [1,1,0,1]]
```

### 4.2.2　隣接リスト

グラフを表現するのに，表 4.1 のようにすべての頂点についてそれぞれが隣接している頂点すべてを並べたリストを使うことができます。このようなリストを**隣接リスト**

表 4.1　図 4.2 のグラフの隣接リスト表現

| A | B,D |
|---|---|
| B | A,C,D |
| C | B |
| D | A. B |

**(adjacent list)** といいます。たとえば図 4.2 のグラフを見ると，**A** の頂点は **B, D** に隣接しています。つまり，**A** は **B, D** という集合に結びつけられているわけです。

これを Ruby で表現するにはどうしたらよいでしょうか。Ruby にはハッシュ（連想配列）というデータ構造が用意されています。これは任意のオブジェクトに他のオブジェクトを対応づけるためのデータ構造です。ハッシュを使うと，**A** から 集合 $\{\mathbf{B}, \mathbf{D}\}$ への対応付けを次のように表現できます。

```
{'A'=>['B', 'D']}
```

他の頂点についても同様ですから，結局次のようなハッシュがあれば，図 4.2 のグラフを表現できることになります。

```
adj_list = {'A'=>['B', 'D'],'B'=>['A','C','D'],'C'=>['B'],'D'=>['B','A']}
```

実際に irb で実験してみましょう。上の行を入力した後，次のような入力と応答を確認してください。

```
adj_list['A']
=> ["B","D"]
adj_list['B']
=> ["A","C","D"]
adj_list['B'][0]
=> "A"
```

### 4.2.3　隣接行列と隣接リストのメモリ使用効率

グラフの頂点の数を $N$ とすると，隣接行列に使われるメモリの量は $N^2$ に比例します。一方，辺の数は $N^2$ よりも低いオーダーで，たとえば $N$ に比例して増加することが多いので，$N$ が大きい場合には大半の行列要素が 0 ということになり，情報の利用効率としては悪くなります。隣接行列を大きなグラフに用いるときには，メモリの使用量を気にした方がよいでしょう。

一方，隣接リストのメモリ使用量は，頂点の数と辺の数の和に大体比例するので，ずっと効率的です。そこでこれ以降は主に隣接リストによるグラフの表現を使うことにします。

**練習問題 4–1**　CSV ファイルに次のように記述されているものとします。

```
A,F,C,B,G
B,A
C,A
D,F,E
E,G,F,D
F,A,E,D
G,E,A
H,I
I,H
J,K,L,M
K,J
L,J,M
M,J,L
```

これを読み込んで，上の adj_list のようなハッシュを作りたい。そのためのプログラムを書きなさい。

### 4.2.4　深さ優先探索

ひとつの頂点から出発して，辺をたどりながら連結されたすべての頂点を訪問することは，グラフを扱う上でもっとも基本的な処理です。ここでは深さ優先探索と呼ばれる探索の実装を，じっくり考えていくことにします。

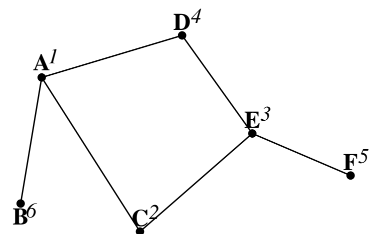


図 4.3　adj_list を使って深さ優先探索したときの頂点の訪問順序

#### 探索のアルゴリズム

図 4.3 を見てください。このグラフの隣接リストは次のようになっているものとします。

