「情報科学」+「プログラミング教育の考え方」

久野　靖[1]

2014.8.5

[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 2014」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」を兼ねる形で実施します。

本資料の内容の主要部分は、久野が東京大学で毎年実施している「情報科学」の内容のうち、おおよそ前半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。ただし、内容の中には受講される皆様がよくご存じのことも含まれていると思いますので、そのような内容の説明はスキップします。

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きます。

> **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　基本

1.1　プログラミングとモデル化

1.1.1　モデル化とコンピュータ

モデル (model) とは、何らかの扱いたい対象があって、その対象全体をそのまま扱うのが難しい場合に、その特定の側面 (扱いたい側面) だけを取り出したものを言います。

たとえば、プラモデルであれば飛行機や自動車などの「大きさ」「重さ」「機能」などは捨ててしまい、縮尺/縮小して「形」「色」だけを取り出したもの、と言えます。ファッションモデルであれば、さまざまな人が服を着る、その「様々さ」を捨てて特定の場面で服を見せる、という仕事だと言えます (もちろんそこには服をよく見せるという意図はあるでしょうけれど)。

コンピュータで計算をするのに、なぜモデルの話をしているのでしょう? それは、コンピュータによる計算自体がある意味で「モデル」だからです。たとえば、「三角形の面積を求める」という計算を考えてみましょう。底辺が 10cm、高さが 8cm であれば

$$\frac{10 \times 8}{2} = 40(cm^2)$$

ですし、底辺が 6cm、高さが 5cm であれば

$$\frac{6 \times 5}{2} = 15(cm^2)$$

です。「電卓」で計算するのなら、実際にこれらを計算するようにキーを叩けばよいですね:

[1] [0] [×] [8] [÷] [2] [=]

しかし、コンピュータでの計算はこれとはちょっと違っています。なぜかというと、コンピュータは非常に高速に計算ができるし、また高速に計算するためのものなので、いちいち人間が「計算ボタン」を押していたら人間の速度でしか計算が進まず意味がないからです。

具体的には、「どういうふうに計算をするか」という**手順** (procedure) を予め用意しておき、実際に計算するときは**データ** (data) を与えてそれからその手順を実行させるとあっという間に計算ができる、というふうになっているのです。そしてこの手順が**プログラム** (program) なのです。

これを実現するためには、計算の手順とデータを分けることが必要です。たとえば面積の計算だったら、手順は

[☆] [×] [◇] [÷] [2] [=]

みたいに書いてあって、あとで「☆は 10、◇は 8」というデータを与えて一気に計算する、みたいにします。[1]これを捉え直すと、「個々の三角形の面積の計算」から「具体的なデータ」を取り除いた「計算のモデル」が手順だ、ということになります。[2]

コンピュータでの計算はモデル、と言うのにはもう 1 つ別の意味もあります。三角形は 3 つの直線 (正確に言えば線分) から成るわけですが、世の中には完璧な直線など存在しませんし、まして鉛筆で紙の上に引いた線は明らかに「幅」を持っていて縁はギザギザ曲がっています。また、10cm

---

[1]もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができますね。

[2]モデルを作る時の「不要な側面を捨てる」という作業を**抽象化** (abstraction) と言います。つまり、具体的な計算を抽象化したものが手順、という言い方をしてもよいわけです。

とか 8cm とか「きっかり」の長さも世の中には存在しません。でも、そういう細かいことは捨てて「理想的な三角形」に抽象化してその面積を考えて計算しているわけです。

逆に言えば、コンピュータで計算する時には常に、現実世界のものをそのまま扱うわけではなくて、必要な部分だけをモデルとして取り出し、それを計算している、ということになります。この意味での抽象化やモデル化には、皆様はこれまで数学の一環として多く接してきたと思いますが、これからはコンピュータでプログラムを扱う時にもこのようなモデル化を多く扱っていきます。

1.1.2　アルゴリズムとその記述方法

前節における「三角形の面積の計算方法」のような、計算 (や情報の加工) の手順のことを**アルゴリズム** (algorithm) と言います。ある手順がアルゴリズムであるためには、次の条件を満たす必要があります。

- 有限の記述でできている。
- 手順の各段階に曖昧さがない。
- 手順を実行すると常に停止して求める答えを出す。[3]

1 番目は、「無限に長い」記述は書くこともコンピュータに読み込ませることも不可能だからです。2 番目は、曖昧さがあるとそれをコンピュータで実行させられないからです。3 番目はどうでしょうか。実際にコンピュータのプログラムを書いてみると、手順に問題があって実行が止まらなくなることも頻繁に経験しますが、そのようなものはアルゴリズムとは言えないのです。[4]

アルゴリズムを考えたり検討するためには、それを何らかの方法で記述する必要があります。その記述方法としてはさまざまなものがありますが、ここでは手順や枝分かれ等をステップに分けて日本語で記述する、**擬似コード** (pseudocode) と呼ばれる方法を使います。**コード** (code) とは「プログラムの断片」という意味で、「擬似」というのはプログラミング言語ではなく日本語を使うから、と考えておいてください。

三角形の面積計算のアルゴリズムを擬似コードで書いてみます: [5]

- triarea: 底辺 w、高さ h の三角形の面積を返す
- 　s ← $\frac{w \times h}{2}$。
- 　面積 s を返す

1.1.3　変数と代入/手続き型計算モデル

上のアルゴリズム中で次のところをもう少しよく考えてみましょう:

- s ← $\frac{w \times h}{2}$。

この「←」は**代入** (assignment) を表します。代入とは、右辺の**式** (expression) [6]で表された値を計算し、その結果を左辺に書かれている**変数** (variable — コンピュータ内部の記憶場所を表すもの) に「格納する」「しまう」ことを言います。つまり、「w と h を掛けて、2 で割って、結果を s

[3]実は、計算の理論の中に「答えを出すかどうか分からないが、出したときはその答えが正しい」という手順を扱う部分もありますが、ここでは扱いません。

[4]停止することを条件にしておかないと、アルゴリズムの正しさについて論じることが難しくなります。たとえば、「このプログラムは永遠に計算を続けるかもしれませんが、停止したときは億万長者になる方法を出力してくれます」と言われて、それを実行していつまでも止まらない (ように思える) とき、上の記述が正しいかどうか確かめようがありません。

[5]以下ではこのように、何を受け取って何を行う手順 (アルゴリズム) かを明示するようにします。上の例で「返す」というのは、底辺と高さを渡されて計算を開始し、求まった結果 (面積) を渡されたところに答えとして引き渡す、というふうに考えてください。

[6]プログラミングで言う式とは、計算のしかたを**数式** (mathematical expression) に似た形で記述したものを言います。先に説明した、電卓で計算する手順を記したようなものと思ってください。

のところに書き込む」という**動作** (action) を表していて、数式のような定性的な記述とは別物なのです。

数式であれば $s = \frac{w \times h}{2}$ ならば $h = \frac{2s}{w}$ のように変形できるわけですが、アルゴリズムの場合は式は「この順番で計算する」というだけの意味、代入は「結果をここに書き込む」というだけの意味ですから、そのような変形はできないので注意してください。困ったことに、多くのプログラミング言語では代入を表すのに文字「=」を使うので、普通の数式であるかのような混乱を招きやすいのです。

これをモデルという立場からとらえると、式は「コンピュータ内の演算回路による演算」を抽象化したもの、変数は「コンピュータ内部の**主記憶** (main storage) ないし**メモリ** (memory) 上のデータ格納場所」を抽象化したもの、そして代入は「格納場所へのデータの格納動作」を抽象化したもの、と考えることができます。

このような、式による演算とその結果の変数への代入によって計算が進んでいくようなモデルを**手続き型計算モデル** (procedural computational model) と呼び、そのようなモデルに基づくプログラミング言語を**命令型言語** (imperative language) ないし**手続き型言語** (procedural language) と呼びます。手続き型計算モデルは、上述のように現在のコンピュータとその動作をそのまま素直に抽象化したものになっています。このため手続き型計算モデルは、最も古くからある計算モデルでもあるのです。

コンピュータによる計算を表すモデルとしては他に、関数とその評価に土台を置く関数型モデルや、論理に土台を置く論理型モデルなどもあるのですが、上記のような理由から、手続き型モデルが今のところもっとも広く使われています。

1.2 アルゴリズムとプログラミング言語

1.2.1 プログラミング言語

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あり、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。

1.2.2 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) [7]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) [8]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:

[7]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。

[8]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。

```
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)[9]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) [10]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。
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。

1.2.3 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。

[9]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。
[10]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。

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: [11]

```
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。

[11]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。

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。[12]

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。
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。[13]
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。

1.2.4　数値の表示に関する補足

d や e をやる場合は、数値を表示する時に十分な桁数がないと細かい違いが分からないので、そのための出力命令の説明をしておきます。先の例のように `irb` を使って自動的に出力させる場合は、桁数などは「おまかせ」になりますが、これを自分で制御する時は次に示す出力命令のうち、「`printf`」を使う必要があります。

- `puts(値)` — 値を (文字列でなければ) 文字列に 変換し、出力する。
- `printf("書式文字列", 値)` — 「書式文字列」 を出力しますが、その中に「出力指定」が埋め込まれていたら、その箇所に後ろの値を**書式** (format) に従って文字列に変換した上で順次埋め込みます。とくに「`%.`N`g`」という出力指定は数値を有効数字 N 桁で表示する指定です。たとえば次のようにすると、`x` の値を有効数字 20 桁で出力し、最後に改行します:

簡単な例題として、渡された数の 3 分の 1 を 20 桁表示するプログラムを掲載します。

```
def onethird(x)
  printf("%.20g\n", x / 3.0)
end
```

これを動かした様子をお見せしましょう。

```
irb> onethird 3
1
=> nil
irb> onethird 1
0.33333333333333331483
=> nil
```

[12] 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。
[13] x の平方根 (square root) は **Math.sqrt(**x**)** で計算できます。

「3 を 3 で割ったら 1」は問題ないですが、「1 を 3 で割ったら」…正解は「0.33333…」と 3 が無限に続くわけですが、コンピュータは無限を扱えないので、ある決まった桁数 (精度) までの計算になります。その結果がこうなるわけです。なお、このメソッド onethird は「実行に伴う動作として」出力がなされ、返す値自体は「nil」(何も返さないことを表す値) になっています。前の例と見比べてみてください。

