プログラミング教育の考え方: データ構造・計算量

久野　靖[1]

2015.8.5

[1]筑波大学ビジネスサイエンス系

はじめに

本講座は「SSR: 大学の講義を聞こう 2015」と「情報処理学会教員免許更新講習: プログラミング教育の考え方: データ構造・計算量」を兼ねる形で実施します。

本資料の内容の主要部分は、久野が東京大学で実施している「アルゴリズム入門 (2014 年までは情報科学)」の内容のおおよそ中央部分に対応しています。本講座ではこの内容を「講義で説明を受け、実際に課題演習で説明された内容を確認し、その後解説を聞く」というサイクルを反復する形で進めて行きます。

なお、同授業の前半部分 (入門のあたり) は昨年の講習会で説明したので、今回はその先という形で実施します。そのため「プログラミング経験者で、繰り返しと枝分かれ程度は分かる人」という前提で進めさせて頂きます。ただ、環境や言語への慣れも必要ですから、最初の章は入門的な部分の再掲としました。その先が今回の主な内容ということになります。

演習については、東京大学のご好意により、教育用計算環境を使用して実施します。どの演習をやるかは講座中で指示しますが、だいたいは複数の演習問題からの選択なので、興味を持ったものを選んで実施してください。後での検討のため、演習ができたら、その結果はメールで久野までお送り頂くようにお願いしています。

> **重要**: 報告メールの Subject:は「演習 2-1, 2-2」のように演習番号を記載すること。また、内容としては**作成したプログラムに加えて、その簡単な説明と、「やってみて分かったこと (感想でもよい)」を必ず記載すること。**

一方、「検討」と記された節は「プログラミング教育の考え方」として本講座のために追加した部分であり、それぞれの箇所において、プログラミングや情報科学をうまく学んでもらうために、どのような配慮をおこなっているか、どのような配慮が望まれるかについての提案と open question となっています。この箇所では、時間が許す範囲で、実際に受講された皆様からご意見を頂き、議論を行いたいと思います。

1 日という限られた時間ですので、どこまで進むかはその時々の状況によるものと考えています。無理に急いでもいいことはありませんので、用意した内容の最後まで到達しないかも知れませんが、予めご了承ください。では、よろしくお願いします。

第1章　Ruby入門: 簡単なプログラミング

1.1　アルゴリズムとプログラミング言語

1.1.1　プログラミング言語

アルゴリズムとはコンピュータなどに実行させたい「手順」であり、**プログラム**とは、アルゴリズムを実際にコンピュータ与えられる形で表現したものです。そして、その具体的な「書き表し方」ないし「規則」のことを**プログラミング言語** (programming language) と呼びます。これはちょうど、人間が会話をする時の「話し方」として「日本語」「英語」などさまざまな言語があるのと同様です。ただし、**自然言語** (natural language — 日本語や英語など、人間どうしが会話したり文章を書くのに使う言語) とは違って、プログラミング言語はあくまでもコンピュータに読み込ませて処理することがが前提の人工的な言語であり、そのため書き方も杓子定規です。

ひとくちにプログラミング言語といっても、実際にはさまざまな特徴を持つ多くのものが使われています。ここでは、プログラムが簡潔に書けて簡単に試して見られるという特徴を持つ、**Ruby**という言語を用いてゆきます。

1.1.2　Ruby言語による記述

では、三角形の面積計算アルゴリズムをRubyプログラムに直してみましょう。本クラスでは入力と出力は基本的に **irbコマンド** (irb command) [1] の機能を使わせてもらって楽をするので、計算部分だけをRubyの**メソッド** (method) [2] として書くことにします。先にアルゴリズムを示した、三角形の面積計算を行うメソッドは次のようになります:

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

詳細を説明しましょう。

1. 「`def` メソッド名」～「`end`」の範囲が1つのメソッド定義になる。
2. メソッド名の後に丸かっこで囲まれた名前の並びがある場合、それらは**パラメタ** (parameter) [3] の名前となる。メソッドを呼び出す時、これらのパラメタに対応する値を指定する。
3. メソッド内には**文** (statement) [4] がいくつあってもよい。それぞれの文は行を分けて記述するか、1行に書く場合は「`;`」で区切る。たとえばこのメソッド本体は「`s = (w * h) / 2.0; return s`」のように1行にしてもよい。
4. 式は原則として先頭から順に1つずつ実行される。

[1] Rubyの実行系に備わっているコマンドの1つで、さまざまな値をキーボードから入力し、それを用いてプログラムを動かす機能を提供してくれます。

[2] メソッドは他の言語で言う**手続き**ないし**サブルーチン** (subroutine) に相当し、一連の処理に名前をつけたもののことです。なお、手順も英語ではprocedureですが、日本語では手順と言う場合は抽象的な (プログラムとして書き下す前の) ものを指し、手続きと言う場合はプログラムに含まれる名前のついたひとまとまりのコードを指すというふうに使い分けられます。

[3] メソッドを使用するごとに、毎回異なる値を引き渡して、それに基づいて処理を行わせるための仕組みです。

[4] プログラムの中の個々の命令のことを、プログラム言語の用語では文と呼びます。

5. **return 文** (return statement)「`return` 式」を実行すると、メソッドの実行は終わり、その式の値がメソッドの値となる。

上の例は擬似コードに会わせるように、面積の計算結果を変数 `s` に入れてからそれを `return` していましたが、`return` の後ろに計算式を直接書くこともできるので、次のようにしても同じです:

```
def triarea(w, h)
  return (w * h) / 2.0
end
```

このように、たったこれだけのコードでも、大変細かい規則に従って書き方が決まっていることが分かります。要は、プログラミング言語というのはコンピュータに対して実際にアルゴリズムを実行する際のありとあらゆる細かい所まで指示できるように決めた形式なのです。

そのため、プログラムのどこか少しでも変更すると、コンピュータの動作もそれに相応して変わるか、(もっとよくある場合として) そういうふうには変えられないよ、と怒られることになります。いくら怒られても偉いのは人間であってコンピュータではないので、そういうものだと思って許してやってください。

1.1.3　動かしてみよう!

では、このコードを動かしてみましょう。まず、エディタで上と同じ内容を `sample1.rb` というファイルに打ち込んで保存してください。この、人間が打ち込んだプログラムを (プログラムを動かす「源」という意味で)「**ソース**」「**ソースコード**」などと呼びます。Ruby のソースファイルは最後を「`.rb`」にするというのが通例です。

例年、ここで「エディタって何?」となる人がいますので、簡単な方法を説明します。既にエディタを使っている人は無視してください。Mac OS では「TextEdit」「テキストエディット」と呼ばれるエディタがいちばん説明なしに操作できるのでこれを説明します。まず Finder の窓を出し、「アプリケーション」フォルダを選んで、その中から上記エディタをドラグしてドックに入れてください。以後はドック内のアイコンを選択することでエディタが起動できます。そうしたらプログラムを打ち込んでください。なお、プログラムの記述に際して日本語文字は当面使わないこととしてください。

重要! テキストエディットを使う場合は「フォーマット」メニューで「標準テキストにする」を選んでから打ち込み始めること。標準テキストでないと `irb` は正しく読めません。

次にエディタでファイルを打ち込んだあと、それを「どこに」保存するかも大切です。以下でコマンドを実行しようとするときには「ホーム」のファイルが直接見えるので、一番簡単なのは「ホーム」に保存することですが、もっと別の場所に整理する流儀の人はそれなりにどうぞ。あと、ファイル形式が「プレーンテキスト」である必要があります。TextEdit で保存時にこれが選べない場合は、「フォーマット」を適宜設定してください (分からなければ質問してください)。

次に、「ターミナル」のプログラムを起動して、コマンドが打ち込める窓を出します。これも、ドックに入っていない人はファインダを使ってドックに入れておくことを勧めます。そしてターミナルの窓の中で **irb** コマンドを実行して Ruby 実行系を起動してください (「`%`」はプロンプト文字列のつもりなので打ち込まないでください):

```
% irb
irb(main):001:0>
```

この「`irb` なんとか`>`」というのは `irb` の**プロンプト** (prompt — 入力をどうぞ、という意味の表示) で、ここの状態で Ruby のコードを打ち込めます。

プロンプトの読み方を説明すると、`main` というのは現在打ち込んでいる状態がメインプログラム (最初に実行される部分) に相当することを意味しています。次の数字は何行目の入力かを表し

ています。最後の数字はプログラムの**入れ子** (nesting — 「はじめ」と「おわり」で囲む構造の部分) の中に入るごとに 1 ずつ増え、出ると 1 ずつ減ります。とりあえずあまり気にしなくてよいでしょう。以後の実行例では見た目がごちゃごちゃしないように「irb>」だけを示すことにします。

次に load(ファイルからプログラムを読み込んでくる、という意味です) で sample1.rb を読み込ませます。ファイル名は**文字列** (string) として渡すので'' または""で囲んでください: [5]

```
irb> load 'sample1.rb'
=> true
irb>
```

true が表示されたら読み込みは成功で、ファイルに書かれているメソッド triarea が使える状態になります。成功しなかった場合は、ファイルの置き場所やファイル名の間違い、ファイル内容の打ち間違いが原因と思われるので、よく調べて再度 load をやり直してください。

なぜわざわざ 3〜4 行程度の内容を別のファイルに入れて面倒なことをしているのでしょうか? それは、メソッド定義の中に間違いがあった時、定義を毎回 irb に向かって打ち直すのでは大変すぎるからです。このため、以下でもメソッド定義はファイルに入れて必要に応じて直し、irb では load とメソッドを呼び出して実行させるところだけを行う、という分担にします。

load が成功したら triarea が使えるはずなので、それを実行します:

```
irb> triarea 8, 5
=> 20.0
irb> triarea 7, 3
=> 10.5
irb>
```

確かに実行できているようです。irb は quit で終わらせられます:

```
irb> quit
%
```

苦労のわりにはあんまり大したことはない感じでしたが、まあ初心者の第 1 歩ということで、着実に進んでいきましょう。

演習 1-1 例題の三角形の面積計算メソッドをそのまま打ち込み、irb で実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 1-2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 1-3 次のような計算をするメソッドを作って動かせ。[6]

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。
b. 「%」という演算子は**剰余** (reminder) を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。
c. 円錐の底面の半径と高さを与え、体積を返す。
d. 実数 x を与え、x を 10 で割った結果を返す。また、同様だが x の 0.1 倍を返す。これらを比較し、何か気がついたことがあれば述べよ。

[5] 本来ならメソッドに渡すパラメタは丸かっこで囲むのですが、Ruby では曖昧さが生じない範囲でパラメタを囲む丸かっこを省略できます。本資料ではプログラム例の丸かっこは省略しませんが、irb コマンドに打ち込む時は見た目がすっきりするので丸かっこを適宜省略します。

[6] 1 つのファイルにメソッド定義 (def ... end) はいくつ入れても構わないので、ファイルが長くなりすぎない範囲でまとめて入れておいた方が扱いやすいと思います。

e. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、何か気がついたことがあれば述べよ。[7]
f. その他、自分が面白いと思う計算を行うメソッドを作って動かせ。

1.2　基本的な制御構造

ここまでに出てきたアルゴリズムおよびプログラムはすべて「1 本道」、つまり上から順番に実行して一番下まで来たらおしまい、というものでした。単純な計算ならそれでも問題ありませんが、手順が複雑になってくると、実行の流れをさまざまに切り換えていくことが必要になります。この、実行の流れ切り換える仕組みのことを、一般に**制御構造** (control structure) と呼びます。

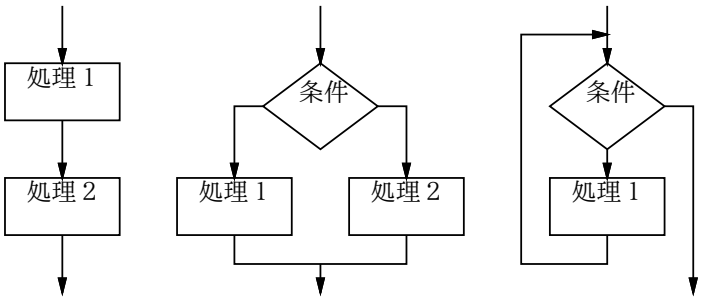


図 1.1: 3 つの基本的な制御構造

制御構造を表現する方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 1.1 にあるような「処理を示す箱」「条件による枝分かれを示す箱」などを矢線でつなげることで、多様な実行の流れを表現します。

流れ図は一見分かりやすそうですが、作成に手間が掛かる、場所をとる、不規則でごちゃごちゃの構造を作ってしまいやすい、という弱点があるため、今日のソフトウェア開発ではあまり使われません。このため本資料でも、流れ図の代わりに擬似コードを主に用いています。

アルゴリズムを記述する時にはさまざまな実行の流れを組み立てますが、今日ではそれらの実行の流れは、図 1.1 に示す 3 つの制御構造を組み合せる形で作り出していくのが普通です:

- **順次実行**ないし**連接** (sequenceing) — 動作を順番に実行していくこと。
- **枝分かれ**ないし**分岐** (branching) — 条件に応じて 2 群の動作のうちから一方を選んで実行すること。
- **繰り返し**ないし**反復** (repetition) — 条件が成り立つ限り一群の動作を繰り返し実行すること。[8]

なぜこの 3 つが基になるかというと、「どんなにごちゃごちゃの流れ図でも、その流れ図と同等の動作をするものを、この 3 つの組み合わせによって作り出すことができる」という定理があり、そのためにこの 3 つさえあればどのような処理の流れでも表現可能だからです。連接については単に動作を並べて書いたものは並べた順番に実行される、というだけなので、以下では残りの 2 つの制御構造をコード上で表現するやり方と、それらを組み合わせてアルゴリズムを組み立てていくやり方を学びます。

[7] x の**平方根** (square root) は **Math.sqrt**(x) で計算できます。
[8] 実行の流れを図示すると環状になるので、**ループ** (loop) とも呼びます。

1.3 枝分かれと if 文

上述のように、枝分かれとは、条件に応じて 2 群の動作のうちから一方を選んで実行するものです。擬似コードでは枝分かれを次のように書き表すものとします (「動作 2」が不要なら「そうでなければ」も書かなくてもかまいません):

- もし ～ ならば、
- 　動作 1。
- そうでなければ、
- 　動作 2。
- 枝分かれ終わり。

Ruby ではこれを **if 文** (if statement) と呼ばれる文を使って表します (右側は「動作 2」のない場合です):

```
if 条件 then
  ... 動作 1 ...
else
  ... 動作 2 ...
end
```

```
if 条件 then
  ... 動作 1 ...
end
```

`then` は Ruby では省略することができます。ただし、「動作 1」を条件と同じ行に書く場合には省略できません。

「条件」については、当面は次の形のものがあると思っておいてください:

- 比較演算 — 「`x > 10`」のように 2 つの値を比べるもの。**比較演算子** (comparison operator) としては、`>`(より大)、`>=`(以上)、`<`(より小)、`<=`(以下)、`==`(等しい)、`!=`(等しくない) がある。[9]
- 条件の組み合わせ — **かつ** (and — ともに成り立つ) を表す「条件1 `&&` 条件2」、**または** (or — 少なくとも片方は成り立つ) を表す「条件1 `||` 条件2」、**否定** (not — ～でない) を表す「`!`条件1」が使える。[10] 複数のかつ、または、否定を組み合わせたり、かっこでくくることもできる。

では具体的な例題として、「入力 x の絶対値を計算する」ことを考えてみます。まず擬似コードを示しましょう:

- abs1: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　そうでなければ、
- 　　$result \leftarrow x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

考え方としては簡単ですね? これを Ruby にしてみましょう:

[9] Ruby では「`!`」は「否定」を表すのに使っています。階乗の記号ではないので注意してください。

[10] Ruby ではさらに演算子として `and`、`or`、`not` も使えますが、結合の強さが記号版と違っていて混乱しやすいので、本資料では使っていません。

```
def abs1(x)
  if x < 0
    result = -x
  else
    result = x
  end
  return result
end
```

実行の様子も示しておきます (0 もテストしていることに注意。作成したコードをテストするときには系統的に洩れなく試してみることが大切です):

```
irb> abs1 8        ←正の数の絶対値は
=> 8               ←元のまま
irb> abs1 -3       ←負の数であれば
=> 3               ←正の数になる
irb> abs1 0        ← 0 の場合も
=> 0               ←元のまま
irb>
```

ところで、同じ絶対値のプログラムを次のように書いたらどうでしょうか?

- abs2: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$-x$ を返す。
- 　そうでなければ、
- 　　x を返す。
- 　枝分かれ終わり。

Ruby 版は次のようになります:

```
def abs2(x)
  if x < 0
    return -x
  else
    return x
  end
end
```

先のとどちらが好みでしょうか? また、別のバージョンとして次のものはどうでしょうか?

- abs3: 数値 x の絶対値を返す
- 　$result \leftarrow x$。
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

「そうでなければ」の部分で何もすることがなければ「そうでなければ」以下を書かないでよいのでしたね。Ruby プログラムも示しておきます (`if` を 1 行に書いてみましたが、Ruby でもこのような時は `then` が必須です):

```
def abs3(x)
  result = x
  if x < 0 then result = -x end
  return result
end
```

3 つのプログラムについて、あなたはどれが好みだったでしょうか?

一般に、プログラムの書き方は「どれが絶対正解」ということはなく、場面ごとに何がよいかが違ってきますし、人によっても基準が違うところがあります。ですから、皆様がこれからプログラミングを学習するに当たっては、自分なりの「よいと思う書き方」を発見していく、という側面が大いにあります。そのことを心に留めておいてください。

演習 1-4 絶対値計算プログラムの好きなバージョンを打ち込んで動かせ。

演習 1-5 枝分かれを用いて、次の動作をする Ruby プログラムを作成せよ。

a. 2 つの異なる実数 a、b を受け取り、より大きいほうを返す。
b. 3 つの異なる実数 a、b、c を受け取り、最大のものを返す。(やる気があったら 4 つでやってみてもよいでしょう。)
c. 実数を 1 つ受け取り、それが正なら「positive」、負なら「negative」、零なら「zero」という文字列を返す。

1.4 繰り返しと while 文

ここまでは、プログラム上に書かれた命令はせいぜい 1 回実行されるだけでしたから、プログラムが行う計算の量はプログラムの長さ程度しかありませんでした。しかし、繰り返しがあれば、その範囲内の命令は何回も反復して実行されますから、短いプログラムでも大量の計算を行わせられます。

まず、繰り返しの最も一般的な形である、条件を指定した繰り返しの擬似コードは次のように書き表すものとします: [11]

- ～ である間繰り返し、
- 　動作 1。
- 繰り返し終わり。

この形の繰り返しは、Ruby では **while 文** (while statement) として記述します:

```
while 条件 do
  ... 動作 1 ...
end
```

条件の次にある `do` も、Ruby では省略することができます。ただし、「動作 1」を条件と同じ行に書く場合は省略できません。本資料では `do` は省略しないことにします。

多くのプログラミング言語でこのような繰り返しは while というキーワードを用いて表すので、このような条件を指定した繰り返しのことを **while ループ** (while loop) とも呼びます。while ループは形だけなら if 文より簡単ですが、慣れるまではどのように実行されるかイメージが湧かない人が多いと思います。while ループの実行のされ方は、次のようなものだと考えてください:

- 「～」を調べる (成立)。
- 動作 1 を実行。

[11]「～」のところには条件を記述しますが、ここに書けるものは if 文の条件とまったく同じです。

- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- …
- 「〜」を調べる (不成立)。
- 繰り返しを終わる。

つまり、条件を調べ、成り立てば動作 1 を実行し、また条件を調べ、…のように繰り返していき、条件が成り立たなくなると繰り返しを終わります。

while を使った簡単な例として、1.0 を繰り返し 2 で割って行きその結果を表示する、というものを示しましょう。

```
def testdiv2
  x = 1.0
  while x > 0.0 do
    puts(x)
    x = x / 2.0
  end
end
```

無限に繰り返すような気がしますか? 次々に半分にしていくと、最後は浮動小数点表現で表せる限界の小さい数になって、さらに半分にすると近似値として「0.0」になるので止まるわけです。

```
irb> testdvi2
1.0
0.5
...
4.0e-323
2.0e-323
1.0e-323
5.0e-324
=> nil
```

つまり、ここで使われている浮動小数点表現では、0 でない最も小さい数というのはおよそ「10^{-324}」くらいであることが分かるわけです。

1.5　計数ループ

while 文は「任意の条件が満たされる間」という極めて基本的なループ機能を提供しますが、実際のプログラムではもっと限定された「0, 1, 2, ..., 9 まで繰り返す」という形のループをよく使います。これは数えるための変数**カウンタ** (counter) と while 文を使って、次のように書けます。

```
i = 0          # i はカウンタ
while i < n do # 「n 未満の間」繰り返し
  ...          #  ここでループ内側の動作
  i = i + 1    # カウンタを 1 増やす
end
```

このように指定した上限まで数を数えながら繰り返していくような繰り返しを**計数ループ** (counting loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用の機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文のほうが書きやすくて読みやすいからです)。

Ruby では計数ループ用の構文として **for 文** (for statement) を用意しています。これを使って上の while 文による計数ループと同等のものを書くと次のようになります:

```
for i in 0..n-1 do
  ...
end
```

これは、カウンタ変数 `i` を 0 から初めて 1 つずつ増やしながら `n-1` まで繰り返していくループとなります (多くのプログラミング言語では、計数ループを表すのに for というキーワードを使うので、計数ループのことを **for ループ** (for loop) と呼ぶこともあります)。

せっかく for 文を説明しておきながら恐縮ですが、以下では計数ループを整数値が持つメソッド `times` を使って書くことにします。なぜ for 文でなく `times` を使うかというと、ブロックを受け取るメソッドは Ruby でさまざまな用途に使える便利な仕組みなので、そちらに慣れたほうがよいと思うからです。これはたとえば次のようになります:

```
100.times do
  ...
end
```

この `times` も先の `to_i` などのように「値 x に対して何かをする」メソッドですが、さらにブロック (コードの並び、`do`〜`end` の部分) を受け取るようになっています。そしてそのブロックを数値の回数 (上の例では 100 回) 実行します。ブロックの指定のための `do` は省略できません。[12]

ところで、計数ループの中でカウンタの値 $(0, 1, 2, \ldots)$ を使いたいこともあります。このため、`times` は各繰り返しごとにカウンタ値をブロックにパラメタとして渡してくれます。上の例ではそれを受け取っていませんでしたが、ブロックの冒頭に`|`名前`|`という書き方でパラメタ (の列) を指定することで、このパラメタを受け取ることができます。複数ある場合は`|x, y|`のようにカンマで区切って並べます。たとえば次のようにすると、0 から 99 までの数を次々に出力することができます:

```
100.times do |i|
  puts(i)
end
```

いろいろありましたが、元に戻って擬似コードでは、計数ループを次のように記します。[13]

- 変数 i を 0 から n の手前まで変えながら繰り返し、
- ... `#` ループ内の動作
- 繰り返し終わり。

演習 1-6 次のような、繰り返しを使ったプログラムを作成せよ。

a. 整数 n を受け取り、2^n を計算する。
b. 整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。
c. 整数 n と整数 $r(\leq n)$ を受け取り、${}_nC_r$ を計算する。

$$ {}_nC_r = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1} $$

[12] それで混乱しやすいので、`while` でも `do` を省略しないことにしたわけです。
[13] 擬似コードはあくまでも「擬似」コードであり、これを Ruby で書く時に for 文にするか `times` にするかは特に指定しません。

1.6　演習問題解説 (一部)

1.6.1　演習 1-5a — 枝分かれの練習

演習 1-5a は例題とほとんど同じです。まず擬似コードを見てみましょう:

- max2: 数 a、b の大きいほうを返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$。
- 　そうでなければ、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

Ruby では次のとおり:

```
def max2(a, b)
  if a > b
    result = a
  else
    result = b
  end
  return result
end
```

これも、次のような「別解」があり得ます:

- max2x: 数 a、b の大きい方を返す
- 　$result \leftarrow a$。
- 　もし $b > result$ であれば、
- 　　$result \leftarrow b$。
- 　枝分かれ終わり。
- 　$result$ を返す。

これの Ruby 版は次のとおり:

```
def max2x(a, b)
  result = a
  if b > result then result = b end
  return result
end
```

どちらが好みですか? これもどちらが正解ということはありません。

ところで、「2 数が等しい場合はどうするのか」について皆様の中には迷った人がいると思います。問題には「異なる数」と書いてあるので考えなくてもよいのですが、仮にそれが書いていなかったとします。そうなると、等しい場合について何らかの指示が本来あるべきですよね。たとえば次のものがあり得ます。

- 「等しい場合はその等しい数を返す」
- 「等しい場合は何が返るかは分からない」
- 「等しい数を渡してはならない」

上 2 つの場合は例解のままで OK です (2 番目では何が返ってもよいので、等しい数でもよい)。最後の場合はどうでしょう。次の考え方があり得ます。

(a) 「渡してはならない」以上、渡されることはないのだから、例解のままでよい

(b) 「渡してはならない」値が渡されたのだから、エラーを表示するなどして警告するべき

どちらにも (互いに裏返しの) 利点と弱点があります。(a) の方が簡潔で短く間違いが起きにくいですが、(b) の方が起きるべきでないことが起きていることが分かるので対処が必要な場合には有用です。

で、あなたは発注者 (久野) の注文を受けてこの課題をやっているわけですから、正解は発注者に「どうしますか」と確認することです。そうすれば、どちらにするかは決められるでしょう。勝手に (b) を選んでプログラムを複雑で間違いやすいものにするのはいかがかと思いますし、発注者が「等しい場合はその等しい数を返す」と書き忘れただけだったら目もあてられませんね。

1.6.2 演習 1-5b — 枝分かれの入れ子

演習 2b はもう少し複雑です。まず考えつくのは、a と b の大きいほうはどちらかを判断し、それぞれの場合についてそれを c と比べるというものでしょうか:

- max3: 数 a、b、c で最大のものを返す
- もし $a > b$ であれば、
- もし $a > c$ であれば、
- $result \leftarrow a$。
- そうでなければ、
- $result \leftarrow c$。
- 枝分かれ終わり。
- そうでなければ、
- もし $b > c$ であれば、
- $result \leftarrow b$。
- そうでなければ、
- $result \leftarrow c$。
- 枝分かれ終わり。
- 枝分かれ終わり。
- $result$ を返す。

かなり大変ですね。これを Ruby にしたものは次のとおり:

```
def max3(a, b, c)
  if a > b
    if a > c
      result = a
    else
      result = c
    end
  else
    if b > c
      result = b
    else
      result = c
```

```
    end
  end
  return result
end
```

こうなると字下げしてないとごちゃごちゃになるでしょう? しかし字下げしてあってもこれはかなり苦しいですね。一般に、if の中に if を入れると非常に分かりづらくなるので、できるだけ避けたほうがよいのです。

ところで、先の別解から発展させるとどうなるでしょう?