```
adj_list = {
  'A'=>['D','B','C'],
  'B'=>['A'],
  'C'=>['E','A'],
```

```
'D'=>['A','E'],
'E'=>['F','C','D'],
'F'=>['E']}
```

頂点 **A** から出発して，隣接リストを参照しながらすべての頂点を訪問する手順を以下で説明します。

1. **A** を現在の訪問先頂点 cv とする。
2. cv が未訪問であれば，
   (a) cv を訪問済みとする。
   (b) もしも訪問予定の記憶に cv があったら取り除く。
   (c) cv を訪問していることを出力する。
3. cv に隣接する頂点のリストの要素をそれぞれ取り出して，v とする。
4. v が訪問済みでなければ，訪問予定の記憶に追加する。
5. 訪問予定の記憶から，**最後の頂点**を取り出して cv とする。
6. cv が nil ならもう記憶されているデータはないので終了。
7. 2 に戻る。

上の手順で訪問していくと，図 4.3 に示された順序でたどることになります。この訪問の仕方の特徴をよく見ましょう。ひとつの頂点が複数の隣接する頂点を持っていた場合，それらを全部訪問してから先に進むのではありません。そのうちのひとつを訪問したら，そこからまた隣接する頂点のリストを見て進んでいくのです。つまり，とにかく先へ先へと進むような方針です。このような訪問の仕方を**深さ優先探索 (depth-first search)** といいます。

**練習問題 4–2**　上の箇条書きで説明した手順に従って，記憶されている頂点がどのように変化していくかを最後まで追跡しなさい。

### 4.2.5　実装に使われるデータ構造

深さ優先探索のアルゴリズムを追ってみると，2 種類の記憶領域が必要になることがわかります。ひとつは，どの頂点が訪問済みなのかを記録するための記憶領域です。これにはハッシュを使うのが便利そうです。

もうひとつは，複数の訪問の候補のうち，未処理のものを記憶しておくための領域です。そのためにはスタックが使われます。

### 訪問済みのフラグはハッシュで

それぞれの頂点が訪問済みかどうかを記憶するためには，ハッシュが便利です。そのハッシュの名前と形は次のようになっていればいいでしょう。

```
visited = {
  'A'=>false,'B'=>false,'C'=>false,
  'D'=>false,'E'=>false,'F'=>false}
```

これは初期値です。処理の過程でどれかの頂点を訪問するたびに，false を true に変更していきます。論理値で訪問済みかそうでないかを表現するわけです。状態を記録するために使われるこの種の記憶は**フラグ**と呼ばれます。

### 隣接リストから訪問記録のハッシュを生成する

32 ページにある adj_list というハッシュから 34 ページの visited というハッシュを生成するには，どうしたらよいのでしょうか。

リファレンスで Hash クラスを調べてみると，keys というメソッドが見つかります。この働きを irb で試してみます。

```
irb(main):001:0> h = {'a'=>10,'b'=>15}
=> {"a"=>10, "b"=>15}
irb(main):002:0> h.keys
=> ["a", "b"]
```

このように keys メソッドはハッシュのキーを配列として返してくれます。

さて，今やりたいことはハッシュ adj_list のキーである 'A','B',... のひとつひとつに false を関連づけたハッシュ visited がほしいわけです。配列からひとつひとつを取り出して処理したいとき， each が便利であることはすでに 2 章で出てきましました。そこで次のような形の処理が思い浮かびます。

```
adj_list.keys.each do |v|
  # ハッシュ visited の中で v に false を対応づけます。
end
```

visited は上のループの前で空のハッシュとして定義されていないといけません。対応付けは visted[**キー**] = **値** という形で実行されます。そこで次のように書けば，目的を達することになります。

```
visited = Hash.new
```

```
adj_list.keys.each do |v|
  visited[v] = false
end
```

**訪問予定はスタックに蓄える**

次に，未処理の頂点の集合を記憶しておくにはどうしたらよいのでしょうか？上の手順を見ると，最後に記憶したデータが最初に取り出されていることが分かります。このようなデータ構造は**スタック**と呼ばれています[*3]。スタックにデータを入れる操作は push，取り出す操作は pop です。

Ruby では，スタックをどのように実現できるのでしょうか？リファレンスで Array クラスのメソッドを見ると，`push` と `pop` というのがあります。これを irb で試してみましょう。