ところで、桁数を色々変えて見たいときに、毎回関数を作り直すのはちょっと面倒です。実は irb で直接 printf 命令など個々の命令を実行させることもできますので、その様子を示しておきます。

```
irb> printf "%.20g\n", 1.0/3.0
0.33333333333333331483
=> nil
irb> printf "%.30g\n", 1.0/3.0
0.333333333333333314829616256247
=> nil
```

表示の桁数だけ増やしても計算の精度は変わらないことが分かると思います (じゃあ、どれくらいの精度なのかな?)。なお、整数についてはこの限りではありません (こちらも、どう限りでないかを確認することも課題のうちだと思ってください)。

1.3 コンピュータ上での数値の表現

1.3.1 十進表現と二進表現

コンピュータが作られた当時の主要な目的は、人間に代わって文字通り「計算」を高速に/大量に/正確に行うことでした。このため、コンピュータでもっとも最初に扱われたデータの種類は**数値** (numerical value) でした。

数を表現する方法としては、**アラビア数字** (Arabic numerals — 0〜9 の数字) を用いた**位取り記法** (positional notation) が圧倒的に多く使われています。私達が使う**十進表現** (decimal representation) ないし**十進法** (decimal system) の位取り記法では、数字として 0〜9 までの 10 種類ですべての数を書き表し、その値は桁が 1 増えるごとに十倍になります。たとえば図 1.1 左の十進法で「342」というのは「1 が 2 個、十が 4 個、百が 3 個」という意味であり、下にゼロをつけるとそれが「十が 2 個、百が 4 個、千が 3 個」になるので全体として十倍になるわけです。

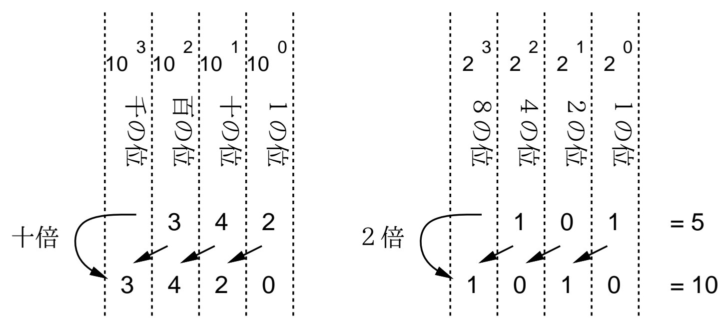


図 1.1: 十進法と二進法

一般に 4 桁の十進法ので表記した数 $abcd$ は次のように解釈できます。

$$a \times 10^3 + b \times 10^2 + c \times 10^1 + d \times 10^0$$

ところで、「10」という値は特別ではなく、別の数を用いることもできます。この、位取りの基準となる数を**基数** (radix) と呼びます。我々が基数として「10」を使っている (十進表現を使っている) のは、単なる偶然 (両手の指を合わせると 10 本あるから) なのです。

そして、コンピュータでは主に**二進表現** (binary representation) ないし**二進法** (binary system) が使われます。これは、コンピュータの実現に使う電子回路では「電流が流れている/いない」「電圧がある/ない」など 2 つの状態を持たせる回路が作りやすいためです。[14]

二進表現では、数値として「0、1」の 2 種類を用い、1 桁右に行くごとに 2 倍の数を表すことになります。たとえば図 1.1 右の「101」は「1 が 1 個、2 が 0 個、4 が 1 個」ですから合わせて 5 です。これの右に 0 をつけてずらすと 2 倍ですから 10 になるわけです。一般に 4 桁の十進法ので表記した数 $abcd$ は次のように解釈できます。

$$a \times 2^3 + b \times 2^2 + c \times 2^1 + d \times 2^0$$

2 進法をもっとイメージしやすくするには、図 1.2 のように「1」「2」「4」「8」「16」…個の○が描かれたカードが並んでいて、そのうちから 1 に対応するカードのみ裕って○の数を合計する、と考えるとよいかも知れません。

1 0 1 0 = 10_{10}

1 1 0 1 = 13_{10}

図 1.2: 1、2、4、8…個の○が描かれたカード

1.3.2 負の数の表現と二の補数

上で説明した二進表現では、N ビットの場合 $0 \sim 2^N - 1$ までの範囲の数が表せます。これを (負の数が含まれないという意味で) **符号なし二進表現** (unsigned binary representation) と呼ぶこともあります。

しかしコンピュータでの計算では、負の数も当然必要です。このため、1 ビットを**符号ビット** (sign bit) として用い、正負の数をともに扱うような表現方法が複数作られました。ここではその中から、現在のほとんどのコンピュータで採用されている**二の補数表現** (two's complement representation) について説明します。

二の補数表現とは、簡単に言えば「符号なし二進表現の上半分 (再上位ビットが 1) の範囲を、そのまま負の数の側に移したもの」と考えるとよいでしょう。たとえば、3 ビットの符号なし二進表現と二の補数の対応は図 1.3 のようになっています。つまり、3 ビットの符号なし二進表現では 0〜7 の範囲の値が表せるのに対し、二の補数では $-4 \sim 3$ の範囲の値が表せます。

二の補数表現の特徴として、符号なし二進表現の計算と同じ回路で (単に最上位からの桁上がりを無視するだけで) 負の数を含んだ計算がそのまま行える、という点があげられます。たとえば、「$-2+3=1$」は「$110+011=(1)001$」 となり、確かに最上位の桁上がりを無視する点以外は符号なし二進表現と同じ計算で行えています。

また、**符号反転** (negation — マイナス 1 を掛けること) の操作は、「各ビットの 0/1 を反転してから 1 を足す」操作で行えます。たとえば、3 は「011」なので、その 0/1 を反転して「100」、さ

[14] 二進表現/十進表現された数のことを**二進数** (binary numbers)/**十進数** (decimal numbers) と呼ぶ流儀もありますが、数そのものはどのように表記しても同じ数なはずなので、これは厳密に言えばおかしい言葉づかいです。また、数学では素数 p に対する「p 進数 (p-adic number)」という用語を全く別の意味で用いています。

値	二進	二の補数
7	111	
6	110	
5	101	
4	100	
3	011	011
2	010	010
1	001	001
0	000	000
-1		111
-2		110
-3		101
-4		100

図 1.3: 3 ビットの二の補数表現

らに 1 を足すと「101」となり、これは確かに -3 の二の補数になっています。逆も一応示しておくと、「101」→「010」→「011」で確かに元の 3 に戻ります。

符号なしの整数についても、2 の補数表現の整数についても、整数という本来は無限個あるもののなかから、与えられたビット数で表せる有限の範囲を「切り取ってきて」表現しているため、演算の結果が表せる範囲を超えてしまうと正しくない結果が得られることになります。具体的には「正の数と正の数を足したのに負の数になった」などのことが起こります。このような、扱える範囲を越える演算を行ったために結果が正しくなくなることを一般に**あふれ** (overflow) と呼びます。

また、2 の補数ではマイナスの数は 0 以上の数より 1 個多く表せるため、「符号を反転したのにまた元の数に戻ってしまう」数が存在することになります (この場合も符号反転時にあふれが起きます)。コンピュータで数値を扱う時は、このようなことを常に意識しておく必要があります。

さて、以上の説明は多くのプログラミング言語 (C、C++、Java など) にあてはまるのですが (これらの言語では主に 32 ビットの 2 の補数表現が使われています)、Ruby ではちょっと事情が違います。上のような限界はあくまでも「ビット数が決まっている」場合に起きることなのでした。これを克服するため、Ruby では整数値の演算結果がある標準のビット数以内で表せなくなった時には適宜ビット数を増やして表せる範囲を自動的に広げるようになっています。このため、Ruby では整数の限界に伴う問題にぶつかることがなくなりますが、その代わり「数が大きくなるにつれて計算に掛かる時間も多くなる」ことになるので、やはり「数学の数とは違う」という注意は必要です。

1.3.3 実数の表現と浮動小数点

ここまでは「正負の整数」を扱ってきましたが、数にはもちろん小数点付きの数もあります。数学の世界では**整数** (integral number) は**実数** (real number) の特別な場合として含まれるわけですが、コンピュータ上の数の表現の場合は整数と実数はまったく違った性質を持っていて、プログラムの上でもきっぱり区別されます。

たとえば、先の三角形の面積のプログラムで割る数を「`2.0`」としたのと「2」としたのでは挙動が違うのに気づいた人もいるかと思います。「10 を 3 で割る」例で確認してみましょう。

```
irb> printf "%.30g\n", 10/3.0
3.33333333333333348136306995002
=> nil
irb> printf "%.30g\n", 10/3      ←両方とも整数だと
```

```
3                                ←切捨ての割り算
=> nil
irb> printf "%.30g\n", 10.0/3    ←片方が実数なら
3.33333333333333348136306995002 ←実数の割り算
=> nil
```

これは「`10.0`」「`3.0`」が実数を表す定数、「`10`」「`3`」が整数を表す定数という違いがあるためです。数学であればそのような違いはなく、2は2というだけなのですが、コンピュータでは整数と実数で扱いが違ってきます。具体的には、割り算「`/`」は分母・分子とも整数の場合は「小学校で小数を習う前の割算」になるので、余りがあっても切り捨てて答えが出されます。一方、分母または分子の少なくとも1方が実数なら、「小数を習った後の割算」になります。だから、割る数を「`2`」にしても、底辺か高さのどちらかを小数点付きで入力すれば小数点付きの結果になります。

では具体的には、有限のビット数で実数を表すのにはどうしたらよいでしょうか? たとえば、10進数で8桁ぶんの整数を表す方法があるのなら、そのうちの下から4桁が小数点以下、その上が小数点以上、のように考えればそれで小数点付きの数が表せる、という考えもあります:

□□□□.□□□□

このような考え方を、小数点が決まった位置に固定されていることから**固定小数点** (fixed point) による実数表現と呼びます。しかし実際には、この方法はあまりうまくいきません。というのは、科学技術計算ではすぐに「30,000,000」だとか「0.0000001」のような数値が出てくるので、この方法ではすぐに限界になってしまうからです。

ではどうしたらよいでしょうか。そのヒントは、理科では上のような数値の表現ではなく、「3×10^8」とか「1×10^{-6}」のような記法が多く使われる、というところにあります。つまり、1つの数値を**指数** (exponent — 桁取り) と**仮数** (mantissa — 有効数字) に分けて扱うことで、広い範囲の数値を柔軟に扱うことができます。この方法は、指数によって小数点の位置を動かすものと考えて**浮動小数点** (floating point) と呼ばれます。