- max3x: 数 a、b、c で最大のものを返す
- result ← a
- もし $b > result$ であれば、$result \leftarrow b$。
- もし $c > result$ であれば、$result \leftarrow c$。
- $result$ を返す。

「もし」の擬似コードが 1 行に書かれていますが、この場合はこちらののほうが見やすいと思ったのでそうしてみました。Ruby でも次のとおり (こんどはどちらが好みですか?):

```
def max3x(a, b, c)
  result = a
  if b > result then result = b end
  if c > result then result = c end
  return result
end
```

一般には、枝分かれの中に枝分かれを入れるよりは、枝分かれを並べるだけで済ませられればそのほうが分かりやすいと言えます。また、この方法では入力の数 N がいくつになっても簡単に対処できるという利点があります。

実は、さらなる別解があります。それは、既に max2 を作ったわけですから、それを利用するというものです。

```
def max3xx(a, b, c)
  return max2(a, max2(b, c))
end
```

このように、一度作って完成したものは後から別のものを作る時の「部品」として使える、というのは重要な考え方です。このことも覚えておいてください。

1.6.3　演習 1-5c — 多方向の枝分かれ

演習 2c は 3 通りに分かれるので、if の中にまた if が入るのはやむをえないはずです。Ruby コードを見てみましょう:

```
def sign1(x)
  if x > 0
    return "positive."
  else
    if x < 0
      return "negative."
    else
```

```
        return "zero."
      end
    end
  end
```

このような「複数の条件判断」はよく使うので、実はこれは `if` の入れ子にしなくても書けるようになっています。具体的には、`if` 文には「`elsif` 条件 `then` 動作」という部分を途中に何回でも入れられ、それを使うと次のようになります:

```
def sign2(x)
  if x > 0
    return "positive."
  elsif x < 0
    return "negative."
  else
    return "zero."
  end
end
```

順序が前後しましたが、擬似コードだと次のようになります:

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　「positive.」を返す。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」を返す。
- そうでなければ、
- 　　「zero.」を返す。
- 枝分かれ終わり。

「そうでなくて〜ならば、」は何回現われても構いません。また、そのどれもが成り立たない場合は「そうでなければ」に来るわけですが、この部分は不要なら無くても構いません。

ところで、最大値の問題にちょっと戻ると、複合条件を使えば「`a > b && a > c`」なら `a` が最大だと分かりますから、これを利用した 3 方向枝分かれで書くこともできます (変数を使わず値を返すスタイルにしてみました):

```
def max3y(a, b, c)
  if a > b && a > c
    return a
  elsif b > c
    return b
  else
    return c
  end
end
```

ただし、この方法でも N が 4、5 と増えてくると条件の中の比較演算が増えて、一般に N^2 に比例してしまいます。だからいけないというわけではなく、N の個数が多くなければ、このやり方を使ってもよいかも知れません。

1.6.4 演習 1-6a〜c — 繰り返し

この辺は簡単なのでプログラムだけ示します (べき乗は計算するだけなら `2**n` でよいのですが、繰り返しを使うという前提なのでループを使います):

```
def pow2(n)
  result = 1
  n.times do result = result * 2 end
  return result
end

def fact(n)
  result = 1
  n.times do |i| result = result * (i+1) end
  return result
end
```

階乗の方は「1 × 2 ×…× N」を計算したいわけですが、`times` が渡して来るカウント値は「0, 1, …, N-1」なので、全部 1 足してから掛けています。しかしそれはちょっと分かりにくいですね。

実は、N`.times` の代わりに N`.step(`M`,` d`)` という別のメソッドを使うと、初項 N、終項 M、増分 d を指定して計数ループを作ることができます (d は指定しないと「1」が使われます)。これを使えば、階乗は次のようにもっと分かりやすくなります。

```
def factx(n)
  result = 1
  1.step(n) do |i| result = result * i end
  return result
end
```

上の `step` は「`i` を 1 から `n` まで 1 ずつ増やしながら」という擬似コードに対応します。

組み合わせの数を整数で計算できるようにするためには「小さい側から」掛けて・割って・掛けて・割ってのようにしないとうまくいきません。$\frac{4\times5\times6\times7}{1\times2\times3\times4}$ のように並べて左から 1 列ずつ乗算・除算の順で計算するわけです。この順序でやれば、除算が常に割り切れるので、誤差なしで計算できます (浮動小数点で計算してしまうと、誤差が現れるのでいまいちだと思います)。

```
def comb(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (i + (n-r)) / i
  end
  return result
end
```

1.7 検討 プログラムと制御構造/記述の多様性

この章は本来ならゆっくりやる内容を圧縮してあるので、アルゴリズム・プログラム・言語・計算・制御構造という重要な概念が次々に出て来ます。順に検討してみましょう。

アルゴリズムが手順であり、それをコンピュータに与えられるように書き表したものがプログラム、その書き方が言語、という説明は分かったような分からないようなものだと思いますが、あまり悩まずにプログラムをまず書いて動かすところから初めて、後で定義に戻って来るのもよい方法ではないかと思います。

最初にまず簡単な数の計算をおこなうプログラムが出てきますが、これ自体だけではわざわざ面倒なプログラム作りをやってまで実行させたい気がしないと思います。

しかし「枝分かれ」が出て来ると、処理の流れが場合によってさまざまな経路を通り、最後にそれらの経路が合流して結果が返される、という形になります。これによって、「途中は複雑でさまざまな経路で計算するが、最後は求める (ひとことで言い表せるような) 結果が得られる、ということになり、手順をプログラムとしてパッケージ化してコンピュータで自動実行させることの意義が見えてくると思います。なお、この「さまざまな場合の合流」は変数に入った値という形で現れるので、変数の重要性もここで確認できると言えるでしょう。

さらに「繰り返し」が加わると、少量のコードで大量の計算を行わせることが可能になるので、これもまたプログラムの価値を認識してもらうよい題材です。さらに、繰り返しが「毎回同じ状態」であればまったく同じ計算が反復されるだけであまり意味がないので、繰り返しごとに異る (目的の状態により近づく) 計算が行えるようにする、という意味でも変数の価値が実感できるかと思います。

もう 1 つの重要な概念は、同じ動作をするプログラムでも書き方は非常に沢山あり、どれもが同等で「単一の正解」は無い、ということです。日本の初等中等教育では生徒に「単一の正解に早く到達すること」が強く求められますが、大学から先ではそのような 1 つの正解など無い問題が多くなり、そのことに適応できないと大学での学習に支障をきたします。

このこともあり、ここでは絶対値を題材として「まったく同じ動作をするプログラムが何通りも書ける」「どれが好みであるかは人によって違っている」「結局、各自が自分の『美しいプログラム』に対するものさしを作り上げて行くしかない」ことを強調します。その後、最大値の課題によって自分でもこのことを確認してもらいます。この課題は、(1) 入れ子になった制御構造、(2) 書き方の工夫と得失、(3) 作成ずみの機能を利用することの有利さ、(4) 仕様の問題への意識、というさまざまな事柄につながるため、時間をかけてやってもらう価値があります。

また、解説の途中に出て来る elsif (他の言語では if-else の連鎖) はとても有用なので、ぜひ使いこなせるレパートリーに加えて欲しいと思います。ただ、最初から沢山やるとつらいので、解説で追加するようにしているわけです。

一方、繰り返しについてはまず基本的な while ループについて紹介した後で計数ループ (for ループ) に進む順番になっています。授業ではここで数値積分のテーマを扱っているのですが、今回は省略しました。

実際にまず多く使うのは計数ループなので、ここでは最終的には、計数ループが使えるようになってもらえればよいかと思います。つまり、2 のべき乗とか階乗とかの計算がループでできればよくて、この先さらに制御構造をやっていく中でより深くマスターできるものと考えています。

open question

- if による枝分かれについて、この程度の題材であれば、きちんと考えてもらえるでしょうか?
- このような扱い方で「唯一の正解」の呪縛から抜けられるでしょうか?
- 「3 つのうち最大」の課題は欲張りでしょうか? また、入れ子、記述方法の比較、再利用、仕様などへの発展は欲張りでしょうか?
- while ループと for ループを両方学んで欲しいというのは欲張りでしょうか? またそのための題材としてよりよいものは無いでしょうか?
- このように「枝分かれ」「繰り返し」を単独で十分練習させることは良いことでしょうか?

第2章　制御構造(2)/データ型

2.1　制御構造の組み合わせ

ここまでは基本的に制御構造を 1 つ使うプログラムでしたが、少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側にさらに制御構造を入れることになります。たとえば、

- もし〜であれば、
- 　条件〜が成り立つ間繰り返し:
- 　　○○をする
- 　以上を繰り返し。
- 枝分かれ終わり。

だと次のようになるわけです。

```
if ...
    while
        ...
    end
end
```

このように規則に従って要素を組み合わせて行くことで (単に並べるのも組み合わせ方のうち)、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じです。

具体例として、「0〜99 の数を順に打ち出すが、ただし 3 の倍数の時だけは fizz と打ち出す」という例を考えてみます: [1]

- fizz1: 3 の倍数の時だけ fizz
- 　変数 i を 0 から 100 の手前まで変えながら繰り返し、
- 　　もし i が 3 の倍数ならば、
- 　　　「fizz」と出力。
- 　　そうでなければ、
- 　　　i を出力。
- 　　枝分かれ終わり。
- 　以上を繰り返し。

これを Ruby に直したものは次のようになります。

[1] 海外で古くからある言葉遊びに **fizzbuzz** というのがあります。これは輪になって「$1, 2, \ldots$」と順に数を唱えますが、ただし数が 3 の倍数なら「fizz」、5 の倍数なら「buzz」、3 と 5 の公倍数なら「fizzbuzz」と (数の代わりに) 言わなければならず、間違えたりつっかえたりしたら負けで輪から抜ける、というものです。日本で有名なのは世界のナベアツの「3 の倍数と 3 がつく数字の時だけアホになります」というネタですが、ナベアツも fizzbuzz をヒントにこのネタを考案したという説があります。

```
def fizz1
  100.times do |i|
    if i % 3 == 0
      puts('fizz')
    else
      puts(i)
    end
  end
end
```

動かしてみましょう。

```
irb> fizz1
fizz
1
2
fizz
(途中略)
97
98
fizz
=> 100
irb>
```

このように、基本的な制御構造を組み合わせていけば、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じだと考えてください。

演習 2-1 上の fizz プログラムを打ち込んでそのまま動かせ。動いたら、繰り返しと枝分かれを組み合わせて次の動作をする Ruby プログラムを作成せよ。

a. 0 から 99 までの数のうち、2 の倍数でも 3 の倍数でもないものだけを順に打ち出す。
b. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数の時は fizz、5 の倍数の時は buzz、3 の倍数かつ 5 の倍数の時は fizzbuzz と (いずれも数値の代わりに) 打ち出す (fizzbuzz 問題)。[2]
c. 0 から 99 までの数を順に打ち出すが、ただし 3 の倍数と 3 がつく数字の時は数値の代わりに aho と打ち出す。

演習 2-2 2 数 a、b の**最大公約数** (greatest common divisor) を求めるアルゴリズムを次に示す:

- gcd1: 整数 x、y の最大公約数を返す
- 　x ≠ y である間繰り返し、
- 　　x ＞ y なら、
- 　　　x ← x - y。
- 　　そうでなければ、
- 　　　y ← y - x。
- 　　枝分かれ終わり。
- 　繰り返し終わり。

[2] fizzbuzz 問題については、「(米国で) プログラマを募集して応募者にこの問題のプログラムを書かせてみたら書けない奴が多い。だから応募者のふるい分けに使っている」という話があります。本当だとしたら、これを書けた人はプロ級かも (そんなわけはない)。

- x を返す。

これを Ruby プログラムにして動かせ。これで最大公約数が求まる理由も併せて説明すること。(ヒント: $x > y$ ならば $gcd(x, y) = gcd(x - y,\ y)$ 等のこと (つまり x と y の大きいほうから小さいほうを引いても 2 数の最大公約数は変化しないこと) を示せばよいわけですね。)

演習 2-3 「正の整数 N を受け取り、N が素数か否かを (`true`/`false` で) 返す Ruby プログラム」を書け。まず擬似コードを書き、それから Ruby に直すこと。(ヒント: N が素数ということは、N を 2〜$N-1$ のいずれで割っても割り切れない、つまり剰余が 0 でないということ。剰余は演算子`%`で計算できるのでしたね。)

演習 2-4 「正の整数 N を受け取り、N 以下の素数をすべて打ち出す Ruby プログラム」を書け。待ち時間 10 秒以内でいくつの N まで処理できるか調べて報告せよ。N が大きくなるように工夫してくれるとなおよい。(ヒント: 処理を速くするためには、(1) 割ってみる数をできるだけ少なくとどめる、(2) 素数の候補とする数をできるだけ少なくとどめる、という 2 点を工夫するとよいでしょう。たとえば 2 は別扱いして奇数だけ扱うなど。)

2.2 さまざまなデータ型

2.2.1 基本データ型

コンピュータではさまざまなデータを 0/1 の列として表現して扱っています。もとのデータが何であるかによって、どのような表現を使うかを適宜選択する必要があります。実際には、プログラムを書く時はプログラミング言語で記述するので、プログラミング言語が提供している表現方法をそのまま利用したり、組み合わせて利用したりしてデータを表現します。この、「表現の種類」ないし「データの種類」のことを**データ型** (data types) と呼びます。

多くの言語では、内部に構造を持たないデータ型である**基本データ型** (primitive data types) を別扱いしています。Ruby はそのような別扱いをしない言語ですが、他の言語で言う基本データ型に相当するよう種類のデータ型はあるので、関連する型も含めてここで見ておくことにします:

- **整数**型 — 2 進表現で整数を表す。「`123`」など。既に学んだように、Ruby では小さい値には固定ビット数の表現が使われ、それで表現できなくなると複数語を使う表現に切り換えられる。後者は内部構造を持つ型なので基本データ型ではない。
- **実数**型 — 2 進浮動小数点表現。「`3.14`」など。Ruby では 64 ビット IEEE754 形式浮動小数点が使われる。
- **文字列**型。文字の並び。「`'abcd'`」など。文字列も中に文字が複数入るという構造を持つので、基本データ型ではない。
- **記号** (symbol) 型。Ruby 以外では Lisp や Smalltalk など一部の言語にだけ見られる型。Ruby では「`:abc`」のように「`:`」の後に名前を書いたものがその名前に対応する記号**リテラル** (literal) となる。[3] 記号型は「複数の値について、互いに同じか違うか区別できることがだけが必要」な場合に使われる。たとえば、猫と犬と馬は別のものなので、変数に`:cat`、`:dog`、`:horse` のどれかを入れておいて区別を覚えておく、などの使い方が典型的。[4]
- **true**/**false** — **真偽値** (truth value — はい/いいえの値)。これらは多くの言語では**論理型** (Boolean) と呼ばれる型を構成するが、Ruby ではそれぞれ `TrueClass`/`FalseClass` というクラス (後述) に属する値である。
- **nil** — 「値がない」ことを表すのに使う目印の値。

[3] リテラルとは「そのまま」という意味で、プログラミング言語では「値を直接書いたもの」を意味します。たとえば `3.14` や`'abc'` はそれぞれ数値リテラル、文字列リテラルです。

[4] 文字列 s に対して s`.intern` というメソッドを呼び出すことで s を名前として持つ記号が得られ、この方法なら空白などを含む記号も作れます。逆に y が記号のとき、メソッド y`.to_s` で y の名前の文字列が得られます。

2.2.2　構造を持ったデータ型

構造を持ったデータ型 (structured data type) ないし**複合データ型** (compound data type) とは、その中に基本データ型を複数個含みえるような種類のデータ型を言います (図 2.1)。以下ではまずプログラム言語としての一般論を説明して、それから Ruby の場合を説明します。

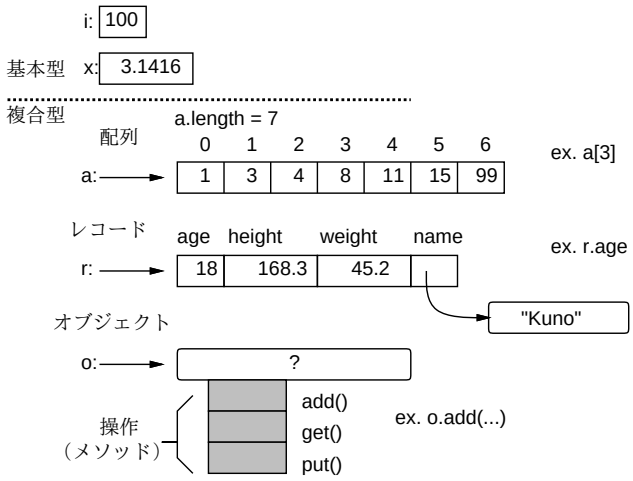


図 2.1: さまざまなデータ型

- **配列**型 — (多くの言語では) 同種類の値が並んだもの。[5]
- **レコード** (record) 型 — 複数の型の値を組にしたもの。それぞれの (中に含まれている) 値を**フィールド** (field) と呼び、名前で参照できる。フィールド参照の前に「`.`」を置く言語が多い。[6]
- **オブジェクト** (object) 型 — 内部的に (レコード型のように) データを保持しているが、外部から内部のデータに直接アクセスするのではなく、**操作** (operation) ないし**メソッド**を呼び出してその機能を利用するようなもの。オブジェクトの機能をサポートするプログラミング言語を**オブジェクト指向言語** (object-oriented language) と呼ぶ。(多くのオブジェクト指向言語では、オブジェクト o に対して $o.method(\cdots)$ でメソッド呼び出しを指定します。Ruby のようにパラメタがない場合は丸かっこも省略できる言語も複数あります。オブジェクト型とレコード型は「内部に値を保持する」という点で似ているので、多くのオブジェクト指向言語ではオブジェクト型とレコード型を統合しています。Ruby もそのような言語の 1 つです。具体例については次の回に扱います。)

図 2.1 を見て不思議に思ったことはないでしょうか。具体的には、基本型 (整数等) では変数の位置に「箱」が書かれていてそこに値が入っていますが、オブジェクト型 (配列等) では少し離れたところにデータを入れる場所があって、変数からはそこに矢印が出ていますね。実はこの矢印はデータのありかを示す**参照** (reference — ありかを指す値で、実体はメモリ上の番地だと思ってよい) です。そして、レコードのフィールド `r.name` に文字列を入れるとすると、実際には文字列は別の場所に入っていて、フィールドにはその場所への参照が入っているわけです。

この「値と参照の区分」はまた後でも出てきますが、とりあえず「`a = b`」のように変数間で代入をした時、基本型では値 (箱の中身) がコピーされますが、オブジェクト型では参照 (矢印) がコピーされるだけで、本体は 1 つのまま (単に 2 つの変数が同じ場所を指すだけ)、と考えてください。

---

[5] 数学の x_i(添字つき変数) みたいなものと考えてください。詳しくは後述します。

[6] たとえば `h` という変数が人のデータを表すレコード型なら、`h.name` には名前 (文字列)、`h.age` には年齢 (整数) が入っている、というふうに使います。

または、全部が矢印であると考えておいても構いません。実は、整数や実数の「中身を変更する」方法はないので、どちらで考えても同じことなのです。しかし、単純な値は箱の中に書いた方が判りやすいので、ここではその方針で説明しています。

さらに、「2 つの変数が同じ場所を指している」状態でそのオブジェクトの中身を書き換えると、もちろんオブジェクトは 1 つだけなので、どちらの変数から見たオブジェクトも同じように変化していることになります。このあたりの挙動は勘違いしやすいので注意が必要です。

2.3 配列

2.3.1 配列の生成

上述のように「配列」とは、「同種のデータを沢山ならべたもの」という意味です。Ruby では値の種別を制約しないので、同種でなくてもよいのですが、先に書いたように配列は x_i のような添字つき変数として使うので、添字によって値がまったく別種のもの、というのは扱いづらく、結局同種のものを入れるのが普通です。

配列を使うには、まず配列を作り出す必要があります。その方法が色々ありますので、ここではそれらについて説明しておきます。

```
a = [1, 2, 3]           # 直接指定
a = Array.new(100, 0)  # 要素数と初期値
a = Array.new(100) do 0 end  # 要素数とブロック
a = Array.new(100) do |i| 2*i end  # 〃
```

1 番目の方法はこれまでにも使ってきた、各要素を直接指定する方法です。[7] この方法は、比較的少数の値を用意する場合に使います。

2 番目は、要素数と初期値を指定する方法で、要素数の多い配列を用意するときにはこの方法が一番単純です。[8]

3・4 番目も、要素数と初期値を指定する方法ですが、初期値として値を計算するブロック (do〜end) を指定するところが違います (この場合はブロックの中で式を直接指定します。メソッドではないので return は書けません)。0 などと定数を指定した場合は 2 番目と変わりませんが、ブロックは (times などと同様)「何番目」というパラメタを受け取ることができるので、それを用いて計算により初期値を決めてもよいのです。

配列は後からメソッド `push` で要素を追加できます。たとえば上の例の 4 番目と次は同じ結果になります。

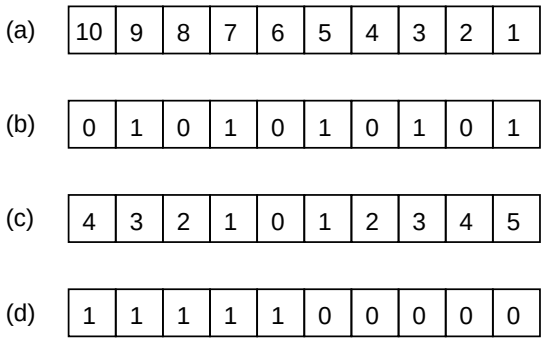
```
a = []                          # 0 要素の配列を作り
100.times do |i| a.push(2*i) end  # 0〜198 を追加
```

なお、現在の**配列の長さ** (array length) ないし要素数は、メソッド `length` で取得できます。上の例では `a.length` は 100 です。

演習 2-5 ブロックを指定する形で 10 要素の配列を生成し、初期値を (a)〜(d) のようにしなさい。

[7] 値を並べて書く方法は「そのまま値を書く」ことから「配列リテラル」と呼ぶこともあります。しかし、配列では初期値を指定するのに変数や任意の式を指定できるので、厳密に言えば「そのまま」ではありません。Ruby の用語でもこの書き方は**配列式** (array expression) というのが正式な呼び方です。

[8] 初期値を指定しないと各要素の初期値は `nil` になります。



なお、初期値を指定するブロックの中でも if を使えます。

```
if 条件 then 式1 else 式2 end
```

この if は「if 式」であり、条件の成否に応じて「式 1」「式 2」のいずれかが全体の値となります。

2.3.2　配列の利用

いちど用意してしまえば、配列の個々の要素は 1 つの変数と同様に扱えます。ここで「どの要素か」を指定するのに `[...]` の中に式を書いて指定します。これを**添字** (index) と呼びます。たとえば上の例だと `a[0]`〜`a[99]` という要素があることになります (0 番目から数えることは慣れないと忘れやすいので注意してください)。

また、Ruby ではまだ用意していない添字番号 (たとえば上で「100 番」とか) の要素を参照すると `nil` が返ります。飛び離れた添字番号 (たとえば上で「200 番」とか) に値を格納すると、そこまでの途中の要素は全部 `nil` で埋められます。

では、配列を与えてその合計を求めるというのをやってみましょう (合計は積分とかで散々やったので簡単ですね):

- arraysum：配列 a の数値の合計を求める
- 　sum ← 0。
- 　i を 0 から配列要素数の手前まで変えながら繰り返し、
- 　　sum ← sum + a[i]。
- 　繰り返し終わり。
- 　sum を返す。

Ruby コードは次のとおり:

```
def arraysum(a)
  sum = 0
  a.length.times do |i|
    sum = sum + a[i]
  end
  return sum
end
```

一応、動かすところの様子を示します:

```
irb> arraysum([1,2,3,4,5])
=> 15
```

実は Ruby では「配列の各要素を取りながら周回するループ」というのもあって、そのほうが少し簡単になります。コードだけ示しておきます:

```
def arraysum1(a)
  sum = 0
  a.each do |x|      # x に配列の各要素が順次入る
    sum = sum + x
  end
  return sum
end
```

合計ならこのほうが少し簡単ですが、「何番目」を必要とする場合もあるので、その場合には計数ループを使うことになるでしょう。[9]

演習 2-6 上記の配列合計プログラムの好きな方をそのまま打ち込んで動かせ。動いたらこれを参考に下記のような Ruby プログラムを作れ。[10]

a. 数の配列を受け取り、その最大値を返す。
b. 数の配列を受け取り、最大値が何番目かを返す。なお先頭を 0 番目とし、最大値が複数あればその最初の番号が答えであるとする。
c. 数の配列を受け取り、最大値が何番目かを出力する。なお先頭を 0 番目とし、最大値が複数あればそれらをすべて出力する。
d. 数の配列を受け取り、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。
e. 数の配列を受け取り、その内容を「小さい順」に並べて出力する (例: 4, 11, 5, 1 → 1, 4, 5, 11)。

演習 2-7 「素数列挙」の問題は、配列を使うとより高速にできる可能性がある。次の 2 つの方針を用いたプログラムを作成し、これまでに作ったものと速度を比較せよ。

a. 素数は値の大きいところではまばらにしかないので、これまでに見つかった素数を配列に覚えておき、新たな素数の候補をチェックする時に「これまで見つかった素数で割ってみて割り切れなければ素数」という方針にすれば、チェックする回数がかなり少なくできる。
b. まったく別の考え方として、N 未満の素数を打ち出すのに次の方針を用いるのはどうだろう: [11][12]