```
irb(main):001:0> ar = []
=> []
irb(main):002:0> ar.push('A')
=> ["A"]
irb(main):003:0> ar.push('B')
=> ["A", "B"]
irb(main):004:0> ar.pop
=> "B"
irb(main):005:0> ar.pop
=> "A"
irb(main):006:0> ar.pop
=> nil
```

後から `push` で入れたデータが最初に取り出されています。また，空の配列から取り出そうとすると，`nil` が返されます。

ここでは訪問予定の頂点を記憶するスタックとして，`stack` という名前の配列を用意することにしましょう。

```
stack = Array.new
```

これは単に `stack=[]` でもかまいません。

[*3] スタックのデータは最後に入ったものが最初に出てくるので，LIFO (Last-In-First-Out) といわれます。

### 4.2.6　アルゴリズムを実装する

実装に必要な準備として，adj_list (32 ページ), visited(34 ページ), stack(35 ページ) はすでに用意できましました。これらを並べた後に，アルゴリズムの手順通りに処理を追加すれば，深さ優先探索は完成です。

#### ループの仕掛け

33 ページの説明では，まず頂点 **A** を訪問先 cv として，繰り返しの処理（ループ）に入っています。このループは cv が nil になったときに終了します。Ruby では，nil と false 以外の値はすべて真として扱われるので，このループは while を使って次のように書けることになります。

```
cv = 'A'
while cv
  # 処理
end
```

#### 処理の本体

あとはループの中の処理を書いていくだけです。まず (2) のところはどう書けるでしょうか。(a) の cv を訪問済みとするには， visited[cv] に true を代入すればいいでしょう。

(b) で訪問予定の記憶というのは，stack のことです。そこに cv に一致するデータがあれば取り除くわけですから，配列から特定の要素を削除するメソッド delete を使えます。なお，「もし」という条件は無視してかまいません。どうせないものが削除されることはないのですから delete メソッドは cv に一致するデータが stack に含まれているときだけ働きます。

(c) はもちろん puts を使います。puts "#{cv} visited." とかするとかっこいいでしょう。

(3)(4) では， cv に隣接する頂点の集合 (adj_list) から，要素を順次取り出して v とし，それを使っていろいろな操作を行っています。こんなときには each イテレータを使ったループが便利です。

```
adj_list(v).each do |v|
   # v を使った処理
end
```

4. は訪問予定の記憶（スタック）に対する追加 (push) ですから， `push` メソッドが使えます。条件を後置 if で与えれば，次のように書けます。

```
stack.push(v) if ! visited[v]
```

5. はスタックへの追加を終えた後の処理ですから，気をつけましょう。スタックから データを pop して `cv` に代入するだけですから，次のように書くだけです。

```
cv = stack.pop
```

その後に `while` ループを閉じる `end` が来て，処理は完結です。

**練習問題 4–3**　上の説明をもとにして，深さ優先探索のソースを書きなさい。
**練習問題 4–4**　練習問題 4–3 のソースを書き換えて，訪問する手続きをメソッドとして呼び出すようにしなさい。メソッド名は `visit` とします。

**練習問題 4–5**　練習問題 4–1 を参考に，CSV ファイルに書かれた連結情報を使って探索を実行するプログラムを書きなさい。

### 4.2.7　再帰的なアルゴリズムで書く

グラフの訪問では，再帰的な手続きがよく使われます。再帰的なメソッド呼び出しでは，現在の状態が自動的にスタックにしまわれて，一段深い処理へ進むために，比較的自然な記述が可能なのです。

再帰的呼び出しのプログラムでは，あるメソッドが自分自身を呼び出すという形が現れます。頂点を訪問していくメソッドを visit という名前で定義することにしましょう。

ここではまず，再帰的に書かれた深さ優先探索のプログラムを紹介しておきましょう。

```
# depth_first2.rb
def visit(cv,adj_list,visited)
  visited[cv] = true
  puts "visited #{cv}"
  adj_list[cv].each do |v|
    if ! visited[v]
      visit(v,adj_list,visited)
    end
  end
end
visited = Hash.new
# adjacent list
adj_list = {
  'A'=>['D','B','C'],
  'B'=>['A'],
  'C'=>['E','A'],
  'D'=>['A','E'],
  'E'=>['F','C','D'],
  'F'=>['E']}