たとえば、同じ10進数8桁ぶんでも、6桁の有効数字と2桁の指数に分けた浮動小数点表現を扱うとすれば、表せる絶対値のもっとも大きい数は「$\pm 9.99999 \times 10^{99}$」、0でない絶対値のもっとも小さい数は「$0.00001 \times 10^{-99}$」ということになり、ずっと広い範囲の数が扱えることになるわけです。

注意! コンピュータでは「小さい字」が使えないので、伝統的に指数部分を「`e`±指数」で表します (e は exponent の e)。たとえば「3.0×10^{22}」であれば「`3.0e+22`」です。このような表示は「エラー」とかではないのでそのつもりで。

実際にはコンピュータでは2進法を使うため、上と同様のことを2進表現で行っています。多くのプログラミング言語の実数データ型では、符号1ビット、仮数部52ビット、指数部 (符号含む)11ビット、合計64ビットの浮動小数点表現が使われています。(このビットの割り当ては、**IEEE754** と呼ばれる標準に従ったものです。)

1.3.4　浮動小数点と誤差

浮動小数点を用いた実数表現には、整数の表現とはまた違った注意点があります。まず、有効数字は当然ながら有限なので、その範囲で表せない結果の細かい部分は**丸め** (rouding — 十進表現で言えば四捨五入) が行われて、**丸め誤差** (roundoff error) となります。言い替えれば、コンピュータによる実数計算は基本的に近似値による計算を行っているものと考えるべきなのです。

また、絶対値が大きく異なる2つの数を足したり引いたりすると、絶対値が小さいほうの数値の下の桁は (演算のための桁揃えの結果) 捨てられてしまうので、これも誤差の原因となります。これを**情報落ち** (loss of information) と言います。極端な例として、演算した結果が元の (絶対値が

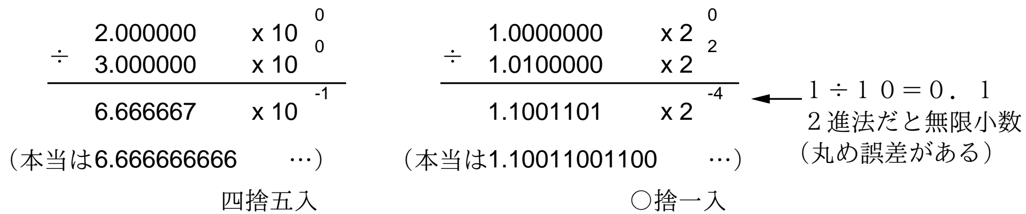


図 1.4: 丸め誤差

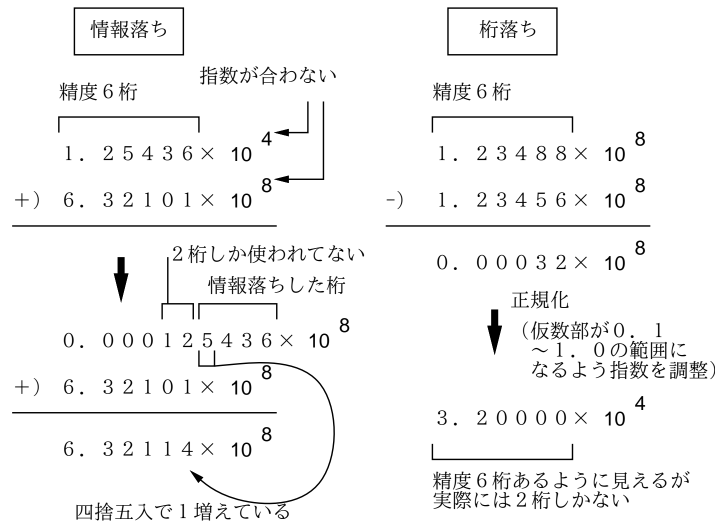


図 1.5: 情報落ちと桁落ち

大きいほうの) 数のまま、ということも起こります。これは、たとえば図 1.4 のような例を思い浮かべてみれば分かると思います。

逆に、非常に値が近い数値どうしを引き算する場合も、上のほうの桁がすべて 0 になるため、結果は元の数の下の部分だけから得られたものとなり、やはり誤差が大きくなります。これを**桁落ち** (cancellation) と言います。素朴に計算すると桁落ちが問題になる例として、次のものを考えてみます。

$$\sqrt{x+1}-1$$

x が 0 に近いとき、$\sqrt{x+1}$ も 1 に近いので桁落ちが起きます。これを避けるためには、次のように変形します。

$$\sqrt{x+1}-1=\frac{\sqrt{x+1}-1}{1}=\frac{(\sqrt{x+1}-1)(\sqrt{x+1}+1)}{\sqrt{x+1}+1}=\frac{x}{\sqrt{x+1}+1}$$

難しくしただけみたいに見えるかも知れませんが、最後の式であれば引き算をしないので桁落ちから逃れられるわけです。ところで、こう変形してみると、$x\sim 0$ のときにはこの式はおよそ $\frac{x}{2}$ だと分かりますね。実際に両方のやり方で計算してみて確認しましょう。

```
def calc1(x)
  return Math.sqrt(x + 1.0) - 1.0
end
def calc2(x)
  return x / (Math.sqrt(x + 1.0) + 1.0)
end
```

最初の素朴版から見てみます。

```
irb> calc1 0.00000000001
=> 5.000000413701855e-12
irb> calc1 0.000000000001
=> 5.000444502911705e-13
irb> calc1 0.0000000000001
=> 4.9960036108132044e-14
irb> calc1 0.00000000000001
=> 4.884981308350689e-15
irb> calc1 0.000000000000001
=> 4.440892098500626e-16
```

x が小さくなると、どんどん $\frac{x}{2}$ から外れて行きます。では修正版ではどうでしょうか。

```
irb> calc2 0.00000000001
=> 4.9999999999875e-12
irb> calc2 0.000000000001
=> 4.99999999999875e-13
irb> calc2 0.0000000000001
=> 4.999999999999876e-14
irb> calc2 0.00000000000001
=> 4.999999999999988e-15
irb> calc2 0.000000000000001
=> 4.999999999999999e-16
```

確かにこちらは大丈夫です。

最後にあと 1 つだけ、浮動小数点表現に関する注意があります。整数では全てのビットのパターンを数値の表現として使っていましたが、浮動小数点では指数部と仮数部の組み合わせ方に制約があるので (たとえば仮数部が 0 であれば値が 0 なので指数部には意味がなく、この時は指数部も 0 にしておくのが普通)、これを利用して正負の**無限大** (infinity — $\pm\infty$) や**非数** (NaN — Not a Number) などの特別な値を用意しています。また、0 にも「+0」と「−0」があったりします。だから、演算の結果としてこれらのヘンな値が表示されても驚かないようにしてください。

演習 1-4 整数の計算と実数の計算で「切捨て除算による違い以外に」結果が違うような計算の例を Ruby で試してみよ。どのような場合に違いが現れるかについて考察すること (この演習をやる場合は、実数を十分な桁数表示しようと思った場合に先に説明した `printf` を使う必要があると思います)。

> ヒント: 「123451234512345 + 1」は「123451234512346」ですね。では、「123451234512345.0 + 1.0」はどうでしょうか。また「12345」をもう 1 回ふやすとどうでしょうか。なぜでしょうか。

演習 1-5 実数型の浮動小数点の演算で誤差が現れるような計算の例を Ruby で試してみよ。どのような場合にどのような誤差が現れるかについて考察すること (この演習をやる場合は、先に説明した `printf` などを使わないと十分な桁数の表示が行われないのでうまく検討できません)。

> ヒント: 本文の例で丸め誤差、情報落ち、桁落ちが発生するような場面を考えてみてくえださい。

> ヒント: たとえば、「ある数に 0.1 を掛ける」場合と「ある数を 10.0 で割る (10 ではなく 10.0 にすること!)」場合では結果が違うような数があります。ところが、「0.125 を掛ける」のと「8.0 で割る (8 ではなく 8.0 にすること!)」の場合は違いはありません。なぜでしょうか。

1.4 演習問題解説 (一部)

演習 1-3a — 四則演算を試す

演習 3a はとりあえずは和の計算でした。メソッド内の計算式を取り替えればいいだけなので簡単です。

```
def add(x, y)
  return x + y
end
```

動かしているところも見てみましょう:

```
irb> add 3.5, 6.8
=> 10.3
```

四則つまり和、差、商、積の場合も上と同じにやればいいわけですが、和、差、商、積のために 4 つメソッドを作る代わりに 1 つで済ませるという方法を考えてみましょう (半分くらいは新しい内容の紹介を兼ねています)。まず先に説明したように、メソッドの最後に値を返す代わりに、puts などで順次画面に書き出す方法があります:

```
def shisoku0(x, y)
  puts(x+y)
  puts(x-y)
  puts(x*y)
  puts(x/y)
end
```

動かしているところは次のとおり:

```
irb> shisoku0 3.3, 4.7
8.0
-1.4
15.51
0.702127659574468
=> nil
```

なるほど 4 つの値が順次打ち出され、最後に shisoku0 の結果としては **nil**(何もないことを示す値) が返されています。

上の方法だと「1 つの結果が返る」のでないのがちょっと、という気がするかもしれません。そこで次に、1 つの文字列を返し、その中に 4 つの数値が埋め込まれている、というふうにしてみましょう。

Ruby では**文字列** (character string — 文字が並んだデータ) は「'...'」または「"..."」のようにシングルクォート (single quotation character) または**ダブルクォート** (double quotation character) で囲んで表しますが、ダブルクォートのほうは内部に色々なものを埋め込む機能がついています。[15] 具体的には、文字列の中に「#{...}」という形のものがあると、中カッコ内の式を**評価** (evaluation — 値を計算すること) して、結果をそこに埋め込んでくれます。

これを利用した「四則演算」のメソッドを示します。

```
def shisoku1(x, y)
  return "#{x+y} #{x-y} #{x*y} #{x/y}"
end
```

[15] ダブルクォートはその埋め込み機能のために**特殊文字** (special character) をさまざまに解釈しますから、そのようなことをせずに文字列をそのまま表示させたい場合はシングルクォートを使ってください。

実行しているところは次のとおり:

```
irb> shisoku1 3.3, 4.7
=> "8.0 -1.4 15.51 0.702127659574468"
```

確かに、「文字列」が打ち出されていて、その中に 4 つの数値が埋め込まれています。

もう 1 つ、Ruby など多くの言語では値の並んだものを**配列** (array) という機能で扱います。Ruby では `[...]` の中に値をカンマで区切って並べることで配列を直接書けるので、これを使って 4 つの数値をまとめて返すことができます。[16]

```
def shisoku2(x, y)
  return [x+y, x-y, x*y, x/y]
end
```

実行しているところは次のとおり:

```
irb> shisoku2 3.3, 4.7
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

ここでは文字列の場合とあまり変わらない感じがするかもしれませんが、配列では返された値の中から「0 番目」「1 番目」など番号を指定して特定の要素を取り出せるので、より便利に使えます。

演習 1-3b — 剰余演算

演習 3b は剰余演算「%」を試すというものでした。演算子を取り換えるだけなので、プログラムは簡単ですね。

```
def jouyo(x, y)
  return x % y
end
```

実行してみましょう:

```
irb> jouyo 8, 5
=> 3
irb> jouyo 20, 5
=> 0
irb> jouyo -8, 5
=> 2
irb> jouyo -21, 5
=> 4
```

ちゃんとマイナスの時も試していただけたでしょうか? ここで「マイナスだとどうだろう」とか思うようになって頂きたいわけです。

それで、マイナスの時も剰余は負にならず、つまり「5 間隔」というのがマイナスまでずっと続いている、というふうに考えるのでしょうね。では、割る数がマイナスだったらどうでしょうか?