- 論理値が並んだ要素数 N の配列を作り、全部「真」に初期化。
- 2、4、6、…と、2 の倍数番目の部分を「偽」に変更。
- 3、6、9、…と、3 の倍数番目の部分を「偽」に変更。
- 同様に、素数の倍数番目を「偽」に変更していく。
- 最後に、「真」で残っているところを順に調べ何番目かを出力。

2.4 演習問題解説 (一部)

2.4.1 演習 2-1 — fizzbuzz

演習 1 は繰り返しと枝分かれの基本的な組み合わせなので、さっさと Ruby コードを示します。まず 2 の倍数と 3 の倍数以外を打ち出すものから:

[9] メソッド `each_index` で配列の添字を順次取り出してループすることもできます。
[10] 「返す」の場合は上の例と同様に `return` を使い、「出力する」の場合は `puts` を使って画面に直接 (その場で) 出力させてください。`return` は使った瞬間にそのメソッド呼び出しは終ってしまうので、複数回 `return` を使うことはできません。
[11] これは「方針」であって、まだ擬似コードでもないことに注意してください。
[12] この方法を考案したのはギリシャの哲学者**エラトステネス** (Eratosthenes) であり、この方法を彼の名前を冠して**エラトステネスのふるい** (sieve of Eratosthenes) と呼びます。なぜ「ふるい」かというと、素数でないもの (各数の倍数) をふるい落としてしまうと、残ったものは素数だ、という方針でできているからです。

```
def fizz2
  100.times do |i|
    if i % 2 != 0 && i % 3 != 0
      puts(i)
    end
  end
end
```

条件が読みにくいかもしれませんが、「2 の倍数でなく、3 の倍数でもないもの」を打ち出すと考えれば、これでよいと分かります。

別案として、条件を変換するかわりに「素直に」枝分かれしてしまい、打ち出さないのは「何もしない」という案もあります。

```
def fizz2
  100.times do |i|
    if i % 2 == 0
      # do nothing
    elsif i % 3 == 0
      # do nothing
    else
      puts(i)
    end
  end
end
```

「何も書いてない」のは不安なので、「何もしない」というコメントを書いてあります。この方が分かりやすいでしょうか。

次に演習 2-1b ですが、これは if-else の連鎖で「3 の倍数」「5 の倍数」「3 と 5 の公倍数 (15 の倍数)」「それ以外」の 4 つに場合分けするのが一番素直です:

```
def fizzbuzz1
  100.times do |i|
    if i % 15 == 0
      puts('fizzbuzz')
    elsif i % 3 == 0
      puts('fizz')
    elsif i % 5 == 0
      puts('buzz')
    else
      puts(i)
    end
  end
end
```

なぜ 15 の倍数を最初に調べているのか分かりますか。それは、else-if の連鎖では上から順に条件を調べていくので、先に「3 の倍数」や「5 の倍数」を調べてしまうと、15 の倍数は 3 や 5 の倍数でもあるので条件が成り立ってその枝が選ばれてしまい、15 の倍数だけの枝には決して来なくなってしまうからです。

ところで、せっかく「3 の倍数なら fizz」「5 の倍数なら buzz」「両方の倍数なら fizzbuzz」となっているのだから、両者をうまく組み合わせたいと思う人もいるかもしれません。そのようなコー

ドも作ってみましょう: [13]

```
def fizzbuzz2
  100.times do |i|
    num = i
    if i % 3 == 0
      print('fizz'); num = ''
    end
    if i % 5 == 0
      print('buzz'); num = ''
    end
    print(num); puts
  end
end
```

考え方としては、変数 num に打ち出す数を入れておきますが、3 の倍数なら fizz、5 の倍数なら buzz を打ち出すとともに num には空っぽの文字列を入れ直すので、最後の print で何も出力しなくなります。3 や 5 の倍数でないなら num には i が入っているので、そのまま出力されます。パラメタなしの puts は最後に行換えをするためです。

ところで、このコードはよくできている、と思いますか? 個人的には、このコードは先のコードより分かりにくいので、好きではありません。2 つの if の切れ目のところに合流がありますし、最後の数の出力を抑制するために変数 num を使ったりして、流れを追うのが難しくなっています。そんなことをするより、4 方向に枝分かれしてそれぞれの場合を明快に処理する先のコードのほうがずっと読みやすくてスマートだと思うのですが、どうでしょうか?

最後は「世界のナベアツ」ですが、「3 がつく数」はどうしましょうか。それは、対象とする数が 1 桁または 2 桁なので、「3 がつく」というのは 1 桁目が 3 か、2 桁目が 3 という意味になります。1 桁目が 3 というのは、10 で割った余りが 3 ということですし、2 桁目が 3 というのは、10 で切捨て除算した結果が 3 ということですね。ここまで分かればあとは書くだけです:

```
def fizz3
  100.times do |i|
    if i % 3 == 0 || i % 10 == 3 || i / 10 == 3
      puts('aho')
    else
      puts(i)
    end
  end
end
```

2.4.2 演習 2-2 — 最大公約数

課題の擬似コードを Ruby に直したものは次のとおり:

```
def gcd1(x, y)
  while x != y
    if x > y
      x = x - y
    else
```

[13] print は puts と同様に文字列や数値を打ち出すけれども、行換えはしないメソッドです。なぜこれを使うかというと、fizz と buzz をくっつけて 1 行に打ち出すためです。

```
      y = y - x
    end
  end
  return x
end
```

なぜこれで最大公約数が求まるのでしょうか? 次のように考えてください (x と y は正の整数であるものとします):

- $x = y$ であれば、$gcd(x, y)$ は x そのもの。当然ですね。
- $x > y$ であれば、$gcd(x, y)$ は $gcd(x - y, y)$ に等しい。[14]
- したがって、$x - y$ を改めて x と置いて $gcd(x, y)$ を求めればよい。
- $x < y$ の場合も同様。
- この手順の反復ごとに、x または y のどちらかがより小さくなるが、0 以下にはならない (大きい方から小さい方を引くから)。
- ということは、この反復は有限回で止まる。
- ということは、そのとき $x = y$ が成り立ち、x が一番最初の x と y の最大公約数に等しい。

繰り返しを使うときは「必ず止まって、止まった時には求める状況が成り立っている」ように設計する、という感じがお分かりになりましたか?

2.4.3　演習 2-3 — 素数判定

素数の判定ですが、擬似コードは次のとおり:

- isprime1: N が素数か否かを返す
- 　　sosu ← 「真」。
- 　　i を 2 から $N - 1$ まで変化させながら繰り返し:
- 　　　　もし N が i で割り切れるならば、sosu ←「偽」
- 　　繰り返し終わり。
- 　　sosu を返す。

変数 sosu は先に「はい」を表す true を入れておきます。そして素数でないと思ったら「いいえ」を表す false 入れ、最後に結果がどちらか見ます。このような使い方の変数のことを**旗** (flag) と呼びます。最初「旗」が立っていて、後で見たら「旗」が降りていたとすれば、誰が降ろしたかは分からないとしても、少なくとも誰かが旗を降ろしたことは確実に分かるわけです。では Ruby コードを見てみましょう:

```
def isprime1(n)
  sosu = true
  2.step(n-1) do |i|
    if n % i == 0 then sosu = false end
  end
  return sosu
end
```

[14] 証明: 最大公約数を G と置くと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍です。ということは、G は $x - y$ と y の公約数です (最大かどうかはまだ分かりません)。ところで、もし最大公約数で「なかった」とすると、最大公約数 $H(> G)$ が別にあるわけで、H は y の約数かつ $x - y$ の約数になります。ということは、H は $x - y + y = x$ の約数でもあります。これは G が x と y の最大公約数であるということに矛盾します。したがって G は $x - y$ と y の最大公約数でもあることになります。

2.4.4 演習 2-4 — 素数列挙とその改良

もっとも素朴な素数列挙プログラムは、上の素数判定を利用すれば簡単にできます。Ruby のコードを直接示しましょう:

```
def primes1(n)
  2.step(n-1) do |i|
    if isprime1(i) then puts(i) end
  end
end
```

これを手もとのマシンで動かしてみましたところ、10 秒間でおよそ 17,000 まで調べられました。これはあまり速くはないです。ところで、先の素数判定 `isprime` は「割り切れる」と分かってもそこでやめないで `n` の手前までずっと割っていきますから、早い段階で割り切れた数に対しては非常に無駄が大きいと思われます。そこで、改良版を作ってみました:

```
def isprime2(n)
  2.step(n-1) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

こちらは割り切れると分かったら直ちに `return` で「いいえ」を返しますから、無駄な割り算はしなくてすみます。上の `primes1` をこちらを使うように直したところ、10 秒間で 50,000 くらいまで調べられました。つまり速度が 3 倍くらいになったわけです。

さらに考えると、割り算は $N-1$ までやる必要はなく、$\sqrt{N}$ まで調べて割り切れなければそれ以上やっても割り切れないと分かります ($\sqrt{N}$ よりも大きい因数があるなら、小さい因数もあるはずですから)。そこで素数判定を次のように改良します:

```
def isprime3(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
```

これで試してみると、10 秒間で 800,000 くらいまで調べられました。

次に、2 は素数であり、2 の倍数は素数でないことが分かり切っているので調べる必要がない、ということを利用しましょう。このため、3 以上の奇数だけで割ってみる「改編版」の素数判定を作って使います。

```
def isprime4(n)
  3.step(Math.sqrt(n), 2) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes4(n)
  puts(2)
  3.step(n-1, 2) do |i|
    if isprime4(i) then puts(i) end
```

```
    end
  end
```

2 は「別建てで」出力しています (これでも仕様としては合ってるわけです)。こんどは 10 秒間で 1,000,000 くらいまで調べられました。

もうちょっとだけ頑張って、では 2 と 3 より大きい素数は 6 の倍数± 1 だけ (それ以外は 2 と 3 の倍数になってしまう)、ということを利用してさらに調べる数を減らしてみましょう。

```
def isprime5(n)
  6.step(Math.sqrt(n), 6) do |i|
    if n % (i-1) == 0 then return false end
    if n % (i+1) == 0 then return false end
  end
  return true
end
def primes5(n)
  puts(2)
  puts(3)
  6.step(n-1, 6) do |i|
    if isprime5(i-1) then puts(i-1) end
    if isprime5(i+1) then puts(i+1) end
  end
end
```

こんどは 10 秒間で 1,400,000 くらいまで調べられました。コンピュータが高速だといっても、大量に計算する場合にはやはり工夫する価値はあるわけです。

演習 2-5 — 配列の生成

これは簡単にコードだけ示します。

```
Arran.new(10) do |i| 10-i end       #(a)
Arran.new(10) do |i| i%2 end        #(b)
Arran.new(10) do |i| (i-4).abs end  #(c)
Arran.new(10) do |i| if i>=5 then 0 else 1 end end  #(d)
```

(a)、(b) は簡単なクイズという感じですね。(c) は「`if i<5 then 4-i else i-4 end`」でもよいのですが、ここでは数値が持つ絶対値のメソッド「x.abs」を使ってみました。または、以前作った絶対値のメソッドを読んでももちろん構いません。(d) は if で枝分かれするのが一番素直ですね。

演習 2-6 — 配列の演習

擬似コードを略して Ruby のコードだけ記します。まず最大:

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

このような形で配列を使う場合は、ふつうは「とりあえず max に最初の値を入れておき、より大きい値が出てきたら入れ換える」方法になります。each は配列の各要素を順に取り出して来るメソッドでした。

次は最大の値が何番目に出てくるかなので、普通の計数ループにします。また、「何番目か」も変数に記録し、最大を更新した時にに同時に更新します:

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.each_index do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

配列の各添字を列挙するには a.length.times を使えばよいのですが、ここに示したように配列のメソッド a.each_index を使うこともできます。

最大を 1 箇所だけ記録するのは変数でもできましたが、最大が複数あった時にその位置を全部打ち出すのには、(1) まず最大を求め、(2) その最大と等しいものがあったら位置を打ち出す、という形で 2 回ループを使う必要があります:

```
def arraymaxno2(a)
  max = a[0]
  a.each_index do |i|
    if a[i] > max then max = a[i] end
  end
  a.each_index do |i|
    if a[i] == max then puts(i) end
  end
end
```

平均より大きい値を打ち出すのもこれと同様です:

```
def arrayavglarger(a)
  sum = 0.0
  a.each do |x| sum = sum + x end
  avg = sum / a.length
  a.each do |x|
    if x > avg then puts(x) end
  end
end
```

ところで、sum の初期値を 0.0 にしていることに注意。こうすれば、sum の内容は常に実数なので、最後の割り算も実数の割り算になります。ただの 0 だと、配列の中身もすべて整数のときは切捨て除算になってしまって、平均の計算が正しくできません。

演習 2-7 — 配列を使った素数列挙

まず最初は、これまでに見つかった素数を配列に覚えておく方法です: [15]

[15] 配列のメソッド push は配列の末尾に新たな値を追加します。

```
def isprime8(a, n)
  limit = Math.sqrt(n).to_i
  a.each do |i|
    if i >= limit then a.push(n); return true end
    if n % i == 0 then return false end
  end
  a.push(n); return true
end
def primes8(n)
  a = []
  2.step(n-1) do |i|
    if isprime8(a, i) then puts(i) end
  end
end
```

素数判定メソッドは素数の入った配列を受け取り、そこから順に素数を調べて候補の数に対して割り切れるかどうか調べていきます。ただし、候補の数の平方根まで来たらそれ以上やっても割り切れないことが分かるので「素数である」という答えを返します。なお、素数だった時は後に備えて配列にその素数を追加しておきます。この方法だと、「2 や 3 の倍数を除外」などのワザを使っていないのにもかかわらず、手元のマシンで 10 秒間で 1600,000 くらいまでの素数が調べられましした。

ただし、このあたりをやっていると分かりますが (もっと早く気づいた人もいることでしょう)、実は数値を表示するという処理にもかなり手間が掛かっています。計測するという観点からは、表示を省略して内部のチェックだけの時間を計った方がいいでしょう。でも、速さの違いを実感していただくには、画面に出力が出た方が分かりやすいので、課題としては画面に出力するようにしてあります。

もう 1 つ (エラトステネスのふるい) はこれまでと大幅に違う方法です:

```
def primes9(n)
  a = Array.new(n, true);
  2.step(n-1) do |i|
    if a[i] then
      puts(i)
      i.step(n-1, i) do |k| a[k] = false end
    end
  end
end
```

まず最初に、添字が $0 \sim N-1$ の配列 a を作り、各要素の値は true としておきます。次に、2 から始めて各候補の数値 i について、a[i] が true ならそれは素数なので打ち出すとともに、その素数の倍数 k について a[k] を false にします。そうすると、調べて行ってまだ true の要素が素数として次々に拾われていくわけです。

この方法は非常に高速で、10 秒間で 3,000,000 以上の素数がチェックできました。最初の素朴版が千のレベル、こちらが百万のレベルだから、比べると 1000 倍!!!も速くなったわけです。日常的なものの世界では「1000 倍の差」というのはなかなかありません。我々の歩く速度がおよそ 4km/h ですが、4,000km/h というのはジェット機でもだめでロケットの速度ですね。これに対し、プログラムの動作速度については簡単に「ものすごい差」が生まれてしまう世界なわけです。

2.5 検討 制御構造・配列によるロジックの構成

この段階では、「繰り返しの中の枝分かれ」のように制御構造を入れ子にして少し込み入ったロジックを構成する練習をします。題材は fizzbuzz で、この問題は簡単版から少し込み入った版までバリエーションが作りやすいことから取り上げています。また、素数の判定や列挙も繰り返しの中に分岐なのと、こちらは速度を上げる工夫がいろいろできる問題なので、その方面で興味を持ってもらうのに適しています。

また、プログラムの構造はほぼ確定的に「1 重ループの中にこれまで学んで来たような場合分け」となっていますが、これで実際にかなり多くのアルゴリズムは書けるので、構造はそんなに悩まないでロジックに注力してもらうという意味でもよい方向づけだと思います。

この部分の目標は、「制御構造を組み合わせてロジックを考えることができ」「それが実際に思った通りに動作すると嬉しい」という感覚を持って貰うことです。プログラミングは結局、楽しいと思って貰え、好きだからやってみようと思ってもらうことが重要だと考えています。徐々に内容が難しくなるとついて来られなくこともあるとは思いますが、どこの部分までは楽しめた、そこから先はいくらでも深くなることが分かった、という状態で終わりたいわけです。

次にデータ構造の話題がありますが、ここでは配列が分かって扱えるようになることが目標になります。配列 (並び) は非常に汎用性があり、多くのアルゴリズムで活用させるため、他のさまざまなのもよりもまず、配列を使いこなせることが有用だと考えるからです。また、さまざまなデータ構造の中でも一番単純で言語での扱いも楽だからということもあります。

ここではまず、Ruby の特徴であるブロックを用いた配列の生成・初期化を体験してもらい、整数がさまざまに並んだ配列を生成することで親しみを持たせます。次に「普通の」内容に進んで、「合計」「最大」などの標準的な課題を扱いますが、これらの課題はここまでに学んだ制御構造の練習としても適しています。あと、素数列挙を頑張ったことがあれば、「配列に素数を覚えて行く」方法や「エラトステネスのふるい」は楽しめる話題になると思います。

全体として、ここまでで 1 つのメソッドの中に入るロジックはひととおり終わったということになり、プログラミング学習のひと区切りになると言えます。

open question

- 難易度が上がったと思われないでうまく「制御構造の中に制御構造」を体験してもらえる題材になっているでしょうか?
- 構造は「ループの中に場合分け」に絞り、そのパターンを応用することで様々なものが書けるようになってもらう、という戦略はうまく行くでしょうか?
- さまざまなデータ型をひととおり説明するという形は「古典的な説明型」の授業に近い感じがしますが、これくらいなら許されるでしょうか?
- 実際に練習するデータ構造として、まずは配列に絞って始める方法は良さそうでしょうか? また、その着手として配列の初期化から始めるという (Ruby に特化した) 方法はどうでしょう?
- 「合計」「最大」などの標準的な課題で十分に配列のことが理解してもらえるでしょうか?
- 素数列挙の例題は「速さを気にさせる」という点で興味を持たせるようになっていますが、どれくらい有効でしょうか。

第3章　手続き/関数と抽象化

3.1　手続き/関数が持つ意味

手続きないし**サブルーチン**とは、ひとまとまりの動作に名前をつけ、他の箇所からの**呼び出し** (call) により実行できるようなもののことです。

多くのプログラミング言語では、手続きから値を返すことができ、「ある決まった手順で値を計算するもの」とも捉えられます。このため、手続きのことを**関数** (function) と呼ぶ言語もあります。そして既に学んできたように、Ruby では**メソッド**が手続きに相当します。また、既にたくさん使ってきたように、手続き呼び出し時に**パラメタ**を渡すことで、その渡した値に応じた動作や処理を行わせることができます。

たとえば前節では「整数 n が素数かどうかを調べる」というメソッドを作り、「素数を列挙する」メソッドからはそれを呼び出していました。そのコードを少し手直して再掲します:

```
def isprime(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end
def primes(n)
  2.step(n-1) do |i|
    if isprime(i) then puts(i) end
  end
end
```

2 つのメソッドに分けると、何がよいのでしょうか? それは、2 番目のメソッド中で「もし i が素数なら」とひとことで書けるようになることです。

別の例として「n 未満の数で、2 つ違いの素数になっているものを打ち出す」という作業を考えてみます。これも上のメソッドがあれば、次のように書けます:

```
def adjacentprimes(n)
  2.step(n-3) do |i|
   if isprime(i) && isprime(i+2) then puts "#{i} #{i+2}" end
  end
end
```

つまり「もし i が素数で、かつ、i+2 が素数 なら」とひとことで言えます。中では複雑な計算が必要な手順であったとしても、まとめて名前をつけることによって、必要なら何箇所からでも呼び出せ、コードも分かりやすくなるわけです。これをもっと一般的に言うと、手続きによって**抽象化**が行える、ということになります。

抽象化とは、不要な細部を省いて問題の検討に必要なことがらだけを残すことです。たとえば、「n が素数かどうか」を調べる方法が一度分かれば、あとはそれを参照すればよいのであって、その中でどのように処理しているかは「不要な細部」として見ないで済むことが利点なのです。

3.2 手続き/関数と副作用

3.2.1 広域変数と副作用

関数という言葉は数学でも使われますが、数学で言う関数は「入力空間ないし**定義域** (domain) から出力空間ないし**値域** (range) への**写像** (mapping)」であって、同じ入力 (パラメタ) を与えた場合は同じ結果を返します。

たとえば $f(x) = x^2$ であれば、$f(2)$ の値は 4 であり、計算するたびに違うということはあり得ません。ですから、関数の値を 1 回計算して取っておき、2 回目は取っておいた値を利用するのでも、2 回とも計算するのとで、結果は一緒です。

これに対し、プログラムにおける関数は「単なる計算手順」ですから、その計算のやり方によっては、毎回違う値を返すこともありますし、どこかに観測できる変化を及ぼすこともあります。これを一般に**副作用** (side effect) と呼びます。

たとえば一番簡単な例として、`puts` は呼び出すたびに画面に文字が出力されますから、1 回呼び出すのと 2 回呼び出すのでは結果が違います。つまり、**入出力** (input/output — キーボードや画面やファイルなどとの間でのデータのやりとり) は副作用の形で扱われます。また、関数や手続きの中で外部の (関数や手続きの外で定義された) 変数を書き換える場合も副作用になります。

これまで使って来た変数は**局所変数** (local variable) と呼ばれ、そのメソッドが実行されている間だけ存在していて、実行が終わると消滅します (メソッドのパラメタも局所変数の一種と考えられます)。これに対し、プログラムの実行中ずっと存在し続け、さまざまなメソッド中から参照できる変数を**広域変数** (global variable) と呼びます。Ruby では先頭に`$`のついた名前の変数が広域変数です。広域変数は通常、複数のメソッド呼び出しをまたがって値を共有するのに使います。たとえば、次に示すようなやり方でいくつもの値を合計することを考えます。

```
irb> sum 1.5  ←次々に指定した値の
=> 1.5        ←合計が返される
irb> sum 2
=> 3.5
irb> sum 0.8
=> 4.3
irb> reset    ←ご破算もできる
=> 0
irb> sum 2
=> 2
irb> sum 0.7
=> 2.7
```

これを実現するためには、メソッド `sum` と `reset` を作りますが、これらの間で (および複数の `sum` 呼び出し間で) 値を保持するのに広域変数を使うわけです。

```
$x = 0
def sum(v)
  $x = $x + v; return $x
end
def reset
  $x = 0
end
```

この場合、`sum` や `reset` は広域変数`$x` を変更するという形の副作用を持っています。手続きが副作用を持つのは、広域変数に対する書き換えだけとは限りません。たとえば、配列をパラメタとして受け取って、その配列の内容を書き換えた場合、その変更を配列を渡した側にも影響します。このようなものも副作用になります。

3.2.2 例題: RPN 電卓

上述の sum と reset では合計という簡単な計算しかできませんでしたが、もう少し込み入った計算もできる仕組みとして、**逆ポーランド記法** (RPN、Reverse Polish Notation) 電卓というのを作ってみます。私たちが普段書いている数式の書き方は**中置記法** (infix notation) と呼び、演算子が被演算子の間に書かれますね。

$8 + 5 \times 3 \rightarrow 23$
$(8 + 5) \times 3 \rightarrow 39$

この方法は私たちが慣れてはいますが、「演算子の強さ (乗除算を優先)」とか「かっこの中を優先」などの規則があり、実は複雑です。これに対し、(1) 演算子は被演算子の後に書き、(2) 被演算子は演算子のできるだけ近くにある「残っている値」とする、という規則で書くのが RPN です。上の 2 つの例を RPN で書くと次のようになります。[1]

$8\ 5\ 3\ \times\ +\ \rightarrow\ 8\ 15\ +\ \rightarrow\ 23$
$8\ 5\ +\ 3\ \times\ \rightarrow\ 13\ 3\ \times\ \rightarrow\ 39$

上の例からも分かるように、RPN を使って式を記述する場合は、かっこが不要です。

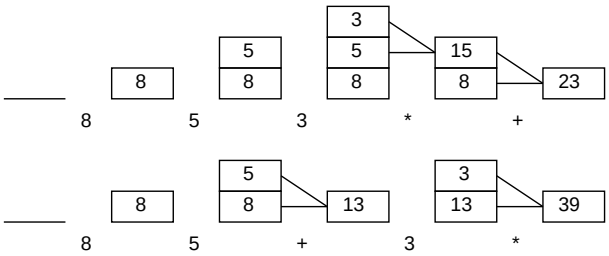


図 3.1: RPN 電卓による計算

RPN を使った計算は、図 3.1 のように値の並びを内部で保持し、数値が出て来た時には値を並びの末尾 (上が末尾です) につけ加え、演算子が出て来た時には最後の 2 つを取って演算し、代わりに結果を最後につけ加えるようにします。そうすると、式の最後まで来た時に並びに残っている値が結果となります。実は Mac の「電卓」は「Command-R」で RPN モードにできるので、少し計算してみると様子が分かります。

さて、先の合計と同様、この RPN 電卓を Ruby で実現してみます。数値の入力は「e」というメソッドで実行し、演算は「add」「mul」をとりあえず用意しました。動かした様子を見ましょう。

```
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> e 3
=> [8, 5, 3]
irb> mul
=> [8, 15]
irb> add
=> [23]
irb> clear
=> []
```

[1] 演算子を「後に」置くことから、**後置記法** (postfix notation) とも呼びます。

```
irb> e 8
=> [8]
irb> e 5
=> [8, 5]
irb> add
=> [13]
irb> e 3
=> [13, 3]
irb> mul
=> [39]
```

ではプログラムですが、並びとしてはもちろん配列を使用します。配列には末尾に値を追加するメソッド「a.push(値)」と末尾から結果を取り除いて返すメソッド「a.pop」が用意されているので好都合です。