# Initialize visited flags
adj_list.keys.each do |key|
  visited[key] = false
end
cv = 'A'  # current vertex
visit(cv,adj_list,visited)
```

# 第 5 章

# いろいろな問題

## 5.1 数の処理に関わる問題

**練習問題 5–1** 自然数 $n$ を 2 進法に変換し，その結果を配列に格納するプログラムを書きなさい。具体的には例えば 89 (2 進法で 1011001) が与えられると，`[1,0,1,1,0,0,1]` という配列を作り出せればよろしい。

# 付録 A

# 解答と解答

## 第 1 章　いろいろな取り決め

**問題 1–1**

### Range オブジェクトにイテレータを作用させる

```
range = 1...1000
sum = 0
range.each{|v| sum += v}
puts sum
```

### Hash オブジェクトにイテレータを作用させる

ハッシュはキーと値をペアにしてもつデータ構造なので，次のように each を作用させて |key,val| で受けると，キーと値のペアが順次この 2 つの変数に代入されていきます。

```
hash  = {"a" => 32, "b" => 17, "c" => 21, "d" => 18}
sum = 0
hash.each{|key,val| sum += val}
puts sum
```

**問題 1–2**

map を使えば，配列のすべての要素を簡単に変更できます。irb で試してください。ここでは単に map ではなく， map! を使って，元の配列を破壊的に変更しています。

```
nums  = ["12","15","21","10","30"]
p nums
nums.map!{|v| v.to_i}
p nums
```

**問題 1–3**

メソッドチェインによる処理では，ひとつの目的に対して複数のやり方が考えられます。一例を示します。

```
str.split(",").map{|s|s.capitalize}.sort.join(",")
```

ここでは map の使い方に注目してください。

## 第 2 章　自力でソースを書くために

**問題 2–1**

次のソースのようになります。ここではハッシュを使っていることが決め手になっていますが，さらに初期化に際して Hash.new(0) と 0 を引数として与えることで，ハッシュの値を 0 に初期化できることを使っています。この仕様はリファレンスで確かめてください。

```
# word_freq.rb
textfile = "news.txt"
words = []
frequency = Hash.new(0)
text = ""
File.open(textfile) do |fo|
  fo.each_char do |c|
    text += c if /[\w\s]/ =~ c
  end
end
words = text.downcase.split(/\s+/).sort
words.each do |w|
  frequency[w] += 1
end
frequency.keys.each do |key|
  puts "'#{key}': #{frequency[key]}  "
end
```

## 第 3 章　再帰的方法

**問題 3–1**

ループを使ったメソッド factorial_loop と再帰的に処理したメソッド factorial_recursive を下に示します。

```
def factorial_loop(n)
  f = 1
  for i in 1 .. n
    f = f * i
```

```
  end
  return f
end

def factorial_recursive(n)
  if n == 0 then
    return 1
  else
    return n * factorial_recursive(n-1)
  end
end

for i in 0 .. 10
  puts factorial_loop(i)
  puts factorial_recursive(i)
end
```

**問題 3–2**

ループを使ったメソッド fibonacci_loop と再帰的に処理したメソッド fibonacci_recursive を下に示します。

```
def fibonacci_loop(n)
  a = b = 1
  (n-1).times do
    a, b = b, a + b
  end
  return a
end

def fibonacci_recursive(n)
  if n == 1 || n == 2 then
    return 1
  else
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
  end
end

for i in 1 .. 15
  puts fibonacci_loop(i)
  puts fibonacci_recursive(i)
end
```

## 第 4 章　グラフ関連のプログラム

**問題 4–1**

```
# csv_adjlist.rb
adj_list = Hash.new

csvfile = ARGV.shift
File.open(csvfile) do |f|
  f.each_line do |line|
    c = line.chomp.split(",")
    if c.size > 1 then
      v = c.shift
      adj_list[v] = c
    end
  end
end
p adj_list
```

**問題 4–2**

[] → [D,B,C] → [D,B] → [D,B,E] → [D,B] → [D,B,F,D] → [D,B,F] → [B,F] → [B] → [] →

**問題 4–3**