```
irb> jouyo 8, -5
=> -2
irb> jouyo -8, -5
=> -3
```

こんどは 2 数ともマイナスの場合をまず考えて、それから等間隔で、というふうに考えるとよいかと思います。[17]

[16] さらに `return` の後だと囲んでいる `[ ]` を省略できますが、場所によっては省略できないので常に書くことを薦めます。

[17] 剰余演算の振舞いは、プログラミング言語によって多少の違いが見られる箇所です。

演習 1-3c — 円錐の体積

演習 3c は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます:

```
def cornvol(r, h)
  return (r**2*3.1416*h) / 3.0
end
```

ちなみに「**」はべき乗 (power) の演算子です。もちろん 2 乗は「r*r」と書いても構いません。

```
irb> cornvol 3.0, 4.0
=> 37.6992
```

ところで「円周率 (circle ratio) が 3.1416 というのは不正確だ」と思う人もいそうですね。しかし、コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行えないのであり、自分で必要と思う適当な桁数を決めてその範囲でやるしかないので、有効数字 5 桁くらいでと思うならこれでよいわけです。[18]

演習 1-4 — 整数と実数の違い

演習 4 は「切捨て除算以外の」整数と実数の違いを試すというものでした。ヒントに載せたものをやってみます。足し算を直接書いてもいいのですが、定義したメソッド add を呼んでいます。

```
irb> add(123451234512345, 1)
=> 123451234512346
irb> add(123451234512345.0, 1.0)
=> 1.23451234512346e+14
```

これはどちらも同じですね。実数は「おまかせ」だと指数記法になりますが、まあ値としては同じことです。では、12345 をもう 1 つ増やしてみます。

```
irb> add(12345123451234512345, 1)
=> 12345123451234512346
irb> add(12345123451234512345.0, 1.0)
=> 1.23451234512345e+19
```

整数は問題ないですが、実数はおまかせだと適当なところで丸められてしまっているので、違いが分かりません。そこで printf で桁数を増やして表示させてみます。

```
irb> printf("%.20g\n", add(12345123451234512345.0, 1.0))
12345123451234512896
=> nil
```

こうして見ると下 3 桁はまったく違うことが分かります。

で、理由ですが、皆様も色々試して頂いたと思いますが、実数の場合は仮数 (有効数字の部分) が 16 桁程度に固定されていますから、それより多い桁数の部分は無意味 (ゴミ) だ、ということです。

こうして見ると、コンピュータで計算するときには、整数による計算、実数による計算をきちんと計画的に使い分ける必要があることが分かります。だとすると、整数を実数に、また実数を整

---

[18] 3.141592653589793 くらいまでは扱える精度があるので、この定数をいちいち書くのは嫌だという人のために Math::PI と書いてもよいようになっています。同様に、自然対数の底 (base of natural logarithm)e は Math::E で表せます。

数に変換する必要がありそうです。Ruby ではそれには、数値が持つメソッド x.to_i(整数に) と x.to_f(浮動小数点に) を遣います。[19]前者は数値 x が整数ならそのままですが、実数なら (端数を切り捨てて) 整数に変換します。後者は x が実数ならそのままですが、整数なら実数に変換します。

```
irb> 3.14.to_i
=> 3
irb> 314.to_f
=> 314.0
```

整数と実数なんて、小数点をつけて書くかどうかで区別すればいいのでは、と思いましたか? 実際にはこれらは、変数に入れて計算している値を整数や実数に切り替えたいところで使います。

演習 1-5 — 実数計算の誤差

演習 4 は整数と実数の違いに関するものしたが、こちらは実数計算の誤差を観察する問題でした。当然、printf を使って十分な桁数を表示するようにします。いちいち関数を書かずに irb だけで直接計算してみましょう:

```
irb> printf "%.20g\n", 1.0/3.0
0.33333333333333331483  ←有効桁数は 10 進で 16 桁程度
=> nil
irb> printf "%.20g\n", 1.0/3.0 * 3.0
1           ←3 倍すると最後の桁が丸められて元に戻る
=> nil
irb> printf "%.20g\n", 7.0 / 10.0
0.69999999999999995559 ←0.7 も 2 進で切りよくない
=> nil
irb> printf "%.20g\n", 7.0 * 0.1
0.70000000000000006661 ←値 0.1 も誤差を含むので
=> nil
```

このように、有限桁数の計算なので微妙に誤差が現れます。しかし一方で、2 進表現を使っているということは、2^N とか $\frac{1}{2^N}$ とかは非常に「切りのよい数」となり、あふれない限りは誤差が出ません。

```
irb> printf "%.40g\n", 7.0 / 16.0
0.4375
=> nil
irb> printf "%.40g\n", 7.0 * 0.0625
0.4375
=> nil
irb> printf "%.40g\n", 7.0 / (2**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**16)
0.0001068115234375
=> nil
irb> printf "%.40g\n", 7.0 / (2**32)
```

[19] これらのメソッドは、これまで出て来たメソッドと書き方が違い、値 (この場合は x) に「対して」実行する、という形で値の後ろに「.」でつなげて書きます。このようなメソッドはこれから沢山出て来ます。