```
$vals = []
def e(x)
  $vals.push(x); return $vals
end
def add
  x = $vals.pop; $vals.push($vals.pop + x); return $vals
end
def mul
  x = $vals.pop; $vals.push($vals.pop * x); return $vals
end
def clear
 $vals = []; return $vals
end
```

演習 3-1 「合計を求める」例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加える代わりに指定した値を引く機能 dec(x)。
b. うっかり間違って reset した時にそれを取り消せる機能 undo(undo の undo はできなくてもよいが、できるようにしてもよい)。
c. これまでに加えた(そして引いた)値の一覧を表示した上で合計を表示する機能 list(reset はできた方がよい。reset の undo もできるとなおよい)。

演習 3-2 「RPN 電卓」の例題をそのまま打ち込んで動かしなさい。動いたら、さらに次の機能を実現するメソッドを追加しなさい。

a. 加算と乗算に加えて減算 (sub)、除算 (div)、剰余 (mod) を追加。
b. 現在の演算結果の符号を反転する操作 inv。たとえば「1 2 add inv → -3」となる。
c. 最後の結果とその 1 つ前の結果を交換する操作 exch。たとえば「1 3 exch sub → 2」となる。
d. ご破産の機能 clear と、開始またはご破産から現在までの操作をすべて横に並べて (つまり RPN で) 表示する機能 show。[2]
e. その他、RPN 電卓にあったらよいと思う任意の機能。

[2]show を実現するためには、すべての演算にその操作内容を記録するコードを追加する必要がある。

演習 3-3 2要素の配列を2つ並べた配列を2×2の行列として扱うことを考える。たとえば「`[[1.0, 0.0], [0.0, 1.0]]`」は単位行列であり、一般に「`[[`a`, `b`], [`c`, `d`]]`」は次の行列を表す。

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

a. 2 × 2 行列の RPN 電卓を作れ。加減算、乗算は作ること。

b. さらに、転置行列、逆行列の演算も作ってみよ。

c. 2 × 2 より大きな 3 × 3、できれば一般の $N \times N$ 行列の RPN 電卓を作ってみよ。

3.3 再帰呼び出し

関数や手続きの興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがあります。これを**再帰** (recursion) と呼びます。たとえば、前章でやった内容から、正の整数 x、y について、その最大公約数は次のように定義できます:

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

これにそのまま従って Ruby のメソッドを書くことができます:

```
def gcd(x, y)
  if x == y
    return x
  elsif x > y
    return gcd(x-y, y)
  else
    return gcd(x, y-x)
  end
end
```

プログラムそのものは大変分かりやすいですが、なぜ「堂々めぐり」にならずに計算が終わるのでしょうか。それは、図 3.2 を見れば分かります。

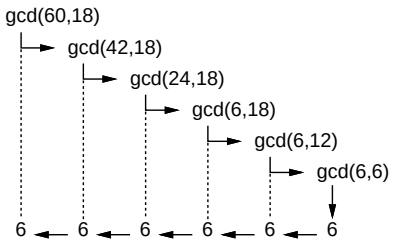


図 3.2: 再帰関数による最大公約数の計算

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従います:

- 問題の「簡単な場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。
- それ以外は問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、少し小さい数の最大公約数問題に変形)。

これがうまくできていれば、堂々めぐりにならずに正しく実行できるわけです。

演習 3-4 上の例題をそのまま打ち込んで動かせ。うまく動いたら、次のような再帰的定義に従った計算を再帰関数として書いて動かせ。また、典型的な実行の様子を表す、図 3.2 のような図を描いてみよ。

a. 階乗の計算。

$$fact(n) = \begin{cases} 1 & (n = 0) \\ n \times fact(n-1) & (otherwise) \end{cases}$$

b. フィボナッチ数。

$$fib(n) = \begin{cases} 1 & (n = 0 \ or \ n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

c. 組み合わせの数の計算。

$$comb(n, r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1,\ r) + comb(n-1,\ r-1) & (otherwise) \end{cases}$$

d. 正の整数 n の 2 進表現。[3]

$$binary(n) = \begin{cases} \text{“0”} & (n = 0) \\ \text{“1”} & (n = 1) \\ binary(n \div 2) + \text{“0”} & (n \text{ が } 2 \text{ 以上の偶数}) \\ binary(n \div 2) + \text{“1”} & (n \text{ が } 2 \text{ 以上の奇数}) \end{cases}$$

3.3.1　再帰呼び出しの応用

3.3.2　再帰呼び出しの興味深い特性

再帰呼び出しの興味深い特性として、「現在実行しているコードと、再帰的に呼び出した自分とは、動作は同一だが (同じプログラムだから当然!)、人格としては別人」だということがあります。たとえば ${}_nC_r$ の計算の様子を図 3.3 に示します。${}_5C_3$ を計算するとして、その「私」は「手下」として ${}_4C_2$ を計算する人と ${}_4C_3$ を計算する人に作業を依頼します。これらの「人」はデータ (n とか r) は「私」とは違っているので別人ですが、動作は「私」と同じわけです。

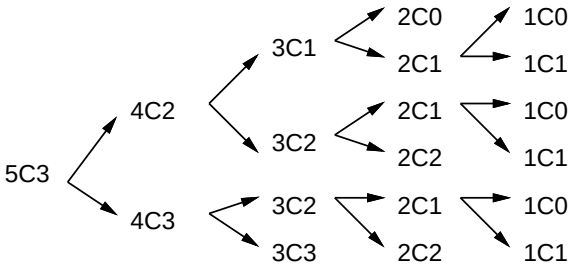


図 3.3: 再帰関数による組合せの数の計算

このような別人格を利用すると、興味深い処理が可能になります。たとえば、1〜3 を打ち出す場合、次のように 1 重ループを使えばできますね。

```
1.step(3) do |i| puts(i) end
```

[3] この場合、関数の返す値は文字列であることと、+ は文字列の連結演算、÷は整数の除算 (切捨て除算) を表していることに注意してください。Ruby では整数どうしの「/」は自動的に切捨て除算になるのでしたね。

では、「1〜3 が 2 つ並んだ全ての組合せ」だと…ループを 2 重にします (to_s は数値を文字列に変換するメソッドで、文字列どうしの＋は「連結」になります)。

```
1.step(3) do |i| 1.step(3) do |j| puts(i.to_s + j.to_s) end end
```

では「3 つ」たど 3 重…一般に n を指定して「1〜3 が n 並んだ全ての組合せ」が作れるでしょうか? プログラムでループを n 個書く方法では、プログラムを生成しない限り無理そうですよね? ところが、次のようにすればできるのです。

```
def nest3(n, s)  # 呼び方: nest3(3, "") ←空文字列を渡す
  if n <= 0 then
    puts(s)
  else
    1.step(3) do |i| nest3(n-1, s + i.to_s) end
  end
end
```

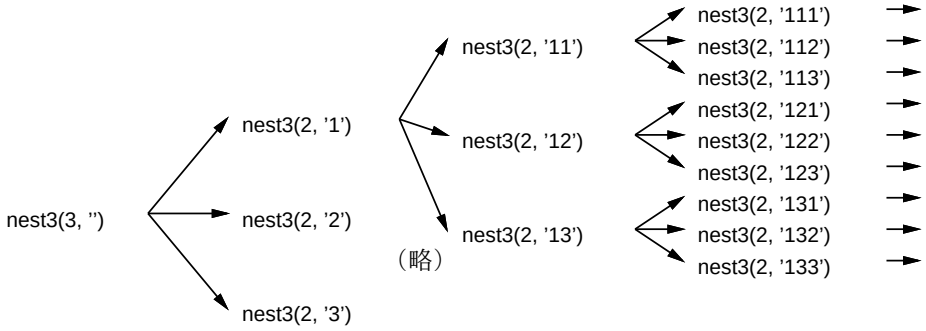


図 3.4: 1〜3 が N 個並んだすべての場合を出力

つまり、それぞれの「私」は自分の担当として 1〜3 を順番に生成し、「親の私」から渡された文字列にそれをくっつけ、「子の私」を呼び出します。入れ子になる (内側の) ループはその「子の私」の中で実行されるわけです。そして 0 の場合は…「文字列を打ち出す」のが仕事になります。この呼び出しの様子を図 3.4 に示します。

3.3.3 再帰呼び出しによる枝分かれ

前述のように、再帰手続きにおいて、自分自身を 2 回以上呼び出す場合、それを用いて、「複数の場合について枝分かれしてそれぞれを処理する」というアルゴリズムが可能になります。

たとえば、「減少列の打ち出し」という問題を考えてみましょう。「5 から始まり、1 ずつ小さくなる整数の列」(ただし 0 にはならない) は…もちろん「5、4、3、2、1」ですね。では「1 または 2 小さくなる列 (1〜2 減少列)」だとどうでしょうか。上に加えて「5、4、3、2」、「5、4、3、1」、「5、4、2、1」等が含まれることになります。それらを「全部」打ち出すにはどうしたらいいでしょうか。

1 つ作業用の配列を用意し、そこに減少列を作成して打ち出すように考えます。その作業を行う再帰手続きのアルゴリズムを示します。

- decr1(n, b) --- 配列 b の末尾に n から始まる 1〜2 減少列を追加し打ち出す
- もし n>1 ならば、
- b の末尾に n を追加。
- b の後ろに n-1 から始まる 1〜2 減少列を追加し打ち出す

- 　　b の後ろに n-2 から始まる 1〜2 減少列を追加し打ち出す
- 　　b の末尾を取り除く
- 　そうでなくて n>0 ならば、
- 　　b の末尾に n を追加。
- 　　b の後ろに n-1 から始まる 1〜2 減少列を追加し打ち出す
- 　　b の末尾を取り除く
- 　そうでなければ、
- 　　b の内容を打ち出す
- 　枝分かれ終わり

このアルゴリズムも再帰的アルゴリズムですから、「n から始まる 1〜2 減少列を追加し打ち出す」ことはできるものとして考えます。そして、たとえば「decr1(5, [])」のように呼び出されたとしたら、まずその 5 は配列に追加して「[5]」とします。そしてその状態で「decr1(4, [5])」と「decr1(3, [5])」をそれぞれ呼び出すわけです。すると、前者は「5、4、…」で始まる 1〜2 減少列、後者は「5、3、…」で始まる 1〜2 減少列を作り出して打ち出すことを担当するわけです。このように、「1 減らす場合」「2 減らす場合」の 2 通りへの枝分かれを、自分自身を 2 回呼び出すことでそれぞれ扱っているわけです。この呼び出しの関係図を図 3.5 に示します。

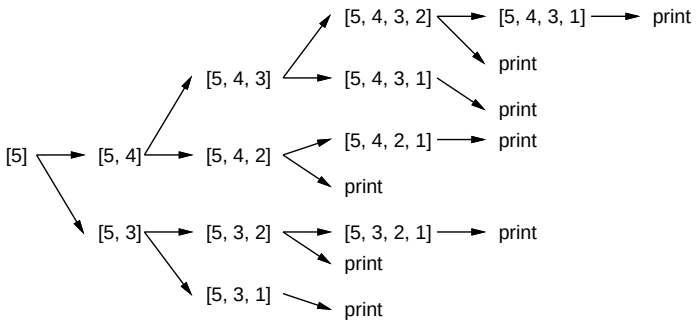


図 3.5: 再帰関数による 1〜2 減少列の生成

なお、n が 1 のときは、2 つ減らしたらマイナスになるのでまずいですから、1 つ減らしたものしか呼び出しません。そして n が 0 のときは…そのときは、もう追加するものはないので、追加する代わりに配列を打ち出します。

では Ruby 版を見てみましょう (メソッド p は配列を見やすく打ち出す機能を持っています)。

```
def decr1(n, b)
  if n > 1
    b.push(n); decr1(n-1, b); decr1(n-2, b); b.pop
  elsif n > 0
    b.push(n); decr1(n-1, b); b.pop
  else
    p(b)
  end
end
```

実行の様子も示します。

```
irb> decr1 5, []
[5, 4, 3, 2, 1]
[5, 4, 3, 2]
```

```
[5, 4, 3, 1]
[5, 4, 2, 1]
[5, 4, 2]
[5, 3, 2, 1]
[5, 3, 2]
[5, 3, 1]
=> 5
```

なお、ここでは自分自身を呼び出す回数は「2 回または 1 回または 0 回 (再帰を止める場合)」だったので if 文で枝分かれしていましたが、「N 回」呼び出す場合も考えられ、そのような場合はループを使って繰り返し呼び出すことになります (次節の例題がそうです)。

3.3.4 再帰呼び出しによる順列の列挙

アルゴリズムとしてよく求められるものの 1 つに、「与えた列のすべての順列を生成する」というものがあります。たとえば「123」という列を与えたら「123」「132」「213」「231」「312」「321」の 6 通りを生成するわけです。

これも先の例題と似たアルゴリズムで扱うことができます。つまり「途中までできた並び」の後に「残っているものから 1 つを取って追加」し、その先については自分自身を呼び出してやらせるわけです。このとき、「残っているもの」をどうやって扱うかが問題になりますが、配列で扱うのであれば、たとえばその要素を取った時に nil(何もないという印) に置き換えておくことが考えられます。こんどは Ruby コードを直接示します。

```
def perm1(a, b)
  if a.length == b.length
    p(b)
  else
    a.each_index do |i|
      if a[i] != nil
        x = a[i]; a[i] = nil; b.push(x)
        perm1(a, b)
        a[i] = x; b.pop
      end
    end
  end
end
```

つまり、結果の列 (b) が元の列と同じ長さなら全部並べ終わったので出力します。そうでない場合は、a のすべての要素について、nil でない (まだ残っている) ものを取り出し、b の末尾に追加し、自分自身を呼び出して処理し、終わったら取ったものを元に戻します (そして別のものを取ります)。実行例を見て頂きましょう。

```
irb> perm1 [1,2,3], []
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
=> [1, 2, 3]
```

ところでコーディングスタイルに関する話題ですが、筆者はこのように入れ子が深くなる場合は「この処理をしたら終わり」という命令を使って入れ子を深くならないように、たとえば次のように書くのが好みです。

```
def perm2(a, b)
  if a.length == b.length then p(b); return end
  a.each_index do |i|
    if a[i] == nil then next end
    x = a[i]; a[i] = nil; b.push(x)
    perm2(a, b)
    a[i] = x; b.pop
  end
end
```

つまり「最後まで来たら打ち出して終わり」「a[i] が nil ならこの周回は終わって次の周回へ飛ぶ (next)」を使うわけです。動作はどちらでも同じですが、どちらが読みやすいでしょうか。

演習 3-5 1〜2 減少列の例題を参考に、次のような列を打ち出す Ruby プログラムを作れ。

a N から始まる単なる減少列 (1〜N までのいくつ減ってもよい) をすべて打ち出す。

b 1〜N までの値を L 個並べた列をすべての場合について打ち出す (全部で N^L 個あるはず)。たとえば「$N = 3, L = 2$」であれば [1,1] [1,2] [1,3] [2,1] [2,2] [2,3] [3,1] [3,2] [3,3] の 9 通りを打ち出す。

c 配列に渡した値を重複を許して L 個並べた列をすべての場合について打ち出す。たとえば「['a', 'a', 'b'], 2」であれば ['a','a'] ['a','a'] ['a','b'] ['a','a'] ['a','a'] ['a','b'] ['b','a'] ['b','a'] ['b','b']」を打ち出す。

演習 3-6 与えられた配列の全ての並び替えを生成できるということは、その中から昇順に並んだものを選ぶことで、元の配列を昇順に整列するアルゴリズムができることになる。実際にそのようなプログラムを作ってみよ。また、この方法の弱点を検討し、できれば改良する方法についても検討せよ。

演習 3-7 自分の名前のローマ字表記を与えると、そのアナグラム (さまざまな順で文字を入れ替えたもの) を表示するプログラムを作りなさい。ただしローマ字として成立しないものは表示しないように工夫すること。

演習 3-8 その他、再帰による場合の列挙を利用して自分の興味のあるプログラムを作ってみなさい。

3.4 演習問題解説 (一部)

3.4.1 演習 3-1: 合計

引き算は簡単ですが、ちょっとひねって負の数にして sum というのもありかと思います。一通り作ってみました (短いメソッドは 1 行に書いています)。

これまでの数値を覚えるためには$list という配列を用意し、数値をこの後ろに追加して覚えて行きます。reset の時は現在の値とこの$list を別の変数に退避しておき、undo では退避しておいたものを元に戻します。

```
$x = 0; $sx = 0; $list = []; $slist = [];
def sum(v) $x = $x + v; $list.push(v); return $x end
def dec(v) sum(-v) end
def list() p($list, $x) end
def reset() $sx = $x; $x = 0; $slist = $list; $list = [] end
def undo() $x,$sx = $sx,$x; $list,$slist = $slist,$list end
```

実行例を示します。

```
irb> sum 1
  => 1
irb> sum 2.5
  => 3.5       ←合計 3.5
irb> dec 1.2   ←1.2 を引く
  => 2.3
irb> list      ←履歴表示
[1, 2.5, -1.2] ←履歴
2.3            ←現在値
  => nil
irb> reset     ←リセット
  => []
irb> sum 1
  => 1         ←また 0 からの和
irb> undo      ←リセットを戻す
  => [[1, 2.5, -1.2], [1]]
```

undo したときは過去の結果/リストと現在のものを交換するので、2 回 undo すると元に戻ります。

3.4.2 演習 3-2: RPN 電卓

これも一通り作ってみました (短いメソッドは 1 行に書いています)。演算を増やすのは基本的なやり型は前回示した add 等と同様で計算だけ変えればよいです。交換は 2 つの値を取り出して逆に入れればよいです。演算した内容を覚えるために、s というメソッドを用意しました。このメソッドは渡された値を文字列に変換して広域変数$str の後ろに連結して行きます。これがあれば、show はこの変数の内容を打ち出せばよいだけです (ついでに演算結果も打ち出していますが)。

```
$vals = []; $str = ''
def clear() $vals = []; $str = '' end
def s(x) $str = $str + ' ' + x.to_s end
def show() p($str, $vals[$vals.length-1]) end
def e(x) $vals.push(x); s(x); return $vals end
def add
  x = $vals.pop; $vals.push($vals.pop + x); s('+')
  return $vals
end
def sub
  x = $vals.pop; $vals.push($vals.pop - x); s('-')
  return $vals
end
```

```
def mul
  x = $vals.pop; $vals.push($vals.pop * x); s('*')
  return $vals
end
def div
  x = $vals.pop; $vals.push($vals.pop / x); s('/')
  return $vals
end
def exch
  x = $vals.pop; y = $vals.pop; s('x')
  $vals.push(x); $vals.push(y); return $vals
end
```

実行のようすを示します。

```
irb> e 1
  => [1]
irb> e 2
  => [1, 2]
irb> e 3
  => [1, 2, 3]  ←1、2、3を入れたところ
irb> mul        ←掛けたら6
  => [1, 6]
irb> add        ←足したら7
  => [7]
irb> show
" 1 2 3 * +"    ←履歴表示
7
  => nil
irb> e 4        ←さらに7を入れ
  => [7, 4]
irb> exch       ←交換
  => [4, 7]
irb> sub        ←引き算
  => [-3]
```

作ってみると、スタックを使った計算のようすがよく分かると思います。

3.4.3 演習 3-3: 行列電卓

2×2 行列の電卓ですが、加減算は要素ごとに演算すればよいので簡単ですね。乗算とか逆行列とかはちょっとごちゃごちゃしますが、まあこれらも 2×2 であればひたすら書けばできるでしょう。

```
$vals = []
def e(m) $vals.push(m) end
def add
  m = $vals.pop; n = $vals.pop
  $vals.push([[n[0][0]+m[0][0], n[0][1]+m[0][1]],
              [n[1][0]+m[1][0], n[1][1]+m[1][1]]])
```

```
    return $vals
  end
  def sub
    m = $vals.pop; n = $vals.pop
    $vals.push([[n[0][0]-m[0][0], n[0][1]-m[0][1]],
                [n[1][0]-m[1][0], n[1][1]-m[1][1]]])
    return $vals
  end
  def mul
    m = $vals.pop; n = $vals.pop
    $vals.push([[n[0][0]*m[0][0] + n[0][1]*m[1][0],
                 n[0][0]*m[0][1] + n[0][1]*m[1][1]],
                [n[1][0]*m[0][0] + n[1][1]*m[1][0],
                 n[1][0]*m[0][1] + n[1][1]*m[1][1]]])
    return $vals
  end

  def trans
    m = $vals.pop;
    $vals.push([[m[0][0], m[1][0]],
                [m[0][1], m[1][1]]])
    return $vals
  end
  def inv
    m = $vals.pop
    d = (m[0][0]*m[1][1] - m[0][1]*m[1][0]).to_f
    $vals.push([[m[1][1]/d, -m[0][1]/d],
                [-m[1][0]/d, m[0][0]/d]])
    return $vals
  end
```

3 × 3 以上になると、直接書くのではなくループを使った方が楽になりますが、そのあたりはまた回を改めてやる予定です。実行例をみてみましょう。

```
irb> e [[1, 2], [3, 4]]
  => [[[1, 2], [3, 4]]]
irb> e [[1, 1], [1, 1]]
  => [[[1, 2], [3, 4]], [[1, 1], [1, 1]]]
irb> sub
  => [[[0, 1], [2, 3]]]
```

引き算とかは問題ないですね。逆行列はどうでしょうか。

```
irb> e [[2, 1], [1, -1]]
  => [[[2, 1], [1, -1]]]
irb> inv
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]]]
irb> e [[1, 0], [5, 0]]
  => [[[0.3333333, 0.3333333], [0.3333333, -0.6666667]], [[1, 0], [5, 0]]]
irb> mul
  => [[[2.0, 0.0], [-3.0, 0.0]]]
```

これは何を計算しているかというと、次の連立方程式を解いています。

$$\begin{cases} 2x & + & y & = & 1 \\ x & - & y & = & 5 \end{cases}$$

これを行列の形に書くと次のようになります。

$$\begin{pmatrix} 2 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

ここで係数行列 `[[2, 1], [1, -1]]` を A、その逆行列を A^{-1} と記し、上式の両辺に左から A^{-1} を掛けます。

$$A^{-1}\, A \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix} = A^{-1} \begin{pmatrix} 1 & 0 \\ 5 & 0 \end{pmatrix}$$

$A^{-1}A$ は単位行列なので消えますから、つまり右辺を計算すると x と y が求まるわけです。実際、$x = 2$、$y = -3$ を代入してみると元の連立方程式が成り立っていることが確認できます。

3.4.4　演習 3-4: 再帰関数

これらは定義のとおり再帰関数にすればできるので、まずはコードを示します。

```
def fact(n)
  if n == 0 then return 1
  else           return n * fact(n-1)
  end
end
def fib(n)
  if n < 2 then return 1
  else          return fib(n-1) + fib(n-2)
  end
end
def comb(n, r)
  if r == 0 || r == n then return 1
  else                     return comb(n-1, r) + comb(n-1, r-1)
  end
end
def binary(n)
  if n == 0 then        return "0"
  elsif n == 1 then     return "1"
  elsif n % 2 == 0 then return binary(n / 2) + "0"
  else                  return binary((n-1) / 2) + "1"
  end
end
```

実行例も一応示しておきます。

```
irb> fact 4
  => 24
irb> fact 5
  => 120
irb> fib 2
```

```
  => 2
irb> fib 3
  => 3
irb> fib 4
  => 5
irb> comb 5, 2
  => 10
irb> comb 6, 2
  => 15
irb> binary 5
  => "101"
irb> binary 7
  => "111"
irb> binary 8
  => "1000"
```

3.4.5 演習 3-5: 列挙

単なる減少列は、ループを使って全部の減少の選択肢を生成できるので、コードとしてはむしろ「1〜2 減少列」より簡単になります。

```
def decrall(n, b)
  if n == 1 then p(b); return end
  1.step(n-1) do |i|
    b.push(i); decrall(i, b); b.pop
  end
end
```

呼び出し方は少し前回と違います (n が先頭にある配列の後ろに減少列を作るという仕様になっているため)。

```
irb> decrall(5, [5])
[5, 1]
[5, 2, 1]
[5, 3, 1]
[5, 3, 2, 1]
[5, 4, 1]
[5, 4, 2, 1]
[5, 4, 3, 1]
[5, 4, 3, 2, 1]
  => 1
```

もう 1 つは、「配列に渡した値を L 個を並べたすべての場合」ですが、これは重複を許してよいので順列より簡単になります。

```
def comball(n, a, b)
  if n == 0 then p(b); return end
  a.each do |x|
    b.push(x); comball(n-1, a, b); b.pop
  end
end
```

実行例も示しておきます。