```
stack = Array.new
visited = Hash.new

# adjacent list
adj_list = {
  'A'=>['D','B','C'],
  'B'=>['A'],
  'C'=>['E','A'],
  'D'=>['A','E'],
  'E'=>['F','C','D'],
  'F'=>['E']}

# Initialize visited flags
adj_list.keys.each do |key|
  visited[key] = false
end

cv = 'A'  # current vertex
while cv
```

```
  if ! visited[cv]
    visited[cv] = true
    stack.delete(cv)
    puts "visited '#{cv}'"
  end
  adj_list[cv].each do |v|
    stack.push(v) if ! visited[v]
  end
  cv = stack.pop
end
```

**問題 4–4**

練習問題 4–3 の解答のソースでは，下の部分で頂点の訪問が行われています。

```
cv = 'A'  # current vertex
while cv
  if ! visited[cv]
    visited[cv] = true
    stack.delete(cv)
    puts "visited '#{cv}'"
  end
  adj_list[cv].each do |v|
    stack.push(v) if ! visited[v]
  end
  cv = stack.pop
end
```

これをメソッド化して，メソッド呼び出しで実行することにします。メソッドの役割は「この頂点をから訪問せよ」という命令に応えることですから，頂点として cv を与えなければならないことは明らかです。したがって，次の形がまず浮かびます。

```
def visit(cv)
  # いろいろな処理
end
```

「いろいろな処理」には， while 以降のソースをそっくりコピーします。そうしておいて，このメソッド全体をプログラムの先頭に移動します

**問題 4–5**

プログラム例を示します。コマンドライン引数からファイル名を指定して走らせると，頂点の一覧を出力して，そのどれかを選ぶように指示します。そこで 'A' のように入力すると，そこから深さ優先探索で訪問する頂点を表示します。

```
# depth_first2.rb
def read_file(csvfile)
```

```
  adj_list = Hash.new
  File.open(csvfile) do |fo|
    fo.each_line do |line|
      c = line.chomp.split(",")
      if c.size > 1 then
        v = c.shift
        adj_list[v] = c
      end
    end
  end
  return adj_list
end

def visit(cv,adj_list,visited)
  visited[cv] = true
  puts "visited #{cv}"
  adj_list[cv].each do |v|
    if ! visited[v]
      visit(v,adj_list,visited)
    end
  end
end

visited = Hash.new
csvfile = ARGV.shift
adj_list = read_file(csvfile)
puts "vertices: "
puts adj_list.keys.join(", ")
print "Input vertex > "
cv = gets.chomp
visit(cv,adj_list,visited)
```

## 第 5 章　いろいろな問題

**問題 5–1**

**【解答例 1】**最も素直なやり方は，元の数を 2 で割っていって，その余りを配列に記録していくことです。ただしその場合には，配列の後の方に位の高い桁が来てしまって，結果が反転してしまいます。Ruby では配列を反転させる reverse というメソッドがあるので，それを使うと簡単です。

```
n = ARGV.shift.to_i
ar = []
while n > 0
  ar << n % 2
```

```
  n /= 2
end
ar.reverse!
p ar
```

**【解答例 2】**次に示すのはずるいやり方です。sprintf 関数は printf と同じく，引数をさまざまなフォーマットで出力することができるので，それを利用します。書式指定に "%b" を用いると 2 進数で表現した文字列が得られます。それに split メソッドを働かせると，["1", "0", "1", "1", "0","0","1"] のような配列が生成します。しかし，この配列の要素は文字列なので，配列のすべての要素を変更するために map メソッドを使えばよいというわけです。実質たった 1 行のプログラムですが，メソッドチェインで複雑な処理を実現しています。

```
n = ARGV.shift.to_i
sa = sprintf("%b",n).split("").map{|v|v.to_i}
p sa
```