```
1.62981450557708740234375e-09
=> nil
irb> printf "%.40g\n", 7.0 * (0.5**32)
1.62981450557708740234375e-09
=> nil
```

ここで例示したのは丸め誤差 (2 進法で正確に表せないことによる誤差) 関係の事柄でしたが、情報落ちや桁落ちについて調べて頂いてももちろん結構です。情報落ち、桁落ちの具体例は前回資料でそれなりに説明したつもりです。

1.5 検討 プログラミングへの入門

コンピュータは我々が日常扱っている「もの」とは違う特性を多く持っていて、プログラミングではそのことが端的に現れます。ですから、「どのように違うか」をきちんと説明して納得してもらうことが、プログラミングを好きになってもらう早道だと考えます。具体的には、次のようなことがらです。

- プログラムは決まった規則で書かれていて、そこから少しでも外れると思ったように動かない。
- その理由は、プログラムはコンピュータが行うすべての動作を厳密に記述するので、記述が違えば起こることも対応して違うようになっているから。たとえば「`/ 2.0`」だと実数計算で 2 で割るが、「`/ 2`」だと (分母も整数なら) 切捨て割り算をする。
- 一方で、決まった規則の中でなら自由度がある。見やすく字下げをするとか、変数の名前を決めるとかはすべて書く人の工夫次第。
- コンピュータは電子回路によって出来ていて、その動作のしかたは厳密に決まっている。たとえば化学の実験をしたら得られる数値は毎回少しは違っているが、コンピュータで計算をしたら何回やっても「誤差まで含めて」「まったく同じ結果が」常に得られる。
- ではなぜ誤差があるかというと、電子回路によって「何桁で」計算するかは決まっていて、それを越えた精度は「もともと無い」から。なおかつ、2 進法の (実数なら) 浮動小数点で計算するので、私達が日常の感覚で考えることと違うことが色々おこる。

浮動小数点計算誤差としては、丸め誤差、情報落ち、桁落ち、打ち切り誤差 (後述) があるのですが、高校レベルではその区分までは求めなくてもよいように思います (大学の試験ではよく問われます)。

このセクションでは、プログラミングに入門するということで、まずは「数値を与えてそれに対して一定の計算をする」という非常に基本的なプログラムだけを扱います。このため、Ruby の記法として学ぶのは次のことだけです。

- `def ... end` でメソッドを定義する。メソッドの先頭で名前とパラメタを定める。
- 計算式と代入で数値の計算をする。
- return で結果を返す。

書き方に関する事柄はできるだけ少なくして、その上で実際に自分でプログラムを記述してもらい、上に挙げたような事柄を体験してもらうことが前提です。「自分で発見したことは、理解していて、覚えていられる」からです。

もちろん、実際に演習をすると細かいところで沢山つまづきます。そこはきちんとサポートして、まず動くことはクリアさせ、動いた結果について考えることに注力してもらいます。

open questions

- この程度の、文字の記述によるプログラミングは、どの学校でも実施できるでしょうか?
- ここに挙げた課題群は、自主的にやってもらうだけの興味を惹くことができるでしょうか?
- はじめて動かすプログラムとして、この程度の簡単さは適切でしょうか? (もっと手前の「計算式1個」から始めるという流儀もあります)。
- 世間一般では「動かさせるだけで手一杯」になってしまい、その先の考えることまで扱えていないのではないでしょうか? それに慣れていると、生徒が「動いたら満足 (考えない)」になる恐れは無いでしょうか? そのことへの対処方法は?

第2章　制御構造(1)

2.1　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。

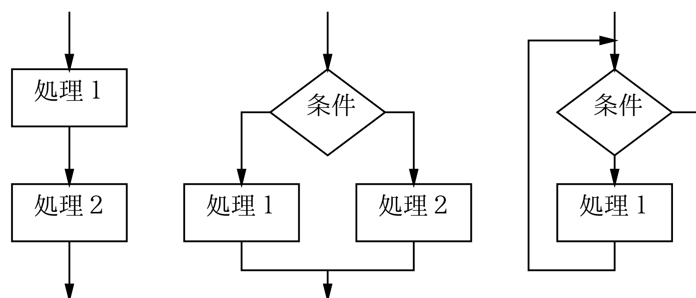


図 2.1: 3つの基本的な制御構造

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 2.1 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 2.1 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。[1]

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。

[1]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。

2.2 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              if 条件 then
  ... 動作1 ...              ... 動作1 ...
else                      end
  ... 動作2 ...
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。[2]
- 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。[3]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:

[2] 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。

[3] 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 2-1 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 2-2 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。

2.3 繰り返しと while 文

ここまででは、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: [4]

- ～ 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。

[4]「～」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。

2.4 数値積分

もっと有用な繰り返しの具体的な題材として、**数値積分** (numerical integration — 定積分の値を数値を計算する方法で求めること) を取り上げてみましょう。皆様はこれまで、定積分を求めるのに、積分の公式を覚えたり、公式のあてはめや変形に苦労したりされてきたと思います。しかしプログラムを使えば、元の関数から直接定積分の値を計算してしまえるのです。

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは図 2.2 のように、その関数のグラフを描いたとして、区間 $[a, b]$ の範囲における関数の下側の面積なのでしたね。そこで、図 2.2 にあるように、その部分に多数の細長い長方形を詰めてみて、その面積を合計すれば知りたい面積の値、つまり定積分の値が求まることになります。各長方形の幅は区間を n 等分した値 dx、高さは $f(x)$ の値なので、その面積を計算するのは簡単です。

この方法であれば、$f(x)$ が数式の形で不定積分が求められないような 関数であっても、問題なく定積分が計算できることになります。(数式の形で一般的に問題の解を求めることを**解析的**

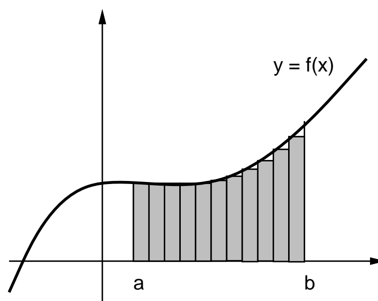


図 2.2: 数値積分の原理

(analytical) に解く、と言います。これに対して、数値で計算して特定の問題の答えを求めることを**数値的** (numerical) に解く、と言います。)

とはいえ、今は「正しい」値が求まるかどうかチェックしたいので、簡単な関数 $y = x^2$ でやってみます。不定積分は $\frac{1}{3}x^3$ ですから、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ ということになります。たとえば $[1, 10]$ だったら $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ となります。ではアルゴリズムを作ってみましょう:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- 　　$dx \leftarrow \frac{b-a}{n}$。
- 　　$s \leftarrow 0$。
- 　　$x \leftarrow a$。
- 　　$x < b$ が成り立つ間繰り返し:
- 　　　　$y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- 　　　　$s \leftarrow s + y \times dx$。
- 　　　　$x \leftarrow x + dx$。
- 　　繰り返し終わり。
- 　　s を返す。

すなわち、x にまず a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ってそれを x に入れ直すことで x を徐々に (dx きざみで) 動かしていき、b まで来たら繰り返しを終わります。このように、繰り返しでは「こういう条件で変数を動かしていき、こうなったら終わる」という考え方が必要なのです。面積のほうは、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えていくことで、合計を求めます。

では Ruby プログラムを示しましょう。「#」の右側に書かれている部分は**注記**ないし**コメント** (comment) と呼ばれ、Ruby ではこの書き方でプログラム中に覚え書きを入れておくことができます。コードの意味が分かりづらい (何のためにこのような計算をしているのか読み取りにくい) 箇所には、その意図を注記しておくようにしてください。また、一時的に命令を実行しないようにするのにも、コメントが便利に使えます。これを**コメントアウト** (comment out) と呼びます。この例でも後で使うコードをコメントアウトしてあります。

```
def integ1(a, b, n)
  dx = (b - a) / n
  s = 0.0
  x = a
# count = 0
  while x < b do
    y = x**2          # 関数 f(x) の計算
    s = s + y * dx
```

```
    x = x + dx
#   count = count + 1
#   puts("count=#{count} x=#{x}")
  end
  return s
end
```

やっていることは先の擬似コードそのままだと分かるはずです。さて、333 が求まるでしょうか? 実行させてみます:

```
irb> integ1 1.0, 10.0, 100
=> 337.557149999999          ←ふーん?
irb> integ1 1.0, 10.0, 1000
=> 332.554621500007          ←小さい
irb> integ1 1.0, 10.0, 10000
=> 333.045451214912          ←大きい…
```

なんだかヘンですね。そこで、繰り返しの回数がいくつになっているかをチェックすることにして、上の行頭の「#」を削って動かし直してみました: [5]

```
irb> integ1 1.0, 10.0, 100
...
count=98 x=9.81999999999999  ←誤差が…
count=99 x=9.90999999999999
count=100 x=9.99999999999999
count=101 x=10.09            ← 101 回目が…
=> 337.557149999999          ←このため多い
```

区間数 100 個なのに長方形を 1 個余計に加えたため、値が大きすぎるわけです。なぜこんなことが起きるのでしょう? それは「$x \leftarrow x + dx$ で x を増やしていき b になったらやめる」というアルゴリズムの問題なのです。そもそもコンピュータでの浮動小数点計算は近似値の計算なので、dx を区間長の $\frac{1}{100}$ にしたとしても、そこに誤差があります。このため 100 回足してもわずかに b より小さい場合があり、その時は余分に繰り返しを実行してしまいます。

2.5 計数ループ

ではどうすればよいでしょうか。繰り返し回数を 100 回と決めているのですから、回数を数えるのは整数型で行い、[6]それをもとに各回の x を計算するのがよいのです。つまり、次のようなループを書くことになります (**カウンタ** (counter) とは「数を数える」ために使う変数のことを言います):

```
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
```

[5]つまり、変数 count に回数を数えつつ x を表示するようにするわけです。このように、コメントアウトしてあったコードを活かして動かすことを「コメントアウトを外す」とも言います。
[6]整数ならば、あふれない限り誤差はありません。

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。[7]

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。[8]

- 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。

2.6 数値積分 (つづき)

余談が終わったので、先の積分プログラムを計数ループを使うように直してみましょう:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算

---

[7] 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。

[8] 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。

- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- 変数 i を 0 から n の手前まで変えながら繰り返し、
- 　$x \leftarrow a + i \times dx$。
- 　$y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- 　$s \leftarrow s + y \times dx$。
- 繰り返し終わり。
- s を返す。

先のプログラムと違うのは、毎回 x を i から計算しているところです。では、これを Ruby にしたものを示します:

```
def integ2(a, b, n)
  dx = (b - a) / n
  s = 0.0
  n.times do |i|
    x = a + i * dx
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
  end
  return s
end
```

これを動かしてみましょう:

```
irb> integ2 1.0, 10.0, 100
=> 328.55715
irb> integ2 1.0, 10.0, 1000
=> 332.5546215
irb> integ2 1.0, 10.0, 10000
=> 332.955451215
```

こんどはきざみを小さくすると順当に誤差が減少していきます。しかし、常に正しい面積である 333 より小さいようですが、これはなぜでしょうか?

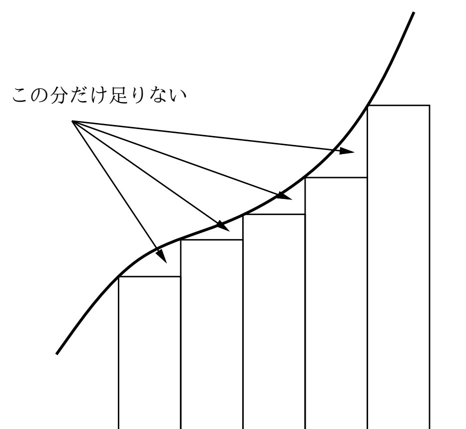


図 2.3: 区間の左端を使う場合の誤差

それは、長方形の面積を計算するのに微小区間の左端の x を使って高さを求めているため、値が増大していく関数では図 2.3 のように微小な三角形の分だけ面積が小さめに計算されてしまうか

らです (逆に減少関数だと大きめに計算されます)。これをもうちょっと何とかする方法については、演習問題にしたので、やってみてください。

演習 2-3 上の演習問題のプログラムを打ち込んで動かせ。動いたら「減少する関数だと値が大き目に出る」ことも確認せよ。できれば、左端ではなく右端で計算するのもやってみるとよい。その後、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

a. 左端の x だけでも右端の x だけでも弱点があるので、両方で計算して平均を取る。
b. 左端や右端だからよくないので、区間の中央の x を使う。
c. 上記 a と b をうまく組み合わせてみる。

演習 2-4 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。

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

d. x と計算する項の数 n を与えて、次のテイラー展開を計算する。

$$
sinx = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots
$$

$$
cosx = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots
$$

実際に値の分かる x を入れて精度を確認してみること。$\pm 10\pi$ とかだとどうか? n はいくつくらいが適切か?

2.7 演習問題解説 (一部)

2.7.1 演習 2-2a — 枝分かれの復習

演習 2a 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。

2.7.2 演習 2-2b — 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。

2.7.3 演習 2-2c — 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。

2.7.4 演習 2-3a〜c — 数値積分

長方形の高さとして区間の左端の $f(x)$ を使うと増大関数で値が小さくなり、右端の $f(x)$ を使うと大きくなるので、「左端と右端の平均を取って」みるという課題でした (減少関数だと逆に大きく/小さくなります)。これは考えてみると、面積を計算するのにその区間の関数を直線で補間した「台形」を考え、その面積を求めているのと同等です。このため、これを数値積分の**台形公式** (trapezoid rule) と呼びます。

台形公式の計算内容は次のようになります (区間の幅を d で表す):

$$s = \sum \frac{1}{2}\{f(x) + f(x+d)\}d$$

これを計算する Ruby プログラムを示しておきます:

```
def integ3(a, b, n)
  dx = (b - a) / n
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n
    y0 = x**2
    y1 = (x+dx)**2
    s = s + 0.5*(y0+y1) * dx
  end
  return s
end
```

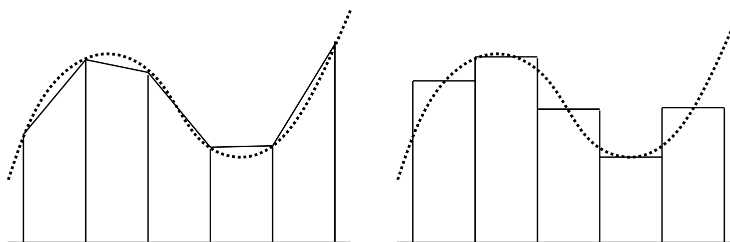


図 2.4: 台形公式と中点公式

台形公式は直線による補間なので、曲線が上に凸だと値は小さく、下に凸だと値は大きくなります。一方、これも演習にありましたが、区間の中央の x を使って長方形で計算すると (これを**中点公式**と言います)、逆に上に凸だと大きく、下に凸だと小さくなります (図 2.4)。だからこれをちょうどよく混ぜたらよいのでは、というのが演習 3c になっていたわけです。実は、左端:中央:右端を 1:4:1 で混ぜると (つまり台形:中点を 1:2 で混ぜると) よい結果が得られます。プログラムも示しておきます:

```
def integ4(a, b, n)
  dx = (b - a) / n
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n
    y0 = x**2
    y1 = (x+0.5*dx)**2
    y2 = (x+dx)**2
    s = s + (y0+4*y1+y2) * dx / 6.0
  end
  return s
end
```

実際に計算させてみましょう (「正解」は 333 だったことに注意):

```
irb> integ4 1.0, 10.0, 100
=> 333.0              ←ぴったり? 本当?
irb> printf "%.20g\n", integ4(1.0, 10.0, 100)
332.99999999999994316 ←確かにすごくよい
=> nil
irb> printf "%.20g\n", integ4(1.0, 10.0, 10)
333        ←分割数を減らしたら逆にぴったり?
=> nil
```

この計算式は**シンプソンの公式** (Sympson rule) と言われ、数値積分では標準的な方法です: [9]

$$s = \sum \frac{1}{3}\{f(x) + 4f(x+d) + f(x+2d)\}d$$

なぜこれがよいかというと、当該区間を 2 次曲線で補間することになるからです。だから積分しようとしている関数が 2 次以下の多項式だと「ぴったし」になり、そのため上の例では区間数が少ないほど (誤差が出ないため) よかったわけです。実際、分割数 1 でもぴったりなので、もはや数値積分と言えないような…

[9] この式では見やすくするため区間の半分を d としていて、そのためプログラムの 6 で割る代わりに 3 で割っています

2 次式の補間になる理由を示しておきます。当該区間の曲線を 2 次式

$$y = ax^2 + bx + c$$

で表せるものとします。また区間の幅を $2d$、左端を x_0、中央を $x_1 = x_0 + d$、右端を $x_0 + 2d$、対応する関数値を y_0、y_1、y_2 とおきます。上の 2 次式の不定積分は $\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ ですから、面積 (定積分) は次のようになります:

$$s = \left[\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx\right]_{x_0}^{x_0+2d}$$

これを整理すると次のようになります:

$$3s = \{a(6{x_0}^2 + 12x_0d + 8d^2) + b(6x_0 + 6d) + 6c\}d$$

ところで

$$y_0 = a{x_0}^2 + bx_0 + c$$
$$y_1 = a(x_0 + d)^2 + b(x_0 + d) + c$$
$$y_2 = a(x_0 + 2d)^2 + b(x_0 + 2d) + c$$

なので、見比べると次の式が成り立つと分かります:

$$3s = (y_0 + 4y_1 + y_2)d$$

というわけで、上の式が出て来るわけです。

数値積分にはシンプソンの公式が一番よいのかというと、必ずしもそうとは言えません。たとえば、ある細かさで積分を計算して、もっと細かくするために d を半分にしたいと思ったとすると、台形公式では既に計算した値をとっておいて、新たに加えた半分ずつの点についての計算を追加すれば済みます。このような計算方法を**漸近的**と言います。漸近的に計算していき、値の変化がなくなったらこれ以上細かさを増やしても意味がないと判断してやめるというのは 1 つの方法です。

2.7.5　演習 2-4a〜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
```

2.7.6 演習 2-4d — テイラー級数で sin と cos を計算

これは「階乗や x^n を計算しつつ」足していくのでちょっと面倒ですね。しかも交互に+/-が変わることも扱う必要があります。

```
def sincos(x, n)
  sign = 1; pow = 1.0; fact = 1; sin = 0.0; cos = 0.0
  n.times do |i|
    cos = cos + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+1)
    sin = sin + sign * pow / fact
    pow = pow * x
    fact = fact * (2*i+2)
    sign = -sign
  end
  return [sin, cos]
end
```

このプログラムでは、べき乗の数を 2 ずつ増やしながら sin と cos のテイラー展開を並行して計算しています。計算式を再録しておきましょう:

$$sinx = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$cosx = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!}x^4 - \frac{x^6}{6!} + \cdots$$

「何項まで計算するか」によって精度が変わって来ますが、言い換えれば実用のプログラムでは項を無限に計算することはできず、どこかで打ち切る必要があります。ということは、打ち切ったその先の項の値のぶんは無視されて誤差となわけです。これを**打ち切り誤差** (cutoff error) といい、既に学んだ丸め誤差、情報落ち、桁落ちと並んで数値計算における誤差の要因の 1 つです。

では実際に計算してみます:

```
irb> Math::PI / 3
  => 1.0471975511966  ← π/3 (60度)はこの値)
irb> sincos 1.0471975511966, 5   ←π/3のsin,cos
  => [0.866025445099782, 0.500000433432913]←微妙
irb> sincos 1.0471975511966, 10  ←項を増やすと
  => [0.86602540378444, 0.499999999999998]←まあOK
irb> sincos 3.141592653589, 10   ←πだと
=> [-5.2812499185062e-10, -1.00000000352908]←微妙
irb> sincos 31.41592653589, 10   ←10π
=> [-167876715320.415, -104528895953.392] ←破綻
```

何が問題なのでしょう? それは、x が大きくなるほどテイラー級数の収束が遅くなるためです。これに対処するため、sin とか cos が周期関数であることを利用し、この方法で計算するのは絶対値の小さい $0 \leq x \leq \frac{\pi}{4}$ の範囲だけにすべきでしょう (この範囲の sin と cos があれば残りの範囲は全部これらをもとに計算できますから)。

そして、x の範囲をこのように限定するなら、テイラー級数の項の数は 8 つくらいあれば十分と分かります (その先の項は分子の絶対値が 1 より小さく、分母は 10^{10} 以上になるので、そこで打ち切っても精度は十分です)。その場合、加えていく順序をテイラー級数の後ろの項から順にしたほうがよいのです。と言うのは、後ろのほうほど絶対値が小さくなるので、前から順に足すと情報落ちしやすくなります。項の数を決めておけば、後ろから足すように書くのも簡単です。

2.8 検討 制御構造・プログラム記述の多様性

この段階でようやく、アルゴリズムや計算的思考の重要な要素である「枝分かれ」「繰り返し」などの制御構造が登場します。ここでは急がずに十分な時間を掛けて、単一の枝分かれや繰り返しが使いこなせることをめざします (課題の中で組み合わせも多少現れますが)。

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) に進むようにしたかったため、やや題材的に窮屈になっています。ここで扱っている数値積分のテーマは、大学の理系 1 年生には無問題ですが、文系や高校生にはつらいところがあるかも知れません。ここではこの題材が「実数計算の誤差問題」に対する復習になるために結構詳しく扱っていますが、もっと簡単な題材で済ませてもいいかも知れません。

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3章　制御構造(2)

3.1　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, ...」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 3-1 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 3-2 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 3-3 「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 3-4 「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。)

3.2 さまざまなデータ型

3.2.1 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。

3.2.2 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 3.1)。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。

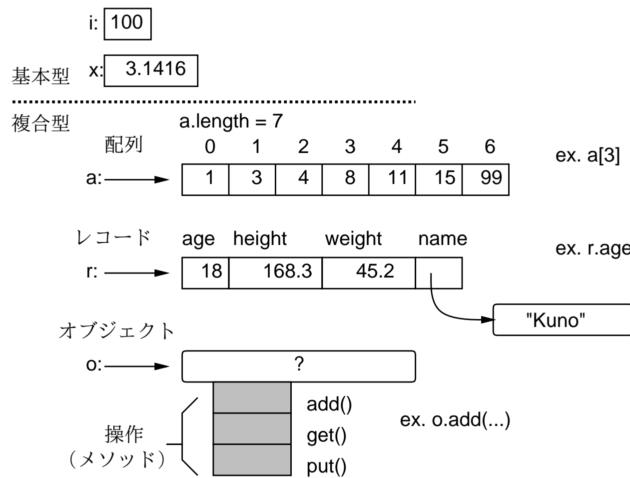


図 3.1: さまざまなデータ型

- **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 3.1 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。

3.2.3 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 3-5 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。

(a)

10	9	8	7	6	5	4	3	2	1

(b)

0	1	0	1	0	1	0	1	0	1

(c)

4	3	2	1	0	1	2	3	4	5

(d)

1	1	1	1	1	0	0	0	0	0

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。

3.2.4　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 3-6 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 3-7 「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。

3.3 演習問題解説 (一部)