```
irb> comball 3, ['a', 'b'], []
["a", "a", "a"]
["a", "a", "b"]
["a", "b", "a"]
["a", "b", "b"]
["b", "a", "a"]
["b", "a", "b"]
["b", "b", "a"]
["b", "b", "b"]
  => ["a", "b"]
```

3.5 検討 手続きの有用性

これまでは基本的に、1つのプログラムは1つのメソッド (関数、手続き) で出来ていたのですが (一部例外あり)、ここではメソッドをさまざまに使って目的とするコードを組み立てることがテーマになります。そうなると、メソッドからどのメソッドを呼ぶかということは、これまでの定型化された制御構造よりもずっと自由度が高くなるので、難易度も上がりますが、逆に言えば設計するときの腕の見せどころにもなります。

最初の話題は、既に出来上がっているメソッドを利用することで考えやすくなるという話題で、これはさして難しくありません。

次の話題は、グローバル変数に書き込むことでメソッドが副作用を持てるというもので、これを利用して一群のメソッドを使い「電卓」のようなまとまったサービスを提供できる例になっています。この部分は、自分でインタフェースをデザインして役に立つものが作れる、ということで魅力を感じてくれる学生もそれなりにいるようです。

その次は再帰呼び出しで、「自分で別の自分を呼び出す」ことで色々なものがうまく記述できることを題材としています。漸化式ふうに記述した関数をそのままメソッドとしてコーディングすると動くようになる、というのは説明としては分かりやすいのではないでしょうか。その後、より高度なものとして「枝分かれした制御点を実現する」再帰に進み、順列などを生成していますが、これはかなり難易度が高く、大学生でもつらい人は多いようです。

open question

- 「抽象化単位としての手続き」「副作用」「サービスの設計」「漸化式っぽい再帰」「高度な再帰」のうち、どのくらいまでを扱うのが適切でしょうか (一番最初のものだけでも十分有用だと考えています)?
- 抽象化単位に絞って、もっと抽象化を色々やってもらうという方向に構成することも考えられます。今回のものと、どちらがよいでしょうか?
- 電卓のような実用ぽいものを作ってもらうことは、モチベーションの点で有効だと思いますが、どうでしょうか?
- 再帰というと「階乗が再帰でできます」みたいなのだけで終わりがちなのですが、ここではかなり詳しくやっています。これくらいやった方がよいでしょうか? 後半はやりすぎでしょうか?

第4章　整列アルゴリズム

4.1　整列アルゴリズムを考える

今回は配列を扱うアルゴリズムの例として、数値の並んだ配列を受け取り、**昇順** (ascending order — 小さいものが先に来るような順番のこと) に並べ換えることを考えましょう。これを**整列** (sorting — と) 言います。[1]

アルゴリズムに入る前に、皆様が現実世界で整列を行うとき、たとえば数字を書いたカードを順番に並べるとき、どのようにするかを考えてみてください。ただし、コンピュータに移すことを前提に考えているので、次のように制限を設けます。

- カードは列にきっちり (間をあけずに左から詰めて) 並べること (配列に対応)。
- 2 本の人差指だけを使ってカードを指して動かす (コンピュータでは本当は操作できるデータは一時には 1 個だけれど、さすがに不自由すぎるので 2 本にします)。
- カードの数値を読んだり比較するときは、2 本の指のどちらかでそのカードを指す (これもコンピュータが操作できるデータは一時には 1 個だけだから)。

この条件で、実際にカードの並べ替えをやってみて頂きます。

演習 4-1 数字のカード (10 枚くらい) をよく切ってから机の上に左から 1 列に並べ、上の条件を守って小さい順に並べ替えてみよ。

4.2　素朴な整列アルゴリズム

4.2.1　バブルソート

整列のアルゴリズムは見つかりましたか。では、一番基本的な整列アルゴリズムを 1 つお見せしましょう。次の擬似コードを見てください:

- bubsort(a): 配列 a を昇順に整列
- 　done ← 偽。
- 　done でない間繰り返し、
- 　　done ← 真。
- 　　i を 0 から a.length-2 まで変えながら繰り返し、
- 　　　もし a[i] と a[i+1] の順番が逆なら、
- 　　　　a[i] と a[i+1] の値を交換。
- 　　　　done ← 偽。
- 　　　枝分かれ終わり。
- 　　繰り返し終わり。
- 　繰り返し終わり。

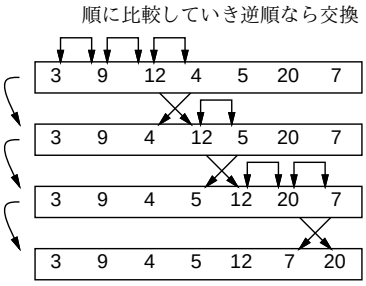


図 4.1: バブルソートによる整列

このコードの肝となるところは、内側のループで隣り合う要素を順に見ていき、逆順になっているところがあれば交換する、というところです。これを次々におこなっていくと、大きい要素が右のほうに移動していきます (図 4.1)。この処理を繰り返していくと、最後は全ての要素が昇順で並び、交換が起きなくなるはずです。

繰り返しを終わってよいかどうか判断するために、`done`(終了) という旗を用意し、まず立ててから上記の比較交換を行います。交換を行ったら、そのことを示すために旗を降ろします。最後まで旗が立ったままだったら、1 回も交換しなかった、つまり 1 箇所も逆順になっているところがなかったということなので、整列が完了しています。

この整列方法を**バブルソート** (bubblesort) と呼びます。各要素が移動する様子が水中から泡が浮かんでくるのに似ているためにこう呼ばれるとされています。

ところで、「`a[i]` と `a[i+1]` を**交換** (swap)」という命令は言語には直接ないので、これを手続きとして記述します:

```
def swap(a, i, j)
  x = a[i]; a[i] = a[j]; a[j] = x
end
```

これで配列 `a` の `i` 番目と `j` 番目の要素を入れ換えることができます (実際にそうなっていることをコードを追って確認しておいてください)。この程度のコードであれば、いちいち手続きとして抽象化しないで直接書いてしまいたい、と思うかもしれません。筆者の考えとしては、それはコードに対する慣れにもよってどちらもありだと思いますが、「交換」するところに「`swap`」と明示的書いてある分かりやすさも捨てがたいと思っています。[2]

バブルソート本体は次のようになります:

```
def bubblesort(a)
  done = false
  while !done do
    done = true
    0.step(a.length-2) do |i|
      if a[i] > a[i+1] then swap(a, i, i+1); done = false end
    end
  end
end
```

では実際に動かしてみましょう:

[1]逆に大きいものが先に来るような順番の場合は**降順** (descending order) と呼びます。一般に列を昇順や降順に並べ換える処理のことを**整列** (sorting) と呼びます。

[2]Ruby では**多重代入** (multiple assignment — 複数の代入を一度に行うこと) が使えるため、`swap` の本体を `a[i],a[j] = a[j],a[i]` と書くこともできます。多重代入機能を持たない言語も多いので、ここでは「普通の」やり方を示しておきました。

```
irb> a = [1, 9, 5, 4, 2]
=> [1, 9, 5, 4, 2]
irb> bubblesort(a)
=> nil
irb> a
=> [1, 2, 4, 5, 9]
irb>
```

bubblesort 自体は値を返さず、配列 a の中身を書き換えて昇順に整列していることに注意。したがって、まず配列 a を用意し、それを bubblesort に渡して整列させ、最後に a を打ち出して並んでいることを確認しています。

4.2.2 単純選択法

バブルソートのように、「求める状態が成り立っていない間、少しでもその状態に近付けることをずっと繰り返す」というのはコンピュータではよくありますが、人間はあんまりそのなやり方はしない気もします。もうちょっと自然な考え方のものとして、次のものはどうでしょうか。

> 数の並びから最小値を取り出してはその並びからは取り除くことを繰り返していく。
> その取り出したものを順に並べると昇順の整列結果になっている。

この方法を、単純に小さいものをそのつど選ぶことから**単純選択法** (selection sort) と呼びます。

これを作るに当たっては、「配列 a の i 番目から j 番目までの間で最も小さい要素が何番目にあるかを返す」操作を下請けメソッドとして用意するのがいいでしょう。それがあったとして、アルゴリズムは次のようになります:

- selectionsort(a): 配列 a を単純選択法で整列
- 　i を 0 から a.length-2 まで変化させながら繰り返し、
- 　　k ← a の i 番から a.length-1 番までの最小要素の番号。
- 　　a[i] と a[k] の内容を交換。
- 　繰り返し終わり。

なぜ「交換」を使っているのかというと、まず選んだ最小の要素を先頭に置くには、先頭にある要素と最小の要素とを交換するのが合理的だからです。その後も、残っているものの中から最も小さい要素を選んではその先頭位置と交換することで、1 つの配列だけですべての作業が行えます (図 4.2)。

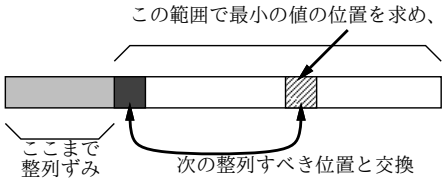


図 4.2: 単純選択法による整列

ではコードを示します。arrayminrange は以前やった最大/最小と同様に考えればよいでしょう (最小値そのものでなくその位置を返すことに注意):

```
def selectionsort(a)
  0.step(a.length-2) do |i|
    k = arrayminrange(a, i, a.length-1); swap(a, i, k)
```

```
    end
  end
  def arrayminrange(a, i, j)
    p = i; min = a[p]
    i.step(j) do |k|
      if min > a[k] then p = k; min = a[k] end
    end
    return p
  end
```

4.2.3　単純挿入法

単純選択法は、数を 1 つずつ処理しますが、それらを「取り出す」時に正しい順になるようにするというものでした。人間にとって自然なもう 1 つのやり方は、取り出すのは「最初に並んでいる順」で、入れる時に正しい位置に入れる、というものです。

> 数の並びから順に数を取り出し、それを新しい列に加えるが、ただしただし新しい列に入れる時に「順番として正しい」位置に挿入するようにする (その後ろにある要素はずらす必要があることに注意)。

この方法は、単純に各要素を次々とあるべき位置に挿入していくことから、**単純挿入法** (insertion sort) と呼ばれます。

これを作るに当たっては、「配列の a の i 番目から j 番目までを 1 つ後ろにずらす操作」を下請けメソッドとして作るのがいいでしょう。それがあったとして、単純挿入法のアルゴリズムは次のようになります:

- insertionsort(a): 配列 a を単純挿入法で整列
- 　i を 1 から a.length-1 まで変化させながら繰り返し、
- 　　x ← a[i]。
- 　　k ← 0。
- 　　k < i かつ a[k] ≤ x である間繰り返し k ← k+1。
- 　　a の k 番目から i-1 番目までを 1 つ後ろにずらす。
- 　　a[k] ← x。
- 　繰り返し終わり。

これも、元の配列からデータを取り除きながらそれをもとに先頭部分に整列されている部分を作っていくので、配列は 1 つだけで済みます (図 4.3)。なお、配列を「後ろにずらす」時に後ろから順にやらないとまずいことに注意してください (図 4.4)。

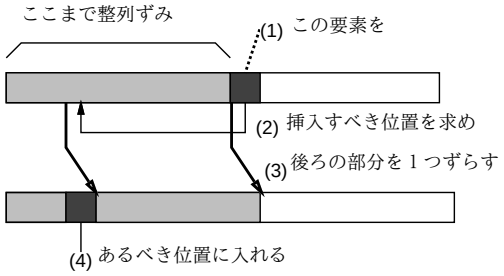


図 4.3: 単純挿入法による整列

ずらすコードと本体のコードは次のとおりです:

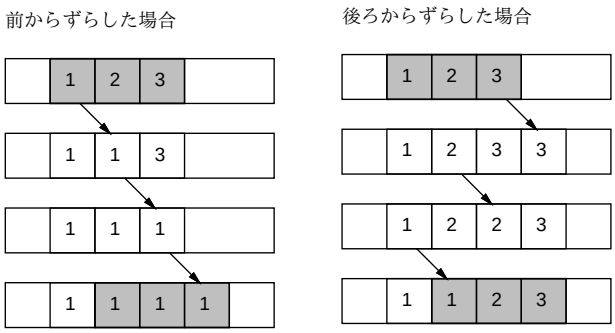


図 4.4: 配列をずらす

```
def insertionsort(a)
  1.step(a.length-1) do |i|
    x = a[i]; k = 0
    while k < i && a[k] <= x do k = k + 1 end
    arrayshiftrange(a, k, i-1); a[k] = x
  end
end
def arrayshiftrange(a, i, j)
  j.step(i, -1) do |k| a[k+1] = a[k] end
end
```

4.3 整列アルゴリズムの計測

4.3.1 時間計測の方法

次の段階として、「整列プログラムの時間を計測する」問題に取り組んでみましょう。「バブルソート」「単純選択法」「単純挿入法」の 3 つの整列アルゴリズムについて、データ量を変化させて**時間計測**を行ってみます。データは次のコードにより個数を指定して**乱数**によりランダムに生成します。

```
def randarray(n)
  return Array.new(n) do rand(10000) end
end
```

rand は乱数を生成するメソッドで、パラメタを指定しないと区間 $[0, 1)$ の**一様乱数** (uniform random number) を (実数値で) 返します。パラメタとして整数 N を指定すると、0 以上 N 未満の整数値の一様乱数を返します。ちょっと試してみましょうか。

```
irb> randarray 10
=> [9257, 4988, 6894, 8064, 329, 4362, 1868, 472, 1527, 6317]
```

次に、時間計測に役立つメソッドを用意します。これは、実行回数とブロックをパラメタとして受け取り、「まず現在の時計を調べ」「指定回数ぶんだけブロックの中身を実行し」「再び時計を調べ」「2 つの時刻の差を表示」します。ただしここでの時計は「CPU をどれだけ使ったか」を示す時計になっています。

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

```
    t2 = Process.times.utime
    puts t2-t1
  end
```

これもちょっと使ってみましょう。

```
irb> bench(100000) do 2 + 1 end
0.078125
=> nil
irb> bench(100000) do 20000000000000000 + 1 end
0.171875
=> nil
```

整数の場合、ある程度より大きくなると計算時間が余分に掛かるようになることが分かります。

さて、これらの材料を使って、整列の速度を測ります。randarray で多めの配列を生成し、bench では回数として 1 回を指定して整列を行い、時間を計測します。これを 1 行にまとめて書くとして、次のようになります:

```
irb> a = randarray(1000); bench(1) do bubblesort(a) end
=> 1.21875    ←計測結果が表示される
```

演習 4-2 バブルソート、単純選択法、単純挿入法のうちから 1 つ好きな整列アルゴリズムを選んで打ち込み、複数のデータ量で時間計測を行い、データ量と所要時間の関係がどうなっているか分析せよ (グラフに描いて観察するなど — グラフはレポートにはつけなくて良いです)。

なお、下請けのメソッドや計測メソッドも忘れずに打ち込む必要があります。具体的には次のようになります。

- バブルソート — bubblesort、swap、randarray、bench
- 単純選択法 — selectionsort、arrayminrange、randarray、bench
- 単純挿入法 — insertionsort、arrayshiftrange、randarray、bench

4.3.2 基本的な整列アルゴリズムの計測

とりあえず、筆者の手元のマシンでのバブルソート、単純選択法、単純挿入法の計測結果を、表 4.1 に示します。これを見ると、バブルソートが圧倒的に遅く、残りの 2 つはそれほど大きな差は

表 4.1: バブルソート/単純選択法/単純挿入法の所要時間 (msec)

データ数	1,000	2,000	3,000
バブルソート	1,219	4,945	11,117
単純選択法	305	1,242	2,766
単純挿入法	375	1,531	3,445

ない、ということが分かります。これは、バブルソートはすべての要素を移すのに隣と 1 個ぶんずつ交換してゆくのでどうしても手間が多くなるのに対し、他の 2 つでは「1 個データを選んで、それを適切な位置に置く」ことを繰り返す形なので、それだけ手間が少なくなるからだと言えるでしょう。

では次に、データの量が 2 倍、3 倍になった時の所要時間を見てみると、こんどはどのアルゴリズムでも所要時間がほぼ 4 倍、9 倍になっていることが分かります。$4 = 2^2$、$9 = 3^2$ ですから、どのアルゴリズムでも「所要時間はデータ量の 2 乗に比例している」と言ってよいでしょう。ということは、データ数が 100,000(100 倍) になった時の所要時間は単純選択法でも $0.3 \times 10,000 = 3000$ 秒 $= 50$ 分 (!) となってしまい、終わるまで待つのはあまり嬉しいものではないと分かります。

4.4 より高速な整列アルゴリズム

4.4.1 マージソート

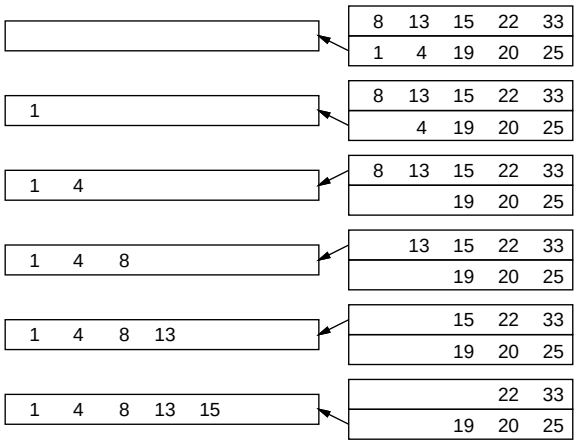


図 4.5: マージの処理

では、整列アルゴリズムでもっと速いものはないのでしょうか。ここで**マージソート** (merge sort) と呼ばれるアルゴリズムを見てみましょう。**マージ** (merge) とは**併合**とも呼ばれ、図 4.5 のように 2 つの整列ずみの列を「あわせて」1 つの整列ずみの列にすることを言います。マージソートの手続きの呼び出し時には次のように、「配列のどこからどこまでを整列する」かを指定するものとします:

```
mergesort(a, 0, a.length-1);
```

擬似コードを示します:

- mergesort(a, i, j) — 配列 a の i 番から j 番の範囲を整列
- 　もし j ≤ i なら、
- 　　なにもしない。
- 　そうでなければ、
- 　　k ← (i + j) / 2。
- 　　mergesort(a, i, k)。mergesort(a, k+1, j)。
- 　　b ← merge(a, i, k, a, k+1, j)。
- 　　{b の内容を a の位置 i〜j にコピーし戻す }
- 　枝分かれ終わり。

考え方としては、まず再帰呼び出しによって列全体を半分ずつにしてゆき、長さ 1 以下の時は「もう整列ずみ」なので何もしないで帰ります。そして再帰から戻ってきたら、2 つの整列ずみの列をマージすることで長い整列ずみの列にします (図 4.6)。

下請けとなるマージの擬似コードは次の通り。

- merge(a1, i1, j1, a2, i2, j2) — a1[i1..j1] と a2[i2..j2] を併合
- 　b ← 空の配列
- 　i1..j1 と i2..j2 の少なくとも一方が空でない間、
- 　　もし i1..j1 が空 または a1[i1] ¿ a2[i2] なら、
- 　　　a2[i2] を b に追加し、i2 を 1 ふやす。
- 　　そうでなければ、

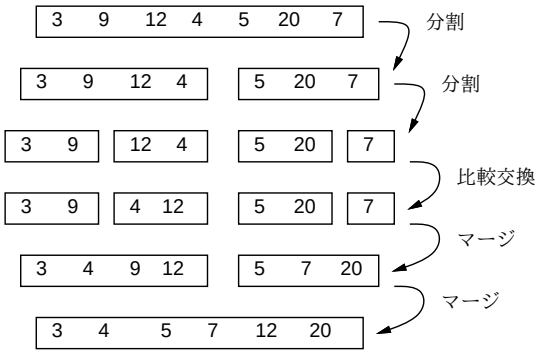


図 4.6: マージソートによる整列

- a1[i1] を b に追加し、i1 を 1 ふやす。
- 枝分かれ終わり。
- 繰り返し終わり。
- b を返す。

では Ruby 版を見てみましょう。

```
def mergesort(a, i, j)
  if j <= i
    # do nothing
  else
    k = (i + j) / 2
    mergesort(a, i, k); mergesort(a, k+1, j)
    b = merge(a, i, k, a, k+1, j)
    b.length.times do |l| a[i+l] = b[l] end
  end
end
def merge(a1, i1, j1, a2, i2, j2)
  b = []
  while i1 <= j1 || i2 <= j2 do
    if i1 > j1 || i2 <= j2 && a1[i1] > a2[i2]
      b.push(a2[i2]); i2 = i2 + 1
    else
      b.push(a1[i1]); i1 = i1 + 1
    end
  end
  return b
end
```

マージソートは次に出て来るクイックソートに比べて速くはないのですが、データを端から順に処理していけるという特徴があります。このため、メモリに入り切らない (ファイルに保管されている) 大量データの処理に多く使われます。その原理は次のようなものです:

- データをファイルから読みながら、メモリに入る最大量ずつクイックソートなどで整列し、別々のファイルに書き出す。
- ファイル 1 とファイル 2 をマージしてファイル 12 を作り、ファイル 3 とファイル 4 をマージしてファイル 34 を作り、…のようにファイルを対にしてマージしていく。

- これを繰り返していって、最後に 1 本のファイルになったら完了。

このような、ディスクなど外部記憶の使用を前提とした整列のことを**外部整列** (external sorting) と呼びます。これと対比して、本文で扱っているような、メモリ上での整列のことを**内部整列** (internal sorting) と呼びます。

4.4.2 クイックソート

もう 1 つ別のアルゴリズムを直接 Ruby プログラムで示しましょう。これは**クイックソート** (quicksort) という、いかにも速そうな名前がついています:

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

非常に短いですが、説明されないと分かりませんね。まず長さ 1 以下なら何もしないのはマージソートと同様です。次に、マージソートと同じく列を 2 つに分けますが、こちらは**ピボット** (pivot) と呼ぶある値 p を選び、「左半分は p 以下、続いて p の値、右半分は p より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になるのでこれで整列が完了するわけです (図 4.7)。

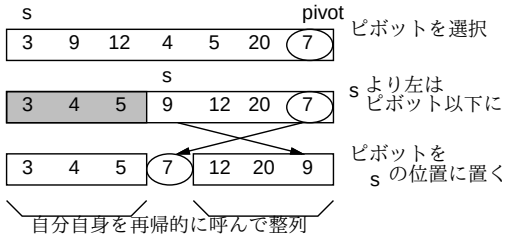


図 4.7: クイックソートによる整列

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないのでランダムに選ぶこととし、上のコードでは右端 (j 番目) の値を p にしています。変数 s は「この番号の 1 つ手前までは p 以下のものを詰めてあるので、次に p 以下のものが見つかったらこの位置に入れる」番号を表しています。そこで、k を i から j-1 まで左から順に調べて、[3]a[k] が p 以下ならそれを s 番目の要素と交換して s を増やすことで、左半分と右半分に分けられます。分け終わったら、最後に j 番目と s 番目を交換することで、保留してあったピボットの値をあるべき位置に置きます。その後、自分自身を再帰的に呼ぶわけですが、s 番目のピボットの位置はこれで合っているので、i〜s-1 と s+1〜j の範囲について自分自身を呼びます。

[3] j 番はピボットが入っているので保留します。

演習 4-3 マージソートとクイックソートの好きなほう (両方でもよい) を打ち込んで動かし、所要時間を計測してみよ。配列サイズを変化させた時の挙動は先にやったバブルソートや単純選択法や単純挿入法と比べてどうか考察せよ。

4.5　時間計算量

4.5.1　時間計算量の考え方

ここまででさまざまなアルゴリズムを実現するプログラムの所要時間の計測について話題にしてきましたが、本節ではアルゴリズムの**性能** (performance) を評価する指針の 1 つである**計算の複雑さ** (computational complexity) ないし**計算量** (complexity) について取り上げます。complexity だと日本語は「複雑さ」になりそうですが、「複雑さ」という日本語では一般的すぎて何のことか分かりにくいので、日本語では「計算量」と呼ぶわけです。なお、計算量には「どれくらいメモリが必要になるか」を表す**領域計算量** (space complexity) もありますが、ここではとりあえず「所要時間」に着目する**時間計算量** (time complexity) を取り上げます。

`selectionsort` を例題にして、これがどれくらいの時間を要するかかを見積もってみましょう。その前に、まず次の前提を置きますが、これはいいですね?

> コンピュータは、ある 1 つの決まった動作はその動作に応じた決まった時間で実行している。

このことは、バブルソートなどの実行時間を測ってもそう大きくは変動しないことからも分かると思います。では次に、選択ソートのプログラムを「実行回数によって区分した」ものを図 4.8 に示します。

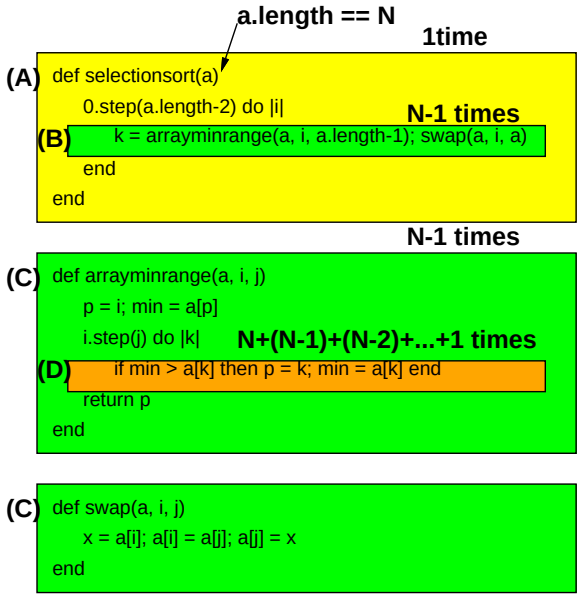


図 4.8: 選択ソートのコード実行回数の検討

ここで、渡された (整列する) 配列の長さを N とします。そうすると、実行回数について、次のことが分かります。

(A) メソッド `selectionsort` の本体部分は「1 回」実行される。

(B) その中の do の内側については、「$N-1$ 回」実行される。

(C) したがって、`arrayminrange` や `swap` も「$N-1$ 回」実行される。

(D) `arrayminrange` の中の do の内側は、最初は N 回、次は $N-1$ 回、次は $N-2$ 回…、ということで、合計 $\frac{N(N+1)}{2}-1$ 回実行される。

ここで、(A) の部分 (から (B)、(C) の部分は除いたもの、以下同様) の実行に掛かる時間を C_0、(B) と (C) の部分を 1 回実行するのに掛かる時間を C_1、(D) の部分を 1 回実行するのに掛かる時間を C_2 と置くと、合計実行時間は次のようになりますね。

$$T = C_0 + (N-1)C_1 + (\frac{N(N+1)}{2} - 1)C_2$$

これを展開して N について整理すると次のようになります。

$$T = \frac{C_2}{2}N^2 + (\frac{C_2}{2} + C_1)N + (C_0 - C_1 - \frac{C_2}{2})$$

ここで、たとえ C_0 や C_1 が C_2 の 100 倍あるとしても (確かに C_1 の部分の方が沢山あるけれど、どう見ても 100 倍は無いですよね?)、N が 1000 とか 10000 とかもっと大きな値で動かすわけなので、結局、次のように近似してもほぼ間違いではなくなってしまうわけです。

$$T \sim \frac{C_2}{2}N^2$$

そういうわけで、時間計算量とは「最も高次の次数だけを問題にする」考え方で、選択ソートについては $O(N^2)$ のように記すわけです。じゃあ係数はどうなんだ? ということですが、係数は「同じ計算量のプログラムどうし」では問題になりますが、計算量が違うプログラムであれば多少の係数の大小があっても結局は次数で決まってしまうので、無視してよい、というわけです。

N 個のデータを入力するようなプログラムでは、そのデータの読み込みに $O(N)$ は最低必要です (なお、N 個の値を扱うとしても、それを内部的に計算するだけなら、計算を工夫して $O(N)$ より小さいアルゴリズムを構成できる場合もあり得ます)。この、$O(N)$ のアルゴリズムのことを (N に比例するわけですから) **線形時間** (linear time complexity) と呼びます。たとえば最大や最小を求める問題はデータを読みながら一巡すれば結果が求まるので、線形時間のアルゴリズムで扱えます。このような問題はコンピュータで簡単に処理できると言えます。

少し込み入ったアルゴリズムは、$O(N^2)$ や $O(N^3)$ などの計算量になります。これを**多項式時間** (polynominal time complexity) と呼びます。さらに時間計算量の大きなものとしては、$O(C^N)$ すなわち**指数時間** (exponential time complexity) となる場合もあり、これだとコンピュータで実用的に扱えるのは小さい N に限られてしまいます。

4.5.2 整列アルゴリズムの時間計算量

では次に、単純挿入法の時間計算量はどうでしょうか。外側のループで i を 1〜N まで変えながらその番号の要素を適切な位置に挿入していきます。挿入位置を探索するのに平均して $\frac{i}{2}$ 個の要素を比較し、挿入位置が見つかったら平均して $\frac{i}{2}$ の要素を後ろにずらす必要があります。なのでこれも $1+2+\cdots+(N-1)$ の定数倍、つまり $O(N^2)$ になります。

バブルソートの時間計算量はどうでしょうか。内側のループでは $N-1$ 回の比較をおこないます。そして、**最善の場合** (ideal case) つまり最初から全部並んでいる場合は、1 回内側のループを実行したらそれで完成です。つまり $O(N)$ となります。しかし**最悪の場合** (worst case)、つまり完全に逆順に並んでいる場合は、内側の 1 回目のループは最も大きい要素を最後の位置に持ってくるだけで終わってしまい、2 回目は 2 番目に大きい要素を最後から 2 番目の位置に…というわけで、内側のループが N 回必要になります。つまり $O(N^2)$ となるでしょう。では**平均の場合** (average case) はどうでしょうか。平均的には、内側のループの繰り返しは N 回は必要ないとしても、N に比例する回数が必要になりそうです。ということは、平均でも定数倍は無視するのでやはり $O(N^2)$ になるわけです。

ここで表 4.1 の結果を振り返ると、単純選択法もバブルソートも N が 2 倍、3 倍になった時所用時間が 4 倍、9 倍になっているので、この計測結果はこれらが確かに $O(N^2)$ の時間計算量であることを裏付けています。

ではマージソートの計算量はどうでしょうか。1 つの `mergesort` の呼び出しを見ると、単純な場合 (長さが 1 以下) は一定時間で済みます。長さ N の場合は、それを前半と後半に分けて、それぞれ自分自身を再帰的に呼び出して整列し、最後にマージします。自分自身に掛かる時間は分けて考えるとして、マージは両方の列の先頭を見て小さいほうを取ることを繰り返せばいいので、$O(N)$ で済みます。さて、再帰呼び出しのほうはどうでしょうか。長さ N の列を半分にしてそれぞれ `mergesort` を呼ぶのですから、2 段目の呼び出しは $O(\frac{N}{2}) + O(\frac{N}{2}) = O(N)$ 。3 段目は 4 分の 1 の列について 4 つ呼ぶのでやはり $O(N)$ 、となります。これが合計何段あるかというと、「N を何回半分にしたら 1 になるか」だから $\log_2 N$ となります。なので、全体では $O(N \log N)$ の計算量となります (計算量の議論では log の底が何かも省略するのが通例です)。

では、クイックソートの計算量はどうでしょうか。1 回ぶんの処理はやはり $O(N)$ で、再帰の段数はピボットの選択が完璧なら $\log_2 N$ 回ですが、ランダムに選んでいるのでその定数倍と考えてよいでしょう。すると定数倍は無視するので、これも計算量は $O(N \log N)$ になります。

ただし、極めて運が悪い場合、つまりピボットの選択が悪くて毎回列の最大か最小の値をピボットにしてしまうと、段数が N になってしまうので、最悪の計算量は $O(N^2)$ ということになります。そんな運が悪いことはないだろうと思うかもしれませんが、既に整列ずみの値を渡されるとまさにそうなってしまうのです。

演習 4-4 クイックソートに既に並んでいる配列を与えると計算量が $O(N \log N)$ から $O(N^2)$ になってしまうことを計測により確認しなさい。また、この弱点を解消する工夫を考えて実現してみなさい。

4.6 整数値のための整列アルゴリズム

4.6.1 ビンソート

ここまでの方法とは考え方がまったく違う整列アルゴリズムである、**ビンソート** (bin sort) を紹介しましょう。このアルゴリズムは、整列する値が整数であり、かつ範囲があまり広くない場合に利用できます。たとえば、整列する値の範囲が 0〜3 の整数だけだったとします (もちろん、そのデータの個数は 1 万も 2 万もあるかもしれません)。

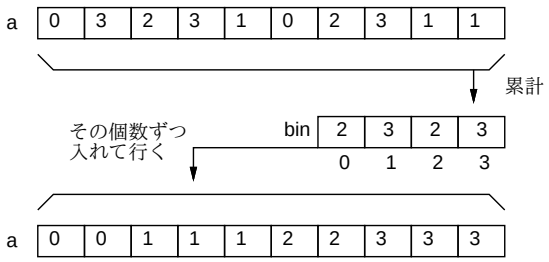


図 4.9: ビンソートによる整列

それなら図 4.9 のように、まず 0、1、2、3 それぞれの値について「何回現れるか」を数えてしまいます。そして数え終わったらこんどは「0 が 2 回、1 が 3 回、…」のように数えた個数ずつその値を繰り返せば、確かに元のデータを並べ換えたのと同じことになるわけです。0〜3 ではあんまり役に立たないと思うでしょうが、実際にはコンピュータのメモリは沢山あるので、0〜9999 とかでも全く問題ありませんし、それなら使い道は結構ありそうですね? [4]

[4]先に掲げた `randarray` が生成するデータもこの範囲の整数であることに注意してください。もちろんわざとそうしたのですが。

演習 4-5 ビンソートのプログラムを作成し、所要時間を計測しなさい。もちろん、ビンソートの時間計算量についても検討すること。

4.6.2 基数ソート

ビンソートの弱点は、現れる値の範囲があまりに広いと (100 万とか 1000 万とか) 巨大な配列を必要とし、効率も悪くなることです。そこで、やはり値が整数である必要があるものの、ビンソートよりも値の範囲に対する許容度が高い整列アルゴリズムである**基数ソート** (radix sort) を紹介しましょう。ここでは簡単のため、負の値はないものとして説明します。

基数ソートでは、整列する値を 2 進表現した時に「下から i ビット目が 1 であるか否か」を調べる必要があります。これを Ruby でどう書くかを説明しておきます。

Ruby では<<という演算子は**左シフト** (left shift) つまりビット列である整数値を 1 ビットぶん左にずらす働きがあります。だから `1 << 2` は $100_{(2)}$ だから 4 だし、一般に `1 << i` で `i` 番目のビットだけが 1 になった数値をえることができます。[5]

次に、&という演算子は**ビット毎 and**(bitwise and) 演算つまり 2 つの数の 2 進表現で「両方とも 1」の位置だけが 1、それ以外は 0 であるような 2 進表現に対応する数が得られます。[6] たとえば図 4.10 のように、`52 & 29` の結果は 20 ということになります。

```
  110100 — 52
& ) 011101 — 29
  ------
  010100 — 20
```

図 4.10: ビット毎 and 演算

ここでようやく、基数ソートの説明に入ります。たとえば、変数 `mask` に 1 ビットだけが「1」になっている値を入れ、その 1 の位置を一番右 (下位) から順に左に移していきます。そして、その `mask` との&の結果が 1 か 0 かで、データを右半分と左半分に分割します (図 4.11)。そうするとあらふしぎ、一番上のビット (ここでは 4 ビットとしました) までやったときには、すべての数は小さい順に並んでいます。

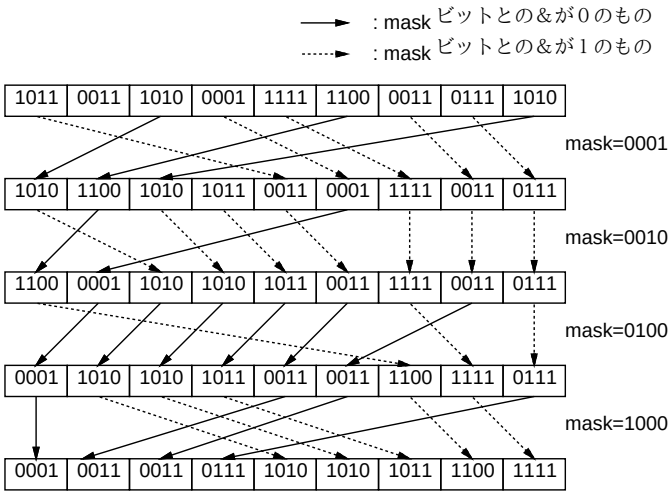


図 4.11: 基数ソートによる整列

これはなぜかというと、1 回目では一番下のビットが 1 のものが左、0 のものが右になるように振り分け、それについて 2 回目に下から 2 ビット目が 1 のものが左、0 のものが右になるように振

[5] そして今回は使いませんが、もちろん>>は**右シフト** (right shift) 演算子です。

[6] 条件の「かつ」は&&でしたが、アンド記号が 1 個の場合はまったく別の意味になるわけです。ちなみに、|はビット毎 **or**(bitwise or) 演算、~はビット毎反転 (bitwise inversion) 演算です。

り分けるわけですが、2回目の振り分けをしても1回目の振り分けの順序は崩れないので、2ビット目が1のものの中や、0のものの中ではそれぞれ、まず1ビット目が1のもの、続いて0のものという順序が維持されています。3ビット目、4ビット目でも同様にそれより下のビットについては順序が維持されているので、結局最後まで来たときには順番が完全に並んだ状態となるわけです。

演習 4-6 基数ソートのプログラムを作成し、所要時間を計測しなさい。もちろん、基数ソートの時間計算量についても検討すること。

演習 4-7 ここまでに出て来なかった整列アルゴリズム (あなたが考案したものでもよい) を1つ選び、所要時間を計測しなさい。もちろん、時間計算量についても検討すること。

4.7 演習問題解説 (一部)

4.7.1 演習 6-3(1) — マージソートの時間計算量

マージソートについて再度見直してみましょう。コードを示します。

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

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

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

では、計算量の検討はどうでしょうか? 最初に「長さ N の配列を1個」整列する形で呼び出すと、1回目の再帰では「長さ $\frac{N}{2}$ の配列を2個」整列することになり、2回目では「長さ $\frac{N}{4}$ の配列を4個」整列することになります。ということは、それぞれの段 (「回目」) では、自分自身の再帰呼

び出しを除くと、合計で長さ N のデータをマージし、元の配列に書き戻します。ですから、1 段について $O(N)$ の時間が掛かります。では再帰は何段起こるでしょうか? それは、長さを半分ずつにしていって 1 になったらおしまいですから、段数を L とすると、$2^L \sim N$、つまり $L \sim \log_2 N$ ということになります。$O(N)$ の処理が L 段あるのですから、全体としての時間計算量は $O(N \log N)$ ということになります。

4.7.2 演習 6-3(2) — クイックソート

クイックソートのコードを再掲します。

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

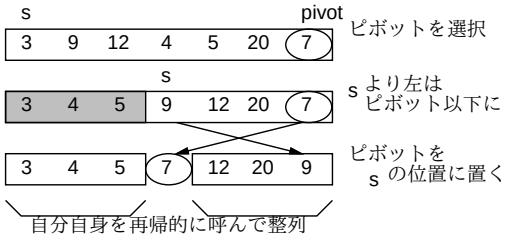


図 4.12: クイックソートによる整列 (再掲)

非常に短いですが、説明されないと分かりませんね。まず長さ 1 以下なら何もしないのはマージソートと同様です。次に、マージソートと同じく列を 2 つに分けますが、こちらは**ピボット** (pivot) と呼ぶある値 p を選び、「左半分は p 以下、続いて p の値、右半分は p より大きい」という状態にしてから、左半分と右半分をそれぞれ自分自身を再帰呼び出しして整列します。そうすると、「p 以下の整列された列」「p」「p より大きい整列された列」になり、整列が完了します (図 4.12)。

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストですが、そんなものは分からないのでランダムに選ぶこととし、上のコードでは右端 (j 番目) の値を p にしています。変数 s は「この番号の 1 つ手前までは p 以下のものを詰めてあるので、次に p 以下のものが見つかったらこの位置に入れる」番号を表しています。そこで、k を i から j-1 まで左から順に調べて、[7]a[k] が p 以下ならそれを s 番目の要素と交換して s を増やすことで、左半分と右半分に分けられます。分け終わったら、最後に j 番目と s 番目を交換することで、保留してあったピボットの値をあるべき位置に置きます。その後、自分自身を再帰的に呼ぶわけですが、s 番目のピボットの位置はこれで合っているので、i〜s-1 と s+1〜j の範囲について自分自身を呼びます。

では、クイックソートの時間計算量はどうでしょうか。理想的な場合、つまり毎回列がおよそ半分ずつになるとすると、再帰呼び出しの深さは $\log_2 N$ になります (たとえば 16 なら 4 段、64 なら

[7] j 番はピボットが入っているので保留します。

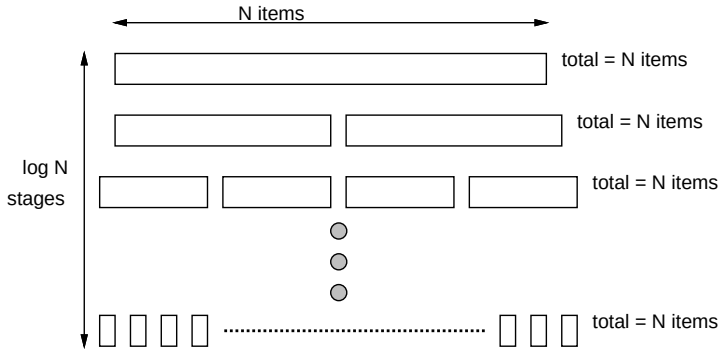


図 4.13: クイックソートのコード実行回数の検討

6 段という感じ)。そして、各深さにおいて、その深さの呼び出しを全部合わせると、N 個のデータ全部をピボットと比較して振り分けることになります。これを合計すると、N 個のデータを振り分けることを $\log N$ 段繰り返しておこなうので、計算量は $O(N \log N)$ となります。

4.7.3　演習 6-5 — ビンソート

ビンソートのアルゴリズムについて、再度簡単に説明します。このアルゴリズムは、整列する値が整数であり、かつ範囲があまり広くない場合に利用できます。たとえば、整列する値の範囲が 0〜3 の整数だけだったとします (もちろん、そのデータの個数は百万とか一千万とかあるかも知れません)。それなら図 4.14 のように、まず 0、1、2、3 それぞれの値について「何回現れるか」を数えてしまいます。そして数え終わったらこんどは「0 が 2 回、1 が 3 回、…」のように数えた個数ずつその値を繰り返せば、確かに元のデータを並べ換えたのと同じことになるわけです。0〜3 ではあんまり役に立たないと思うでしょうが、実際にはコンピュータのメモリは沢山あるので、今回のように範囲が 0〜9999 とかくらいなら全く問題ありません。

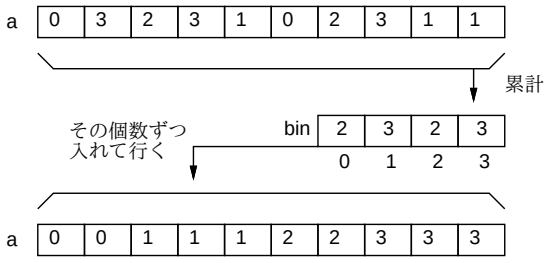


図 4.14: ビンソートによる整列 (再掲)

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

```
def binsort(a)
  bin = Array.new(10000, 0)
  a.each do |i| bin[i] = bin[i] + 1 end
  k = 0
  bin.length.times do |i|
    bin[i].times do a[k] = i; k = k + 1 end
  end
end
```

4.7.4 演習 6-6 — 基数ソート

基数ソートを十進で説明し直します (図 4.15)。たとえば 3 桁であれば最初は下から 1 桁目に着目して「0」〜「9」の箱に分類し、次にそのままの順で取り出した後、下から 2 桁目に着目して「0」〜「9」の箱に分類します。そしてまたそのままの順で取り出した後、下から 3 桁目で分類すると全部並んでいます。

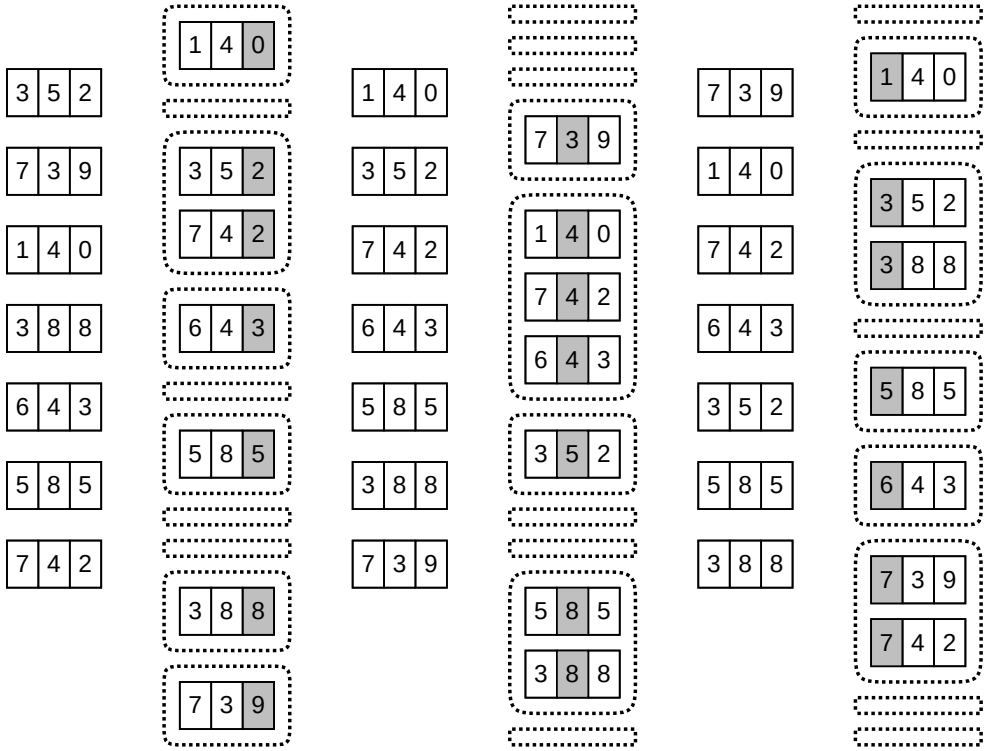


図 4.15: 十進法の場合の基数ソートの例

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

次に 2 進による基数ソートのコードを示します。整列する値のビット数をパラメタで渡しています。[8] 各周回ごとに当該ビットの値に応じてデータを配列 b と c に振り分け、終わったらこの順で a にコピーし戻しています:

```
def radixsort(a, bits)
  b = Array.new(a.length); c = Array.new(a.length)
  bits.times do |pos|
    mask = 2**pos; bc = 0; cc = 0
    a.length.times do |i|
      if (a[i] & mask) == 0
        b[bc] = a[i]; bc = bc + 1
      else
        c[cc] = a[i]; cc = cc + 1
      end
    end
    bc.times do |i| a[i] = b[i] end
```

[8] 10000 までの値だと 14 ビットで済むので 14 を指定すればよいです。

```
      cc.times do |i| a[bc+i] = c[i] end
    end
  end
```

4.7.5 演習 6-2〜5 — 整列アルゴリズムの時間計測

各種整列アルゴリズムの計算時間を N を変えて手元のマシンで計測した結果を 4.2 に示します。

表 4.2: さまざまな整列アルゴリズムの時間計測

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

$O(N^2)$ のアルゴリズムである 単純挿入法、バブルソートは N が大きくなると急激に遅くなって役に立たなくなります。一方、$O(N \log N)$ のマージソートとクイックソートは十万くらいのデータであれば十分実用になると言えます。

ビンソートは極めて速いことは分かりますね。では、このアルゴリズムの時間計算量はどうでしょう。まず、すべてのデータを順に走査するという点では $O(N)$ だと 言えますが、それだけではありません。値の数を E とすると、最後に大きさ E の配列を全部調べながら値を生成するので、このための時間も E が大きいと問題になります。なので、時間計算量は $O(N+E)$ になるわけ です。今回は E が 10,000 なので、途中まではこちらの方が主に問題になります。表 4.2 の一番下に $N+E$ を示しましたが、所要時間がだいたいこれに比例していることが読み取れます。

そして、ビンソートは E 個ぶんの配列を必要とすることも忘れてはいけません。アルゴリズムによっては、大量のメモリを使うことで時間を速くすることができますが、ビンソートはまさにその例です。ビンソートが要する記憶領域は元のデータ数 N と数えるための配列の数 E を併せたものだから、これを「**領域計算量**が $O(N+E)$ である」のように言い ます。つまり、値の範囲が広くなると、ビンソートは領域計算量の点でも不利になるわけでです。

なお、これまでに出て来たアルゴリズムのほとんどは領域計算量 $O(N)$ ですが、マージソートと基数ソートは「別の場所に移してから戻す」ので $O(2N)$ になって います。[9]

最後に基数ソートの時間計算量ですが、キーのビット数ぶんだけ振り分け処理をおこなうので、時間計算量は $O(N \log E)$ ということになりま す。これを $\log E$ が今回は定数 (14) と考えれば、時間計算量は**線形時間**ということになります。実際、表 4.2 をチェックすると所要時間が N にほぼ比例していることが分かります。

ただし、時間そのものはほとんどの場合においてクイックソートよりも劣っています。これはつまり、データが非常に多くなると、先に説明したようにオーダーの差がすべてを支配しますが、それほどでもない場合には定数項の差が無視できず、オーダーの大きいアルゴリズムでも処理時間が短くて済む場合がある、ということを意味しています。たとえば、データが数個しかないのであれば、クイックソートを使うよりも単純選択方を使うほうが適切なわけです。

[9] 基数ソートでは b と c を a と同じサイズで取るので 3 倍と思うかもしれませんが、ケチるなら b と c を 1 つの配列にして「前から」と「後ろから」データを詰めてゆけばよいのです。

4.7.6 演習 6-3 — クイックソートの弱点

先に掲げた quicksort のコードが整列ずみの配列に対しては遅いという弱点を実際に示すには、次のように 2 回ずつ整列してみればよいでしょう:

```
def test
  a = randarray(1000)
  bench(1) do quicksort(a, 0, 999) end
  bench(1) do quicksort(a, 0, 999) end
  a = randarray(2000)
  bench(1) do quicksort(a, 0, 1999) end
  bench(1) do quicksort(a, 0, 1999) end
  a = randarray(3000)
  bench(1) do quicksort(a, 0, 2999) end
  bench(1) do quicksort(a, 0, 2999) end
end
```

これを実行してみると、結果は次のようになりました:

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

確かに、1 回目は $O(N)$ に近い (実際には $O(N \log N)$ のはず) けれど、2 回目はずっと遅くて $O(N^2)$ に近くなっているようです。

これを改良するにはどうしたらいいでしょう? それには、ピボット値を取る時にいつも「端っこ」から取っていたからまずいので、代わりにランダムに取るようにすればよいでしょう: [10]

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

これを実行してみると、上の表と異なり、1 回目でも 2 回目でもほぼ同じ時間で整列が終わるようになっています。

4.8 時間計算量ふたたび

ここまでは整列アルゴリズムを題材に時間計算量をあれこれ考えて来ましたが、ここからは他のアルゴリズムについてもやって見ましょう。時間計算量の求め方を非常に簡単にまとめると、次のようになります。

[10] プログラムの修正は***の 1 行を挿入しただけです。つまりここで、i〜j の範囲の整数 p を 1 つランダムに選び、a[j] と a[p] を交換してから、あとはこれまでと同様に処理するわけです。

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

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

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

実際にプログラムを動かす時の感覚としては、$O(n)$ までは「すごく速い」、$O(n \log n)$ は「まあまあ速い」、$O(n^2)$ は「遅い」、$O(2^n)$ は「ひどく遅いので、小さい n にしか役に立たない」というふうに考えたらよいでしょう。

演習 4-8 以下に出て来る Ruby のメソッドにさまざまな n を与えたときの時間計算量を見積もりなさい。また、実際に bench で掛かる時間を計測して確認しなさい。bench のソースコードは次に再掲する。

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

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

a. n^2 を計算するメソッドその **1**

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

b. n^2 を計算するメソッドその **2**

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

c. n^2 を計算するメソッドその **3**

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

d. 1.0000000001^n を計算するメソッドその **1**

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

e. 1.0000000001^n を計算するメソッドその **2**

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

f. 1.0000000001^n を計算するメソッドその **3** [11]

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

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

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

h. 1〜n の値のすべての順列を生成する (印刷は省略)

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

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