3.3.1 演習 3-1 — 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 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
```

3.3.2 演習 3-2 — 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?

3.3.3 演習 3-3 — 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。

3.3.4 演習 3-4 — 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 3-5 — 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 3-6 — 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 3-7 — 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〜$N-1$ 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。

3.4 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?
- 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?

第4章　手続き

4.1　手続き/関数と抽象化

4.1.1　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。

4.1.2 手続き/関数と副作用

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。

4.1.3 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。

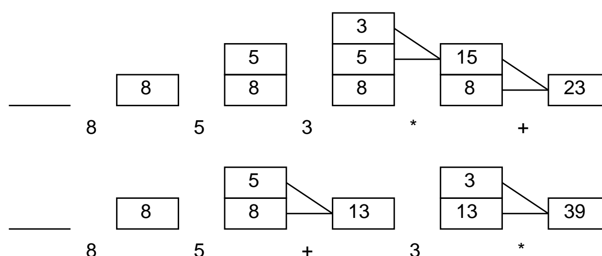


図 4.1: RPN 電卓による計算

RPN を使った計算は、図 4.1 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 4-1 「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 4-2 「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 4-3 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。

4.2 再帰呼び出し

4.2.1 再帰手続き/関数の考え方

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 4.2 を見れば分かります。

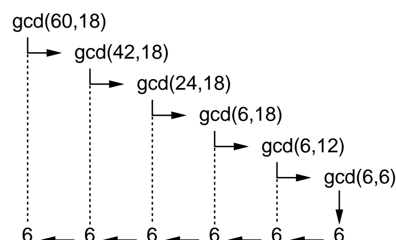


図 4.2: 再帰関数による最大公約数の計算

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 4-4 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4.2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]

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

4.2.2 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4.3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。

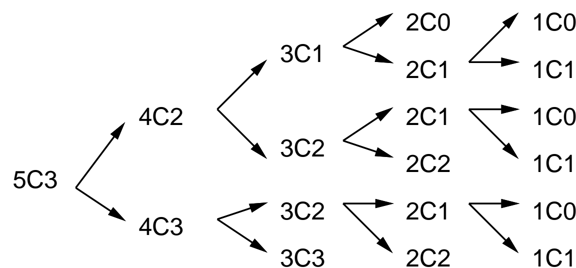


図 4.3: 再帰関数による組合せの数の計算

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
```

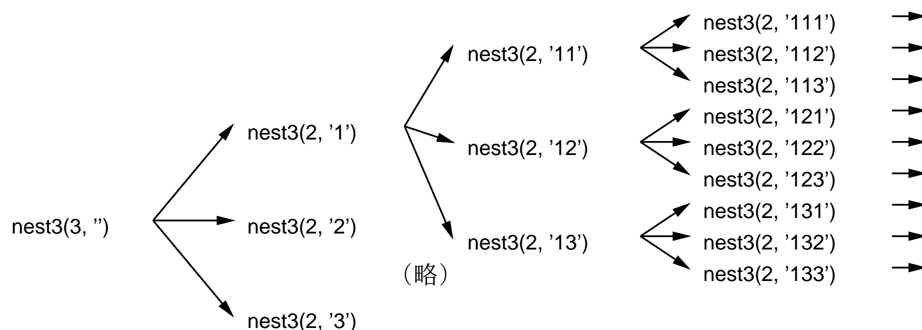


図 4.4: 1〜3 が N 個並んだすべての場合を出力

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 4.4 に示します。

4.2.3 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4.5に示します。

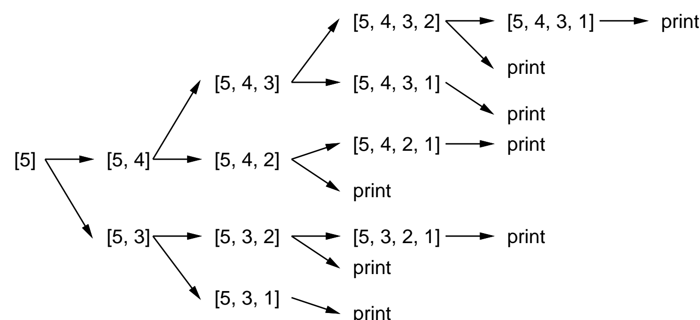


図4.5: 再帰関数による1〜2減少列の生成

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)。

4.2.4 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 4-5 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 4-6 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 4-7 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 4-8 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。

4.3 演習問題解説: 関数

演習 4-1: 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。

演習 4-2: 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。

演習 4-3: 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。

4.4　演習問題解説: 再帰、枝分かれ

演習 4-4: 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"
```

演習 4-5: 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"]
```

4.5 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5章　画像を生成する

5.1　2次元配列と画像の表現

5.1.1　2次元配列の生成

2次元配列 (配列の配列) を用いた前回の演習問題では、直接すべての値を「[[a, b], [c, d]]」のように指定することで2次元配列を生成していました。しかしこの方法は大きさが大きくなるととても大変です。1次元 (1列) の配列を作る時に、ブロックを使って初期値を設定する方法がありました。

```
a = Array.new(100) do |i| 2*i end
```

これを応用することで大きな2次元配列を作ることができます。つまり、ブロックの中にさらにArray.new(...) を入れれば、「配列が並んだ配列」つまり2次元配列ができるからです。

```
a = Array.new(10) do Array.new(10, 1) end
  # 10 × 10 ですべて「1」の行列
a = Array.new(10) do |i| Array.new(10) do |j| i*j end end
  # 「九九の表」
```

「2次元配列」と言いましたが、実際には図5.1のように配列のそれぞれの要素が配列、という構造になっています。でも普段はもっと簡単に「縦横2次元に要素が並んでいる」というイメージで問題ないでしょう。

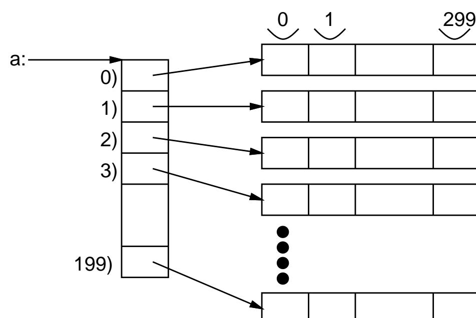


図 5.1: 2次元配列

演習 5-1 5×5の2次元配列で、次の図の(a)〜(d)のような内容のものをブロックつきのArray.new() を使って生成してみなさい。

(a)

0	1	2	3	4
-1	0	1	2	3
-2	-1	0	1	2
-3	-2	-1	0	1
-4	-3	-2	-1	0

(b)

1	0	0	0	0
1	1	1	1	1
1	2	4	8	16
1	3	9	27	81
1	4	16	64	256

(c)

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

(d)

1	0	0	0	1
0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1

なお、2次元配列を irb で表示するときは「pp」というメソッドを使うと見やすくできます。ただしこれを使うには「`require 'pp'`」というおまじないを実行させておく必要があります。

```
irb> require 'pp'    ← pp を使うための準備
  => true
irb> a = Array.new(5) do |i| Array.new(5) do |j| i*j end end
  => [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8],
  [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]    ← 見やすくない
irb> pp a            ← pp を使うと
[[0, 0, 0, 0, 0],    ←そろえてくれる
 [0, 1, 2, 3, 4],
 [0, 2, 4, 6, 8],
 [0, 3, 6, 9, 12],
 [0, 4, 8, 12, 16]]
  => nil
```

5.1.2 レコード型の利用

前章で説明したように、配列が「同じ型 (種類) の値が並んだもので、添字 (番号) により要素を指定する」のに対し、**レコード**は「違う型 (種類) の値でもよい、複数の値が組み合わさったもので、どの値 (フィールド) かは名前で指定する」ものでした。Ruby ではレコード型は `Struct.new` を使ってまずレコードクラスを定義し、その後そのレコードクラスを使って個々のレコード (データ) を作ります。具体的には、レコードクラスの定義は次のようにします: [1]

```
レコード名 = Struct.new(:名前, :名前, ...)
```

ここで「:名前」は先に説明した **記号** (symbol) リテラルで、これによりフィールドの名前が指定できます。個々のレコードを作るのは次によります:

```
p = レコード名.new(値, 値, ...)
```

これによりレコード型の値が作られ、指定した値が各フィールドの初期値になります (順番はレコード定義の時に指定した順になります)。上の例ではそのレコードを変数 p に入れています。

たとえば、コンピュータ上で画像を扱う時は、多数の**ピクセル** (pixel — 画素とも呼び、画面上の小さな点に対応) として扱うこと、そして各ピクセルの色は赤 (R)、緑 (G)、青 (B) の強さを 0〜255 の範囲の整数で表す方法が多く使われることはご存じと思います。このピクセルの情報を表すレコードを定義してみましょう:

```
Pixel = Struct.new(:r, :g, :b)
```

実際にこのレコードを使う時は、次のようになります:

```
p = Pixel.new(255, 255, 255)  # RGB とも 255 の値
```

Ruby では配列の時と同様、レコードも new を使って作り出さないと使えないのに注意してください。一旦作り出したあとは、「`p.r`」「`p.g`」「`p.b`」のように変数名の後にレコードのフィールド名をつけ加えたものが通常の変数と同様に使えます。配列と似ていますが、レコードの場合はフィールドは「名前」なので、プログラムを書いた時に決まってしまい、実行時には固定という点が違います。

[1] レコード名は大文字で始まらなければなりません。

5.1.3 ピクセルの2次元配列による画像の表現

さて、ピクセル1個の説明が終わったので、今度はこれを「2次元に (縦横に) 並べて」画像を作ることを考えます (図5.2左)。これをRubyで表現する場合、各Pixelを上で説明したようにRubyのレコード型で表現し、それを縦横に並べるわけです。たとえば縦方向 (高さ) が200ピクセル、横方向 (幅) が300ピクセルの画像を作るとします。そのためには、先に学んだ2次元配列の初期化のとき、個々の要素をピクセルにすればよいのです。

```
$img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end end
```

これによって作られるデータ構造は図5.2右のようになります。

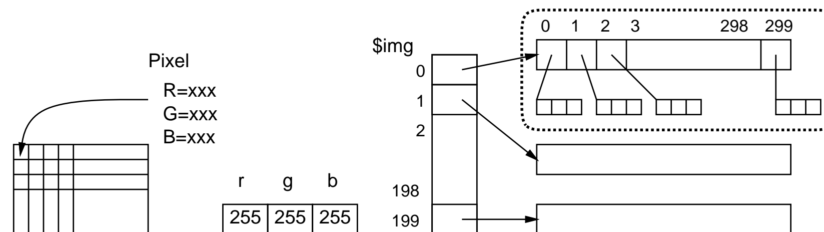


図 5.2: 画像のデータ構造とレコードの2次元配列

5.1.4 画像中の点の設定と書き出し

画像が無事生成できましたが、このままではRGB値とも「255」(最も明るい値) なので「真っ白」な状態です。そこで次に、指定した座標 (x, y) の点 (ピクセル) を指定したRGB値に書き換えるメソッドを作ります。

```
def pset(x, y, r, g, b)
  if 0 < x && x < 300 && 0 < y && y < 200
    $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
  end
end
```

if文は何のためにあるかというと、X座標とY座標が画像の範囲内 (ここでは0〜299、0〜199) に入っているときだけ書き込むようにするためです。こうしておくと、呼び出す側で間違って (または簡単のため) 画像のふちを越えた場所に書き込む呼び出しをしても単に無視できます。