```
    # p(b)
  else
    a.each_index do |i|
      if a[i] != nil
        x = a[i]; a[i] = nil; b.push(x)
        perm1(a, b)
        a[i] = x; b.pop
      end
    end
  end
end
```

4.9 検討 時間計算量の重要性

この箇所では、様々な整列アルゴリズムを提示するとともに、それらの比較を題材として時間計算量について説明しています。

整列はその意図はごく簡潔に表現できる (「昇順に並べる」) わりに、そのアルゴリズムは多様なものが古くから提案されてきていて、1 つのことがらにこれほど様々なアルゴリズムが知られている事項は他に思い当たらないほどです。そのため、「これほどに様々な人が様々な工夫や着想をもとにアルゴリズムを提案してきている」ということを具体的に知ってもらう上でも、最善のテーマであると考えます。

ここでの 1 つの工夫として、乱数によりサイズ N のデータを生成して、それを整列するのに要する時間を計測してもらい、自分で比較検討してもらうことで考えてもらう、という点があります。ただ単に理論上こうです、というよりも、実際に計ってみて「遅さ」を実感することが、問題の本質を納得してもらう上でずっとよいと考えるためです。

そして、最初の段階では、単純選択法、単純挿入法、バブルソートという分かりやすい (しかし明確に違う) 整列アルゴリズムを取り上げ、$O(N^2)$ のアルゴリズムの「遅さ」を実感してもらった後でより速いマージソートやクイックソートを説明し、計測してもらうことで、より良い $O(N \log N)$ 計算量のものが圧倒的に速いことを理解してもらおうとしています。

次に時間計算量の考え方について説明し、結局「最も多く実行されるコードの実行回数のオーダー」が問題であるというところを納得してもらうようにしています (ここが簡単ではないと思いますが)。その後、$O(N)$ のアルゴリズムであるビンソート、基数ソートを紹介することで、計算量が違うと圧倒的に有利であることを再度認識してもらいます。

実は本来一番学んで欲しいことは、指数計算量がいかに遅いかということかも知れませんが (暗号解読の困難さもこのことが土台になっている)、それはこの範囲には入っていません。しかし高校で一部だけ扱うのなら、そこまでここで説明した方がよいのかも知れません。

open question

- ここに出て来る整列アルゴリズムはいずれも、高校生でも十分理解できると思ってよいでしょうか?
- それぞれの整列アルゴリズムを実現しているプログラムについては、どうでしょうか?
- 計算量のところで出て来る log については、理解してもらえるでしょうか?
- 最終的に、計算量とはこういうものだ、という納得が得られるでしょうか? そのために不足していることはあるでしょうか?
- 最後に出て来る演習問題のコードの計算量は求められるようになるでしょうか?

第5章　オブジェクト指向と動的データ構造

5.1　オブジェクト指向とは

皆様はここまでに、配列、レコードなどのデータ構造を用意し、それを操作するメソッドを複数組み合わせてアルゴリズムを実現する、という形でプログラムを作成してきました (図 5.1 左)。最初の方で説明したように、このようなモデルを**手続き型計算モデル**、このモデルに基づくプログラミング言語を**手続き型言語**と言います。

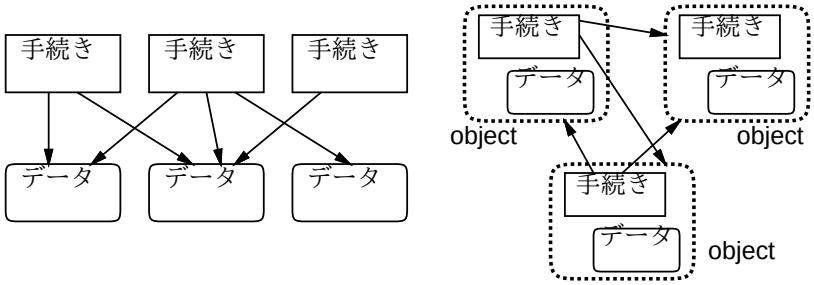


図 5.1: 手続き型モデルとオブジェクト指向

このプログラミングスタイルは長い間主流として使われてきましたが、近年のようにプログラムが大きくなり複雑化してくると、次のような弱点が問題になってきました:

- データ構造と手続きが分離していて、両者の対応が取りにくい。
- 手続きが複雑になると、どの部分が何を行っているかの把握が困難になる。
- 各データ構造がどの手続きからでも (原理的には) アクセスできるため、本来アクセスすべきでないデータ構造に触ってしまうことによるトラブルが起きやすい。

オブジェクト指向 (object-orientation) は、上記の点を克服すべく手続き型モデルを拡張した概念で (図 5.1 右)、プログラムが扱う対象を多様な**もの**、ないし**オブジェクト** (object) として捉える考え方です。

我々が日常扱っている「もの」にはそれぞれ固有の機能や特性があり、我々は「内部構造」には関わらなくてもこれらの「もの」の機能や特性を活用できます。たとえばイスであれば「座る」「高さを調節」「移動させる」などの操作ができますし、ペンであれば「キャップをつける/外す」「描く」などの操作ができ、それぞれ固有の色などもあります。しかし、これらを利用したり参照するのに、イスやペンの内部構造を理解している必要はありません。プログラミングもこれと同様にできれば人間にとってずっと扱いやすくなる、というのがオブジェクト指向の基本的なアイデアです。

今日では多くのプログラミング言語がオブジェクト指向を取り入れています。そのような言語 (**オブジェクト指向言語**) では、手続きとそれが扱うデータが組になっているため対応がつけ易く、個々の手続きは自分の担当しているデータのみを直接扱うため簡潔に保ちやすく、データは対応する手続き以外からは操作されないため、不用意に壊される心配が少なくなります。データを外部から直接アクセスされないようにすることを**カプセル化** (encapsulation) ないし**情報隠蔽** (information hiding) と呼びます。

5.2 クラスとインスタンス

前節で述べたような「もの」を言語上でどのように表すかを考えてみましょう。ものには種類ないし**クラス** (class) があるものと考え、その種類ごとに「どんな性質を持つか」「どのような操作ができるか」を定義していく、というのが 1 つの方法です。このような考え方に従うオブジェクト指向言語を**クラス方式** (class based) のオブジェクト指向言語と呼びます。Ruby、Java、C++ などの言語はクラス方式のオブジェクト指向言語です。

なお、別の方式として**プロトタイプ方式** (prototype based) のオブジェクト指向言語があります。これは、「お手本」となるオブジェクトを (概念的に) コピーして類似したオブジェクトを用意する方式で、JavaScript などが採用しています。

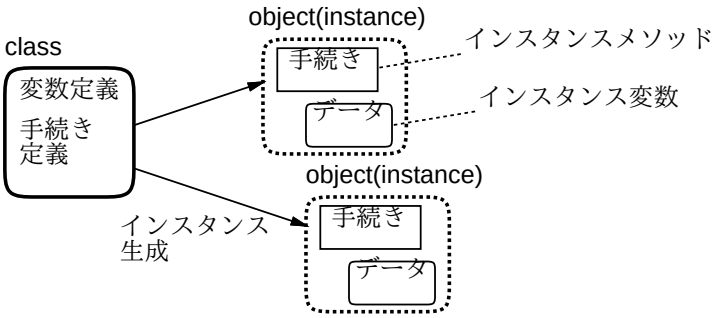


図 5.2: クラスからインスタンスを生成

では、ものの種類の定義、つまり**クラス定義** (class definition) には何が含まれるべきでしょうか (図 5.2)。上述のように、それぞれの「もの」には固有のデータと固有の操作があるので、それを**変数**と**手続き**ないし**メソッド**で表すのが自然な方法です。これらをそれぞれ、**インスタンス変数** (instance variable)、**インスタンスメソッド** (instance method) と呼びます。

ただし、ここで言う変数とメソッドは、これまで使ってきた変数やメソッドとは少し違っています。つまり、あるクラスを定義し、それをもとに「そのクラスに所属するもの」 — オブジェクト指向言語の言葉で言えば**インスタンス** (instance — 実体と呼ぶこともあります) を生成したとすると、クラス内で定義した変数やメソッドは「そのクラスのインスタンスに付随した」ものとなります。

たとえば図 5.3 では、クラス定義 `Xyz` を「ひな型」として 2 つのインスタンスを生成し、変数 `x1` と `x2` に入れています。この時、この 2 つのインスタンスは内部に持っているインスタンス変数群も使用できるメソッド群も同じですが、インスタンスとしては別個、つまりインスタンス変数群はそれぞれ別個になっています。つまりクラス定義に書かれているとおりのインスタンス変数群とインスタンスメソッド群を持つということです。

そして、インスタンスメソッドを呼び出す時には、単にメソッド名を言ったのでは「どのインスタンスに付随する」メソッドかが特定できないので、インスタンス x に対して「x. メソッド名」という形で指定します。これを**メッセージ送信記法** (message sending) と呼びます。ここまでにもこの記法は、配列などさまざまな (Ruby が提供してくれている) オブジェクトの機能を呼び出す時に利用していました。

そして、メッセージ送信記法で「`x1.m0(...)`」「`x2.m1(...)`」のように インスタンスメソッドを呼び出すと、それらのインスタンスメソッドの中でインスタンス変数を参照した時は、それぞれ `x1`、`x2` のインスタンス変数が使われることになります。

たとえば、「犬」というクラスを作って、そこで「名前」「走っている速さ」というインスタンス変数を持たせたとすると、どの犬もこれら 2 つのインスタンス変数を持っているという点は同じですが、そこに格納されている値、つまりそれぞれの犬の名前やそれぞれの犬の走っている速さは、どの犬かによって、つまりインスタンスによって違う、というわけです。

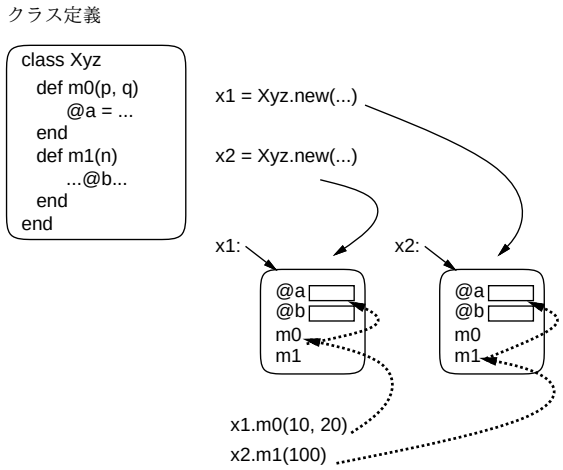


図 5.3: クラスとインスタンス

あと、Ruby ではクラスもオブジェクトなので、クラスに直接付随するメソッドである**クラスメソッド** (class method)、クラスに直接付随する変数である**クラス変数** (class variable) も存在します。クラスメソッドを呼び出す場合はメッセージ送信記法のオブジェクトのところにクラスの名前を指定します。

5.3 Ruby によるクラスの定義

5.3.1 簡単なクラス定義

では Ruby の場合についてクラス定義の方法を説明しましょう。まずクラスは、次の構文により定義します:

```
class クラス名
  ...
end
```

クラス名は必ず英大文字で始めることになっています。そして、この中にメソッド定義を書くと、自動的にインスタンスメソッドになり、メッセージ送信記法で呼び出せるようになります。また、インスタンス変数はこれまでの変数と異なり、名前の最初が「`@`」で始まります。

そして最後に、クラスからインスタンスを作るには「クラス名`.new(...)`」という特別なメソッド呼び出し を使います。この時、もしインスタンスメソッドの中に `initialize` という名前のものがあればそれが呼び出され、その時 `new` に渡したパラメタがそっくりそのまま渡されてきます。つまり名前どおり、初期化のためにこのような仕組みになっているわけです。

ここでは使いませんが、クラスメソッドを定義する場合は「`def` クラス名`.` メソッド名 … `end`」のような `def` をクラスの外側に書いて定義します。また変数名を「`@@`」で始めるとその変数はクラス変数になります。

では、簡単なクラス定義の例を見てみましょう (これは先に説明で使った「犬」をクラスとして定義したものです):

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0
  end
  def talk
```

```
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
end
```

initialize では名前を受け取り、その値でインスタンス変数@name を初期化します。インスタンス変数@speed は 0 に設定します。メソッド talk では自分の名前を喋ります (喋る犬?)。addspeed では渡された値だけスピードを増して、それを表示します。動かしてみましょう:

```
irb> a = Dog.new('pochi')
=> #<Dog:0x81b5b2c @name="pochi", @speed=0.0>
irb> b = Dog.new('tama')
=> #<Dog:0x81b049c @name="tama", @speed=0.0>
irb> a.talk
my name is pochi
=> nil
irb> b.talk
my name is tama
=> nil
irb> a.addspeed(5.0)
speed = 5.0
=> nil
irb> b.addspeed(8.0)
speed = 8.0
=> nil
irb> a.addspeed(10.0)
speed = 15.0
=> nil
```

ポチのインスタンスとタマのインスタンスは別のもので、名前や速度を別個に持っていることがお分かりになると思います。このように「もの」単位で扱えるところが、オブジェクト指向の特徴なのです。

演習 5-1 この例題を打ち込んで動かせ。動いたら、「ほえる」メソッド bark(引数は無し) と、「ほえる回数」を設定するメソッド setcount(引数は回数) を追加せよ。とくに設定しない場合は 3 回ほえるものとする。[1]

演習 5-2 次のような機能と使い方を持つクラスを作成せよ。使用例の通りに使えることを確認すること。

a. 「覚える」機能を持つクラス Memory。put(x) で新しい内容を記憶させ、get で記憶内容を取り出す。

```
irb> m1 = Memory.new            # 作る
=> #<Memory:0x81d59e0 @mem=nil>
irb> m1.put(5)                  # 5 を覚えさせる
=> 5                            # put の返値は何でもいいことにする
```

[1] もちろん、ほえる回数を憶えるインスタンス変数を追加する必要があるはずです。

```
irb> m1.get                    # 取り出す
=> 5                           # 5
irb> m1.get                    # 再度取り出す
=> 5                           # やはり 5
irb> m1.put(10)                # 10 を覚えさせる
=> 10
irb> m1.get                    # 取り出す
=> 10                          # 10
```

b. 「文字列を連結していく」クラス Concat。add(s) で文字列 s を今まで覚えているものに連結する (最初は空文字列から始まる)。get で現在覚えている文字列を返す。reset で覚えている文字列を空文字列にリセット。

```
irb> c = Concat.new            # 作る
=> #<Concat:0x81c7e94 @str="">
irb> c.add("This")             # 追加
=> "This"
irb> c.add("is")               # 追加
=> "Thisis"
irb> c.get                     # 取り出す
=> "Thisis"
irb> c.add("a")                # 追加
=> "Thisisa"
irb> c.reset                   # リセット
=> ""
irb> c.add("pen")              # 追加
=> "pen"
irb> c.get                     # 取り出し
=> "pen"
```

なお、文字列どうしを連結するのは「+」でできます。

c. 「最大 2 つ覚える」機能を持つクラス Memory2。put(x) で新しい内容を記憶させ、get で記憶内容を取り出す。2 回取り出すと 2 回目はより古い内容が出てくる。取り出した値は忘れる。覚えている以上に取り出すと nil が返る。

```
=> #<Memory2:0x80fdab8 @mem2=nil, @mem1=nil>
irb> m2.put(1)    # 1 を入れる
=> 1
irb> m2.put(3)    # 3 を入れる
=> 3
irb> m2.put(5)    # 5 を入れる
=> 5
irb> m2.get       # 取り出す → 5
=> 5
irb> m2.get       # 取り出す → 3
=> 3
irb> m2.get       # 取り出す → nil (2 つまでしか覚えてない)
=> nil
irb> m2.put(7)    # 7 を入れる
=> 7
```

```
irb> m2.put(9)    # 9を入れる
=> 9
irb> m2.get       # 取り出す → 9
=> 9
irb> m2.put(11)   # 11を入れる
=> 11
irb> m2.get       # 取り出す → 11
=> 11
irb> m2.get       # 取り出す → 7
=> 7
```

もし興味があれば、「最大 N 個覚える」もやってみるとよい。

5.3.2 例題: 有理数クラス

今度は、もう少し有用なものを作ってみます。これまで、実数の計算には誤差がつきものだという説明をしてきましたね。具体的には、浮動小数点計算では割り切れない除算は循環小数になるので、必ず誤差が生じます。

そこで代わりに、数値を $\frac{\text{分子}}{\text{分母}}$ という形で保持すれば誤差なく除算結果を保持できるはずです (もちろん、加減算の時は通分して計算し、最後に約分します。そしてもちろん、$\sqrt{2}$ や π などの無理数は扱えません)。

そのような**有理数** (rational number) クラスを作ってみましょう。このクラスでは、インスタンス変数`@a`と`@b`に分子と分母をそれぞれ保持するようにしています:

```
class Ratio
  def initialize(a, b = 1)
    @a = a; @b = b
    if b == 0 then @a = 1; return end
    if a == 0 then @b = 1; return end
    if b < 0 then @a = -a; @b = -b end
    g = gcd(a.abs, b.abs); @a = @a/g; @b = @b/g
  end
  def getDivisor
    return @b
   end
  def getDividend
     return @a
  end
  def to_s
    return "#{@a}/#{@b}"
  end

  def +(r)
    return Ratio.new(@a*r.getDivisor+r.getDividend*@b, @b*r.getDivisor)
  end
  def gcd(x, y)
    while true do
      if x > y then x = x % y; if x == 0 then return y end
      else          y = y % x; if y == 0 then return x end
```

```
        end
      end
    end
  end
```

initialize のパラメタに代入が書いてありますが、これは**デフォルト値** (default value) つまりそのパラメタを省略した場合はこの値を使ってねという意味になります。ですから、「Rational.new(3)」で $\frac{3}{1}$ になるわけです。initialize の中身がごちゃごちゃしていますが、これは (1) 分母が 0 の時 (不定) は分子を 1 とする、(2) 分母は 0 でないなら常に正とする (負の数は分子が負)、(3) 値ゼロは $\frac{0}{1}$ で表す、(4) 必ず既約分数にする、という**正規化** (normalization — なるべく形を揃えること) を行っているからです。

分母だけ、分子だけを取り出したい場合のために、メソッド get_divisor、get_dividend を用意しました。このような、インスタンス変数をアクセスするだけのメソッドのことを**アクセサ** (accessor) と呼びます。Ruby ではアクセサがなければインスタンス変数の内容は外部からは参照できません。これにより、カプセル化が実現され、また後で内部のデータ表現を変えた時にも外部に影響が及ばないで済みます。

また、文字列への変換メソッド to_s も用意しました。これは puts などによる打ち出しなどの時に自動的に「a/b」という形の文字列を生成できるので、用意しておくと便利です。そして、演算としてはとりあえず加算だけを用意しました。加算は「+」で表したいので、メソッド名を「+」にしてあります。このようにして、演算子を定義できるのは Ruby の特徴の 1 つです。ただし、C++ などの言語でも演算子定義が可能です。

では動かしてみましょう:

```
irb> a = Ratio.new(3,5)
=> #<Ratio:0x81f978c @b=5, @a=3>
irb> puts a
3/5
=> nil
irb> b = Ratio.new(8,7)
=> #<Ratio:0x81f00d8 @b=7, @a=8>
irb> puts b
8/7
=> nil
irb> puts a+b
61/35
=> nil
```

確かに、通分して計算してくれていますね。なお、なぜ puts で打ち出しているかというと、puts は引数を文字列に変換して出力するため to_s を呼んでくれるからです。単に irb の機能で打ち出させるのだと、オブジェクトを表す「#<Ratio ....>」というのが表示されてしまいます。

演習 5-3 有理数クラスをそのまま打ち込んで動かせ。動いたら、四則の他の演算も追加し、動作を確認せよ。できれば、これを用いて浮動小数点では正確に行えない「実用的な」計算が正確にできることを確認してみよ。

演習 5-4 **複素数** (complex number) を表すクラス Comp を定義し、動作を確認せよ。これを用いて何らかの役に立つ計算をしてみられるとなおよい。

演習 5-5 クラス定義を活用した「面白い」Ruby プログラムを作って動かせ。面白さの定義は各自に任されるものとする。

5.4 動的データ構造/再帰的データ構造

5.4.1 動的データ構造とその特徴

データ構造 (data structure) とは「プログラムが扱うデータのかたち」を言います。ここでは、プログラムの実行につれて構造を自在に変化させられる、**動的データ構造** (dynamic data structure) について学びます。これに対し、ここまでに扱ってきたプログラムのように、それぞれの変数に決まった形のデータが入っていて、全体の形が変わらないものを**静的データ構造** (static data structure) と呼びます。一般に、**静的** (static) とは「プログラム記述時に決まる」、**動的** (dynamic) とは「プログラム実行時に決まる」という意味があります。

動的データ構造は、プログラム言語が持つ「データのありかを指す」機能を用いて作られます。Ruby では、複合型 (配列、レコード等) や一般のオブジェクトの値は実際は、それらのデータやオブジェクトのありかを指す**参照**になっているので、これを利用します。既に忘れている人がいるかもしれませんが、**レコード**とは複数のフィールドが集まったデータであり、Ruby では `Struct.new` においてフィールド名を表す記号を必要なだけ指定して定義するのでしたね。

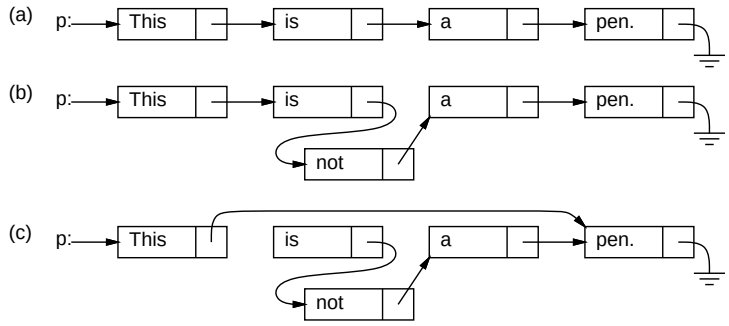


図 5.4: 単連結リストの動的データ構造

たとえば、次のレコードを見てみましょう:

```
Cell = Struct.new(:data, :next)
```

これは 2 つのフィールド `data` と `next` を持つ `Cell` という名前のレコードを定義していますが、ここで各セルのフィールド `next` に「次」のセルへの参照を入れることで、「数珠つなぎ」の動的データ構造を作ることができます (図 5.4(a))。[2] このような「数珠つなぎ」の構造のことを**単連結リスト** (single linked list) ないし**単リスト**と呼びます。

この `Cell` の使い方は、各 `Cell` の `next` がまた `Cell` になっていて、自分の中に自分が入っているように思えます。これは再帰関数と同様で、このようにデータ型 (構造) の中に自分自身と同じデータ型 (構造) への参照を含むものを**再帰的データ構造** (recursive data structure) と呼びます。実際には自分自身が入っているわけではなく図 5.4 のように「同種のデータへの参照」が入っているだけですから、何ら問題はありません。

一番最後のところ (アース記号で表している) は「何も入っていない」という印である **nil** が入っています。このあたりも、「簡単な場合は自分自身を呼ばずにすぐ値が決まる」再帰関数とちょっと似ていますね。

ところで、動的データ構造だと何がよいのでしょうか? たとえば、図 5.4(a) で途中に単語「not」を入れたくなったとします。文字列の配列であれば、途中に挿入するためには後ろの要素を 1 個ずつずらして空いた場所に入れる必要があります。しかし、単連結リストでは、矢線 (参照) を (b) のようにつけ替えるだけで挿入ができてしまうのです。逆に、数単語削除したいような場合も、(c)

[2] 本当はフィールド `data` も文字列オブジェクトを参照しているので、文字列を箱の外に描いて矢線で指させるべきなのですが、ごちゃごちゃして見づらくなるのでここでは箱の中に直接描いていています。

のように参照のつけ換えで行えます。このように、動的データ構造は柔軟な構造の変更が行えるという特徴を持っています。

参照のつけ替えは、具体的にはどうすればいいのでしょう? 参照というのは要するに「場所を示す値」なので、その値をコピーすることは「矢印の根本を別の場所にコピーする (矢印自体も 2 本になる) と考えればいいのです。たとえば、図 5.5 では、`p.next.next` というのは B の箱の `next` フィールド、`p.next` は A の箱の `next` フィールドなので、「`p.next = p.next.next`」で B の箱を迂回して A の箱の `next` フィールドに C の箱を指させることになります。参照を入れてある単独変数 (この例では `p`) などでも同様です。

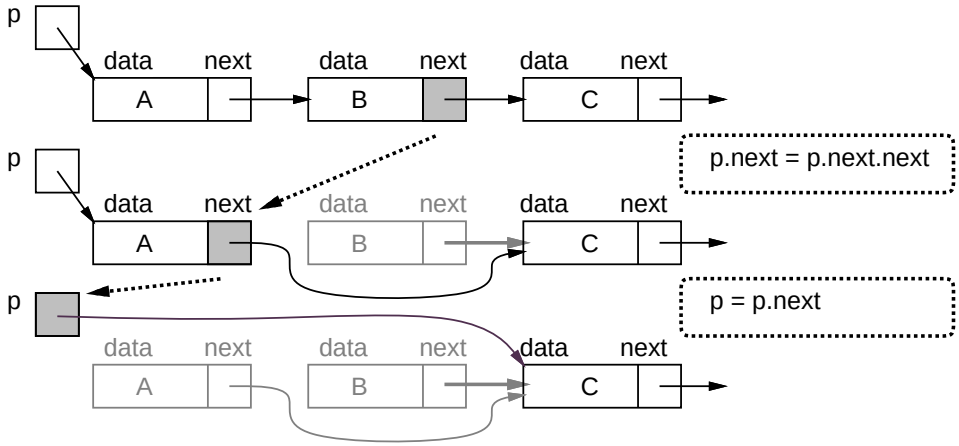


図 5.5: 参照のつけ替え

ときに、図 5.4 や 5.5 で使わなくなった箱はどうなるのでしょう? Ruby では、使われなくなったオブジェクトの領域は**ごみ集め** (garbage collection) と呼ばれる機構によって自動的に回収され、再利用されます。「使っている」かどうかは、どれかの変数からたどれるかどうかで決まります。たとえば図 5.4 の場合は、先頭のセルを変数 `p` が指していて、ここからたどれるセルは「使っている」と見なされるのです。

5.4.2 例題: 単連結リストを使ったエディタ

ではここで、単連結リストを使った例題として、簡単な**テキストエディタ** (text editor) を作ってみましょう。「簡単」なので、編集に使うコマンドは次のものしかありません:

- 「`i` 文字列」 — 文字列を新しい行として現在位置の直前に挿入する。
- 「`d`」 — 現在位置の行を削除する。
- 「`t`」 — 先頭行を表示し、そこを現在位置とする。
- 「`p`」 — 現在位置の内容を表示する。
- 「`n`」または改行 — 現在位置を次の行へ移しその行を表示する。
- 「`q`」 — 終了する。

実際にこれを使っている様子を示します (すごく面倒そうですが、実際にこういうプログラムを使ってファイルの編集をしていた時代は実在しました):

```
>iThis is a pen.      ←挿入
>iThis is not a book. ←挿入
>iHow are you?        ←挿入
>t                    ←先頭へ
  This is a pen.
```

```
>                    ←次の行
  This is not a book.
>                    ←次の行
  How are you?
>                    ←次の行
  EOF                ←おしまい
>t                   ←再度先頭へ
  This is a pen.
>iI am a boy.        ←挿入
>                    ←次の行
  This is not a book.
>iWho are you?       ←挿入

>t                   ←再度先頭へ行き全部見る
  I am a boy.
>
  This is a pen.
>
  Who are you?
>
  This is not a book.
>
  How are you?
>
 EOF
>q                   ←おしまい
```

これをこれから実現してみましょう。

5.4.3 エディタバッファ

以下では、単リストのデータ構造を先頭や現在位置などの各変数も含めてクラスとしてパッケージします:

```
class Buffer
  Cell = Struct.new(:data, :next)
  def initialize
    @tail = @cur = Cell.new("EOF", nil)
    @head = @prev = Cell.new("", @cur)
  end
  def atend
    return @cur == @tail
  end
  def top
    @prev = @head; @cur = @head.next
  end
  def forward
    if atend then return end
    @prev = @cur; @cur = @cur.next
```

```
    end
    def insert(s)
      @prev.next = Cell.new(s, @cur); @prev = @prev.next
    end
    def print
      puts("  " + @cur.data)
    end
  end
```

レコード定義もクラスに入れることにしました。また、「1 つの値を複数箇所に代入する」のに=を連続して書いてみました。もちろん、2 つの代入に分けても一向に構いません。

このクラスでは、単リストのセルを上記の Cell レコードであらわし、これを指すための変数として次の 4 つを使っています:

- @head — 一番先頭に「ダミーの」セルを置き、そのセルを常にこの変数で指しておく (ダミーがあると、先頭行を削除するのを特別扱いしないで済ませられるため、プログラムの作成が楽になります)。
- @cur — 「現在行」のセルを指しておく。
- @prev — 「現在行の 1 つ前」のセルを指しておく (挿入や削除の時にこの変数があるとコードを書くのが楽です)。
- @tail — 一番最後にも「ダミーの」セルを置き、そのセルをこの変数で指しておく (表示することがあるので内容は「EOF」(end of file) としてあります)。

initialize では 2 つのダミーセルと上記 4 変数を用意します。head の次が tail であるように Cell.new にパラメタを渡していることにも注意。

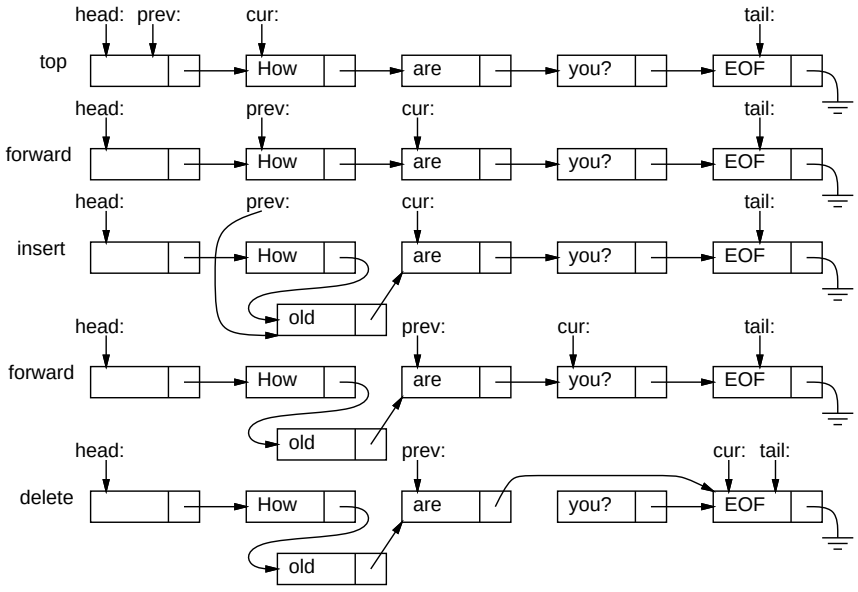


図 5.6: エディタバッファに対する操作

では次に、メソッドを見てみましょう。図 5.6 に、適当なバッファの状態で操作を行った例を示します (最後の delete は課題の参考用です)。atend は現在行が末尾にあるか (@tail と等しいか) を調べます。top は@prev と@cur を先頭に設定します。forward は@prev と@cur を 1 つ先に進めますが、現在行が@tail の時は「果て」なので何もしません。print は現在行の文字列を表示し

ます。insert は新しいセルが@prev となり、元の@prev のセルの次が新しいセル、新しい@prev の次が@cur のセルとなります。これを動かした様子を見てみましょう: [3]

```
irb> e = Buffer.new
=> ...
irb> e.insert('abc')
=> ...
irb> e.insert('def')
=> ...
irb> e.insert('ghi')
=> ...
irb> e.top
=> nil
irb> e.print
   abc
=> ...
irb> e.forward
=> ...
irb> e.print
   def
=> nil
```

確かに文字列が順序どおり挿入でき、それをたどることができています。このクラスは「行の挿入や削除が自在にできる機能を持ったオブジェクト」を作り出しています。内部では込み入ったデータ構造を管理していますが、その様子はクラスの外側からは見えません。このように内部構造は**カプセル化**によって隠して、操作を通じて整合性のある汎用的な機能を提供するものを、**抽象データ型** (abstract data type — ADT) と呼びます。

クラス方式のオブジェクト指向言語では、抽象データ型はクラスによって定義するのが自然です。前章に出てきた有理数クラスや複素数クラスも抽象データ型の例だといえます。

演習 5-6 図 5.7 は「How」という行と「are」という行の間に「old」という行を挿入する様子 (A → B)、および、「old」「are」「you?」という 3 行のうちから「are」を削除する様子 (B → C) を示しています。資料 (ないし同じ図を描き写したもの) の上に赤ペンで次のものを記入しなさい。

a. (A) の図の上に、(A) から (B) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

b. (B) の図の上に、(B) から (C) につながりが変化するための矢線のつけ替えを、(1)、(2) のようにつけ替えを行う順番つきで記入しなさい。ただし、矢印のつけ替えを行う時には、その出発点がどれかの変数そのものであるか、またはどれかの変数から矢線でたどれる箱であることが必要である。

演習 5-7 クラス Buffer を打ち込み、動作を確認せよ。動いたら、以下の操作 (メソッド) を追加してみよ。

a. 現在行を削除する (EOF 行は削除しないように注意…)

[3] irb の結果表示はうるさいので省略しています。

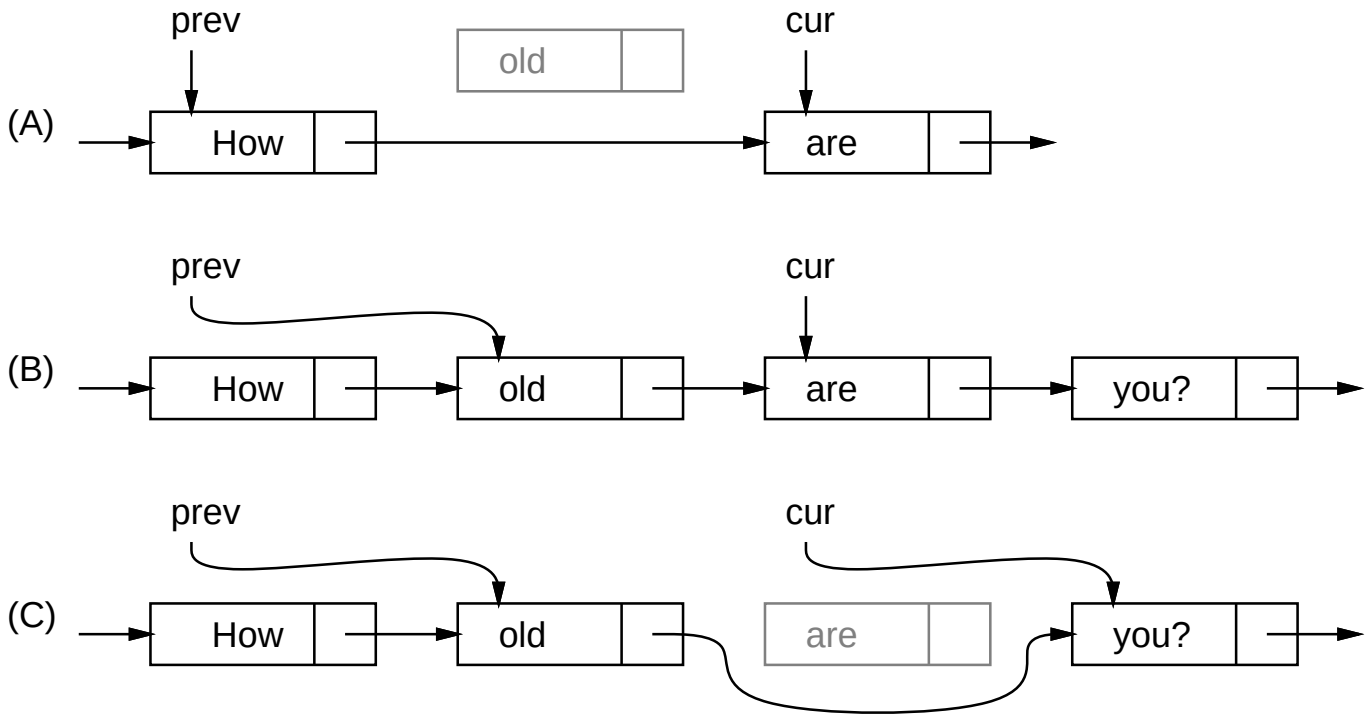


図 5.7: 挿入と削除のようす

b. 現在行と次の行の順序を交換する (EOF は交換しないように…)

c. 1 つ前の行に戻る (実は大変かも)

d. すべての行の順番を逆順にする (かなり過激)

演習 3 単方向リストでは各セルが「次」の要素への参照だけを保持していたが、各セルが「次」と「前」2 つの参照を持つようなリストもある。これを**双連結リスト** (double linked list) ないし**双リスト**と呼ぶ。編集バッファの双リスト版を作り、その得失を検討せよ。[4]

5.5 エディタドライバ

5.5.1 コマンド型プログラムとしてパッケージする

バッファのメソッドを呼ぶだけでも編集はできますが、面倒です。先にお見せしたように「コマンド (+パラメタ)」ですらすら編集ができるように、エディタとして動作するコードも作ってみました。内容はとても簡単で、バッファを生成し、その後無限ループでプロンプトを出し、1 行読んでは先頭の 1 文字でどのコマンドを実行するか枝分かれします (コメントにしてあるのはあなたが作るか、後で機能を追加するためのものです):

```
def edit
  e = Buffer.new
  while true do
    printf(">")
    line = gets; c = line[0..0]; s = line[1..-2]
    if c == "q"    then return
    elsif c == "t" then e.top; e.print
```

[4] ちなみに、双リストなら単リストでの「頭」と「最後」を 1 つで兼ねることもできます。無理に兼ねなくてもよいですが。

```
      elsif c == "p" then e.print
      elsif c == "i" then e.insert(s)
  #   elsif c == "r" then e.read(s)
  #   elsif c == "w" then e.save(s)
  #   elsif c == "s" then e.subst(s); e.print
  #   elsif c == "d" then e.delete
      else                e.forward; e.print
      end
    end
  end
```

文字列の一部を取り出すには「[位置..位置]」という添字指定を使います。1文字だけの場合でも文字列として取り出したい場合は位置を2つ指定するので、先頭の文字は `line[0..0]` で取り出しているわけです。なお、Ruby 1.9からは位置1つだけでも文字列として取り出せるようになりましたが、それ以前は位置1つだけだと「**文字コード** (character code) を表す数値」が返されます。

`i`(insert) コマンド等では、2文字目から最後の文字の手前まで (最後の文字は改行文字) も必要なので、これも取り出しています。Rubyでは文字列や配列中の位置として負の整数を指定すると末尾からの位置指定になります。

どのコマンドでもない場合 (や改行だけの場合) はいちばんよく使う「1行進んで表示」にしました。

演習 5-8 エディタドライバを打ち込んで先のクラスと組み合わせて動作を確認せよ。動いたら以下のような改良を試みよ (クラス側を併せて改良しても、このメソッドだけを改良しても、どちらでも構いません。文字列を数値にする必要が生じたら、メソッド `to_i` を使ってください)。

a. 演習1で追加した機能が使えるようにコマンドを増やす。
b. 現在行の「次に」新しい行を追加するコマンド「a」を作る (追加した行が新たな現在行になるようにしてください)。
c. 現在行の内容をそっくり打ち直すコマンド「r」を作る。
d. 「g 行数」で指定した行へ行くコマンド「g」を作る。
e. コマンド「p」を「p 行数」でその行数ぶん打ち出すように改良 (その際、できれば現在位置は変更しないほうが望ましいです)。
f. その他、自分が使うのに便利だと思うコマンドを作る。

5.5.2 文字列置換とファイル入出力

せっかくエディタができたのに、行内の置き換えとかファイルの読み書きができないと実用になりませんから、これらを一応解説しておきます。

まず、行内の置き換は「s/α/β/」により現在行中の部分文字列αをβに置き換えるというコマンドにしました。エディタドライバからはバッファのメソッド `subst` を呼ぶだけとしたので、こちらの中身を示します:

```
def subst(str)
  if atend then return end
  a = str.split('/')
  @cur.data[Regexp.new(a[1])] = a[2]
end
```

文字列のメソッド split は、渡されたパラメタ「/」のところで文字列を分割した配列を返します。その 1 番目を Regexp(パターン) オブジェクトに変換して文字列に添字アクセスすると、そのパターンの箇所があれば、代入によりそこを別の文字列に置き換えられます。

ファイルの読み書きみは、＃ 5 で学んだ open でファイルを開き、読む場合は付属ブロック内でそのファイルの各行を insert、書く場合は逆にバッファの各行をファイルに puts で書き出します:

```
def read(file)
  open(file, "r") do |f|
    f.each do |s| insert(s) end
  end
end
def save(file)
  top
  open(file, "w") do |f|
    while not atend do f.puts(@cur.data); forward end
  end
end
```

演習 5-9 自分が改良したエディタでどれか 1 課題ぶん全部の編集を行い、体験を述べよ。エディタの機能として何があれば必要十分なのか、エディタの使いやすさは何によって決まるかについて考察すること。[5]

演習 5-10 動的データ構造を活用した、何か面白いプログラムを作れ。面白さの定義は各自に任されます。

5.6 演習問題解説 (一部)

5.6.1 演習 5-1 — クラス定義の練習

この演習はメソッドとインスタンス変数が追加できればよいということで、コードだけ掲載しておきましょう:

```
class Dog
  def initialize(name)
    @name = name; @speed = 0.0; @count = 3
  end
  def talk
    puts('my name is ' + @name)
  end
  def addspeed(d)
    @speed = @speed + d
    puts('speed = ' + @speed.to_s)
  end
  def setcount(c)
    @count = c
  end
  def bark
```

[5] なお、エディタのバグによりせっかく作ったプログラムがぐちゃぐちゃになるなどの被害に逢ったとしても、当局は一切関知しませんので、そのつもりでお願いします。

```
    @count.times do puts('Vow! ') end
  end
end
```

5.6.2 演習 5-2 — 簡単なクラスを書いてみる

この演習はクラスの書き方の練習みたいなものなので、見ていただけば十分でしょう。Memory2はちょっと頭を使う必要がありますかね。

```
class Memory
  def initialize()
    @mem = nil
  end
  def put(x)
    @mem = x
  end
  def get()
    return @mem
  end
end

class Memory2
  def initialize()
    @mem2 = @mem1 = nil
  end
  def put(x)
    @mem2 = @mem1; @mem1 = x
  end
  def get()
    x = @mem1; @mem1 = @mem2; @mem2 = nil; return x
  end
end

class Concat
  def initialize
    @str = ""
  end
  def add(s)
    @str = @str + s
  end
  def get()
    return @str
  end
  def reset()
    @str = ""
  end
end
```

5.6.3 演習 5-3 — 有理数クラス

この演習も Ratio クラスのメソッドを「同様に」増やせばよいだけなので、難しくはありません。追加するメソッドだけ掲載します:

```
def -(r)
  return Ratio.new(@a*r.get_divisor-r.get_dividend*@b,
                   @b*r.get_divisor)
end
def *(r)
  return Ratio.new(@a*r.get_dividend, @b*r.get_divisor)
end
def /(r)
  return Ratio.new(@a*r.get_divisor, @b*r.get_dividend)
end
```

要は、引き算は足し算と同様、乗算は分母どうし掛け、除算はひっくり返して掛けるということですね。

5.6.4 演習 5-4 — 複素数クラス

複素数も 2 つの値 (実部、虚部) の組なので、有理数によく似ています。ただし個々の値として整数でなく実数を使います。演算はちょっと面倒 (とくに除算) ですが、作る時に約分とか分母が 0 とか考えなくてよい部分は簡単になります:

```
class Comp
  def initialize(r = 1.0, i = 0.0)
    @re = r; @im = i
  end
  def get_re
    return @re
   end
  def get_im
     return @im
  end

  def to_s
    if @im < 0 then
      return "#{@re}#{@im}i"
    else
      return "#{@re}+#{@im}i"
    end
  end
  def +(r)
    return Comp.new(@re + r.get_re, @im + r.get_im)
  end
  def -(r)
    return Comp.new(@re - r.get_re, @im - r.get_im)
  end
  def *(r)
```

```
    return Comp.new(@re*r.get_re - @im*r.get_im,
                    @im*r.get_re + @re*r.get_im)
  end
  def /(r)
    norm = (r.get_re**2 + r.get_im**2).to_f
    return Comp.new((@re*r.get_re + @im*r.get_im) / norm,
                    (@im*r.get_re - @re*r.get_im) / norm)
  end
end
```

to_s でヘンなことをやっているのは、「$a+bi$」の形で表示させようとした時、虚数部が負の場合には「$a-bi$」にしたいためです。

5.6.5 演習 5-7 — エディタバッファのメソッド追加

この演習については、メソッドのみだけ掲載します。まず削除:

```
def delete
  if atend then return end
  @cur = @prev.next = @cur.next
end
```

これは前回授業でもかなり説明しましたが、要は (1) 最後の EOF は消さないようにする、(2)@cur は現在行の 1 つ先にする、(3)@prev の「次」も同じく現在行の 1 つ先にする、ということですね。どれかが足りないとおかしくなるので注意。次に交換です:

```
def exch
  if atend || @cur.next == @tail then return end
  a = @prev; b = @cur.next; c = @cur; d = @cur.next.next
  a.next = b; b.next = c; c.next = d; @cur = b
end
```

このように込み入ったつなぎ換えは、作業変数を使った方が間違えないで済みます。まず現在行か次の行が「おしまい」だったら交換できないのでそれを除外し、あとは最終的に並ぶ 4 つのセル (中央の 2 つが交換) を変数 a、b、c、d に入れて、つなぎ直し、@cur を変更します。

次は 1 つ戻るですが、せっかくある程度メソッドが作ってあるわけですから、「@prev を覚えておき、先頭に行ってから現在行が覚えておいた行になるまで 1 行ずつ進む」方法で作ってみました:

```
def backward
  if @prev == @head then return end
  a = @prev; top; while @cur != a do forward end
end
```

全部反転はちょっと大変そうですね。要は、先頭から順にたどりながら、今まで「前→後」の対だったものを「後←前」の順になるように参照をつなぎ換える (ただし先頭と末尾はそれなりに対処)、という方針です:

```
def invert
  top; if atend then return end
  a = @cur; b = @cur.next; a.next = @tail
  while b != @tail do c = b.next; b.next = a; a = b; b = c end
  @head.next = a; top
end
```

先頭と末尾の対処はまず最初に先頭の次を@tail にし、最後にループを抜けてきた時のセルを先頭 (@head の次) にする、ということです。なお、バッファ内に 1 行しかない時はループ周回数が 0 で、その時もちゃんと動作することに注意。双方向リストの実現例は長くなりますし、やってみたい人のお楽しみなので省略させてください。

5.6.6 演習 5-9 — エディタの機能強化

まず、「指定した行へ行く」機能はバッファ側で「何行目」を管理するのがよいので、エディタバッファ全体を示します。基本的に、インスタンス変数@lineno を追加して、現在行が変化するメソッドではこれを更新します。invert みたいにぐちゃぐちゃにいじる場合は最後に top を呼ぶので、ここで 1 にリセットされるから考える必要はありません。そしてこれがあれば backward はもっと簡単になるので、これも直しました。行番号を間違いなく維持するのはきわどそうに見えますが、@lineno をアクセスするのもバッファ内容や現在位置を変更するのも Buffer の中だけなので、この中できちんと処理すれば大丈夫です。つまり、オブジェクト指向の持つ**カプセル化**の機能によって、プログラムが正しく構成し易くなっているわけです。

```
  class Buffer
    Cell = Struct.new(:data, :next)
    def initialize
      @tail = @cur = Cell.new("EOF", nil)
      @head = @prev = Cell.new("", @cur)
      @lineno = 1
    end
    def getlineno
      return @lineno
    end
    def goto(n)
      top; (n-1).times do forward end
    end
    def atend
      return @cur == @tail
    end

    def top
      @prev = @head; @cur = @head.next; @lineno = 1
    end
    def forward
      if atend then return end
      @prev = @cur; @cur = @cur.next; @lineno = @lineno + 1
    end
    def insert(s)
      @prev.next = Cell.new(s, @cur)
      @prev = @prev.next; @lineno = @lineno + 1
    end
    def print
      puts("  " + @cur.data)
    end
# delete、exch は上掲のとおり。backward は以下のように変更
    def backward
```

```
      goto(@lineno - 1)
    end
    def invert
      top; if atend then return end
      a = @cur; b = @cur.next; a.next = @tail
      while b != @tail do
        c = b.next; b.next = a; a = b; b = c
      end
      @head.next = a; top
    end

    def subst(str)
      if atend then return end
      a = str.split('/')
      @cur.data[Regexp.new(a[1])] = a[2]
    end
    def read(file)
      open(file, "r") do |f|
        f.each do |s| insert(s) end
      end
    end
    def save(file)
      top
      open(file, "w") do |f|
        while not atend do f.puts(@cur.data); forward end
      end
    end
  end
```

エディタドライバ側も一応示します (「位置変更しない指定行数プリント」も、最初に行番号を覚えて、印刷し終わったらそこに戻ればよいので簡単です):

```
  def edit
    e = Buffer.new
    while true do
      printf(">")
      line = gets; c = line[0..0]; s = line[1..-2]
      if c == "q"    then return
      elsif c == "t" then e.top; e.print
      elsif c == "p" then
        e.print; l = e.getlineno;
        s.to_i.times do e.forward; e.print end; e.goto(l)
      elsif c == "i" then e.insert(s)
      elsif c == "r" then e.read(s)
      elsif c == "w" then e.save(s)
      elsif c == "s" then e.subst(s); e.print
      elsif c == "d" then e.delete
      elsif c == "x" then e.exch
      elsif c == "b" then e.backward
```

```
      elsif c == "v" then e.invert
      elsif c == "a" then e.forward; e.insert(s); e.backward
      elsif c == "c" then e.delete; e.insert(s); e.backward
      elsif c == "g" then e.goto(s.to_i)
      else                e.forward; e.print
      end
    end
  end
```

5.7 検討 オブジェクトと動的データ構造

オブジェクト指向は入門向けの授業ではやや難易度が高いのですが、今日の実用ソフトウェアはオブジェクト指向が前提のものが多いため、できれば取り扱って置きたいところです。

ここではありがちな例題ですが「動物のクラス」で原理を説明した後、パズル的な問題を用いてインスタンス変数、メソッドを自分で書く練習をしてらうようになっています。その後、有理数の例題でもう少し有用な例を見てもらい、その後でそれと類似したものを作る演習を配置しています。

動的分配や継承なども重要な概念なのですが、この授業の範囲内ではそれらは一応の説明はあるものの、少ししか出てきません (今回は分量の関係で省略)。これは、これらの機能が有用になるのはもう少し複雑で大規模なプログラムになったときなので、簡単な例題でその有用性を実感してもらうのは少し難しいと考えるからです (例題としては抽象構文木クラスを使っています)。

次に動的データ構造の話題がありますが、これはここまでに沢山扱ってきた配列によるデータ構造とは大幅に違うものを体験してもらいたいという意図で取り上げています。題材は単連結リストですが、パズル的な要素もあり、また「エディタの内部構造を実装する」という実用に近い側面もあるので、興味を持ってもらいやすいかと思います。

最初は単連結リストと単独で操作しますが、その次の段階では単連結リストを 1 つのクラス内にパッケージして、行挿入や行削除などのメソッドを実装することで、1 つのクラスを完結した機能単位として設計し実装するという感覚を持ってもらうように務めています。また、この段階で新たな機能を追加する場合、抽象データ型の考えによりうまく工夫できることも体験してもらえるように演習を設計しています。

最後に、バッファをコマンド経由で操作することで、ある程度実用的なエディタプログラムの例を示しています。今は実用ソフトウェアの複雑度が高いのでなかなか実用のものを見せるのは難しいのですが、CUI であればこの程度でも有用なものが作れるという実例としてよいと考えます。

open question

- オブジェクト指向についてどの程度まで学ぶことが必要でしょうか、または望ましいでしょうか?
- 抽象データ型的な考え方 (手続きとデータをパッケージする) を強調するという選択はどうでしょうか?
- 基本的な考え方の部分 (インスタンス、インスタンス変数、メソッド) に絞り、継承やポリモルフィズムは取り上げないという選択はどうでしょうか?
- クラス定義の練習問題はどの程度のものが望ましいでしょうか?
- 動的データ構造 (ポインタ) についてどの程度まで学ぶことが必要でしょうか、または望ましいでしょうか?
- 一番易しい単連結リストに絞るという選択についてはどうでしょうか?

- 単連結リストの挿入・削除などを練習するという方針についてはどうでしょうか?
- 単連結リストをクラス内にパッケージしてさまざまな操作を構築していくというやり方はどうでしょうか?
- 行エディタという題材についてはどうでしょうか?

まとめ

本講座では大変駆け足でしたが、大学1年次理系クラス選択科目「情報科学」で扱っている内容のおよそ半分までを紹介するとともに、実際にいくつかの演習を体験していただき、また要所ごとに教える際の工夫や分かっていない議論点について紹介してきました。本講座が先生がたに何らかの参考となれば嬉しく思います。