さて次に、こうして画像中に書き込むことができるようになりましたが、画像を実際に「見る」ためには何らかのファイル形式で書き出す必要があります。ここではできるだけ簡単な形式として **PPM** 形式を選び、その形式でファイルに書き出すメソッド `writeimage` を作りました。[2]

```
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a| a.each do |p| f.write(p.to_a.pack('ccc')) end end
  end
end
```

メソッドの説明は次の通りです。

[2]普段私たちがWebなどで見ている画像形式はGIF、JPEG、PNGなどで、ブラウザもこれらの画像を表示するようにできていますが、これらのファイル形式は圧縮などの機能が備わっているため、そんなに簡単なコードで書き出すことができないのです。

- writeimage は画像のファイル名 (文字列) を指定して呼び出す。
- open は指定した名前のファイルをバイナリ (binary) 形式で書き出す (write) 準備をして、その出力チャネル (データの通り道) をブロックに渡して呼び出す。
- ここではブロックでチャネルを f という名前で受け取る。
- まず、ファイルに「P6 300 200 255」と出力する。これは「PPM 画像のカラー形式で、幅 300 ×高さ 200、RGB 値の最大は 255」を表す指定になっている。[3]
- 続いて、画像の2次元配列の各行について、さらにその行の中の各ピクセルについて、(1) ピクセルを配列に変換し (p.to_a)、その配列を3バイトのバイナリデータに変換し (.pack('ccc'))、ファイルに書き込む (f.write(…))。

5.1.5 例題: 画像を生成し書き出す

ではいよいよ、画像を作って書き出すメインのメソッドまで含めた全体像を見て頂きましょう。

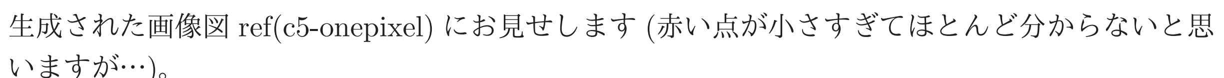
```
Pixel = Struct.new(:r, :g, :b)
$img = Array.new(200) do Array.new(300) do Pixel.new(255,255,255) end end
def pset(x, y, r, g, b)
  if 0 < x && x < 300 && 0 < y && y < 200
    $img[y][x].r = r; $img[y][x].g = g; $img[y][x].b = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6\n300 200\n255")
    $img.each do |a| a.each do |p| f.write(p.to_a.pack('ccc')) end end
  end
end
def mypicture
  pset(100, 80, 255, 0, 0)
  writeimage('t.ppm')
end
```

mypicture がメインになりますが、ここでは (100, 80) の位置に真っ赤な点 (RGB のうち R だけが最大なので) を打ち、t.ppm というファイルに書き出します。実際にこれを動かすには、ターミナルの窓を「2つ」開いて次のようにします。

- 片方の窓ではこれまで通り irb を動かし、mypicture を実行させる。
- もう片方の窓では、できあがった t.ppm を普通に表示できる形式に変換する。

```
% convert t.ppm Desktop/t.png     ←PNG 形式に
```

- そうすると、デスクトップに t.png が現れるので、これを開くと画像が見える。



生成された画像図 ref(c5-onepixel) にお見せします (赤い点が小さすぎてほとんど分からないと思いますが…)。

演習 5-2 上の例題を打ち込み、そのまま動かしなさい (色の RGB 値は 0〜255 の範囲で適宜変えてみるとよいでしょう)。動いたら、次のように変更してみなさい。

[3]画像ファイルの先頭にはだいたい、このような形で画像の種別やサイズを記述したデータが置かれています。この部分のことをヘッダ (header) などと呼びます。

図 5.3: 赤い点が 1 個

a. 水平または垂直または斜め (右上がり) に線を引くようにしてみる。
b. 長方形または円形を描いてみる (輪郭だけ描くのでも内側を指定した色で塗りつぶすのでもよい)。
c. 三角形を描いてみる (塗りつぶすのは多少工夫が必要かと)。
d. その他、好きな図形や模様や色を表現してみる。

mypicture の中にコードを追加して pset を呼び出すことを想定していますが、場合によってはさらにメソッドを追加する方が作りやすいかも知れません。

なお、上のコードの pset は「Y 座標が大きいほど下」に点を打ちます。コンピュータ上の画像は伝統的にこうなっていますが、皆様は「Y 軸が上向き」に慣れているので、pset を直して使う方がいいかも知れません。

5.2 画像に関するさまざまな補足

5.2.1 計算により図形を塗りつぶす

先の練習問題はどうでしたか。グラフのように「x を変えながら y=f(x) を計算して pset(x, y, ...)」と考える人が多いと思いますが、実はこの方法で輪郭線を描くと細かったり途切れたりしてあんまりよくありません。[4]むしろ図 5.4 のように「塗りつぶす」方がきれいにできやすいです。

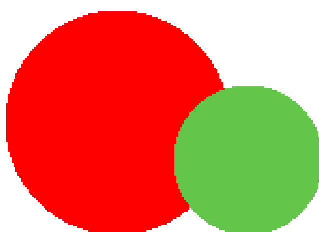


図 5.4: 2 つの内部まで色を塗った円

このような塗りつぶしをおこなうメソッド fillcircle を見てみましょう。

[4]途切れないように工夫したとしても、「1 ピクセル幅」というのは今日の画面解像度では「極めて細い」線になりますから。

```
def fillcircle(x0, y0, rad, r, g, b)
  200.times do |y|
    300.times do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end
```

このメソッド内では、times の中にさらに times、つまりループの中にさらにループがあるので、このようなものを **2 重ループ**と呼びます。この内側での 2 つの変数の進み方は、次のようになります:

```
0,0  0,1  0,2,  0,3  0,4  ....  0,288 0,299
1,0  1,1  1,2,  1,3  1,4  ....  1,288 1,299
2,0  2,1  2,2,  2,3  2,4  ....  2,288 2,299
...
...
198,0  198,1  198,2,  198,3  198,4  ....  198,288 198,299
199,0  199,1  199,2,  199,3  199,4  ....  199,288 199,299
```

縦横に揃えて書いてありますが、要は外側ループで y の値を 0〜199 まで変化させ、**そのそれぞれの値について**内側の x の値を 0〜299 まで変化させます。そして、これで画像上のすべての点 (座標) を洩れなく列挙しているわけです。つまり、ここで列挙される (x, y) の集合は次のようになるわけです (これが画像の全範囲)。

$$\{\ (x, y) \mid 0 \le x < 300,\ 0 \le y < 200\ \}$$

ところで、円というのは中心 (x_0, y_0) からの距離が rad 以内の点の集合ですから、次のように表せます。

$$\{\ (x, y) \mid |(x, y) - (x_0, y_0)| \le rad\ \}$$

これをプログラムで扱いやすいように、距離の 2 乗を使う形に直します。

$$\{\ (x, y) \mid (x - x_0)^2 + (y - y_0)^2\ \le\ rad^2\ \}$$

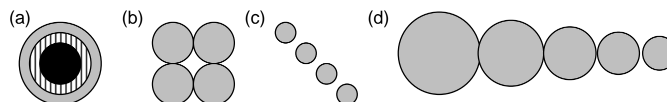
さて、先のコードでは 2 重ループの内側に if 文がありますが、その条件がまさにこの条件であり、従って「円に含まれるすべての点 (x, y) に対して色を設定する (塗る)」ことになるわけです。

実際にはこれを呼び出す必要があるので、円を 2 つ描き、ファイルに画像を書き出すメソッドを `mypicture1` という名前で用意しました (その結果が図 5.4 なのでした)。

```
def mypicture1
  fillcircle(110, 100, 60, 255, 0, 0)
  fillcircle(180, 120, 40, 100, 200, 80)
  writeimage("t.ppm")
end
```

長方形などさまざまな図形についても、このように「すべての点のうち、図形内部に含まれるという条件を満たす点のみに色を設定する」という形でコードを作ることができます

演習 5-3 円を塗るメソッドを先のプログラムに追加して動かせ。動いたら、`fillcircle` の呼び出し方を調節して、次の図のように円を配置してみよ。



演習 5-4 次のような手続きを追加して円以外の図形を塗ってみよ。

a. ドーナツ型を塗るメソッド。
b. 長方形または楕円を塗るメソッド。
c. 三角形を塗るメソッド。
d. その他、自分の好きな形を塗るメソッド。

演習 5-5 どの図形でもよいが、色を塗る際に、単色で単純に塗るのでなく、次のような塗り方ができるようにしてみよ。

a. 2 色を指定して、ストライプ、ボーダー、チェックなどで塗れるようにする。
b. 色を塗る際に、「重ね塗り」できるようにしする。つまり**透明度** (transparency)$0 \leq t < 1$ を指定し、各 R/G/B 値について単に新しい値で上書きする代わりに $t \times c_{old} + (1-t) \times c_{new}$ のように混ぜ合わせた値にする。
c. 徐々に色調が変わっていくようにする。(注意: RGB 値は 0〜225 の「整数」でなければならない! 実数で計算した場合はその値が x の場合、「x`.to_i`」で**小数部分を切り捨てて**整数にできる。)
d. ぼやけた形、ふわっとした形などを表現してみよ。
e. その他、美しい絵を描くのにあるとよい機能を工夫してみよ。

演習 5-6 「連番の名前を持つ複数の画像ファイルを生成」することでアニメーションを生成してみよ (下記参照)。

演習 5-7 「美しい絵」を生成するプログラムを作れ。何が美しいかの定義は各自に任されるものとします (アニメーションにしたければしてもよいです)。

5.2.2 おまけ 1: 三角形や凸多角形を塗るには

上で述べた、「図形内という条件を記述する」方法についてもう少し具体例を示しましょう。たとえば、XY 軸と平行な長方形だったら、その XY 座標の小さい側の角を (x_0, x_1)、大きい側の角を (x_1, y_1) とした場合、条件は次のように表せますから簡単ですね。

$$\{\ (x, y) \mid x_0 \leq x \leq x_1 \wedge y_0 \leq y \leq y_1\ \}$$

しかし、XY 軸に対して傾けたい場合は、もっと一般的に考える必要があります。例として三角形を取り上げましょう。三角形は 3 つの辺で囲まれた領域ですよね (当り前だ)。1 つの直線は、平面を 2 つの半平面に分けます。直線だと指定しにくいので、直線に含まれる線分を $(x_0, y_0) - (x_1, y_1)$ で指定することにして、この半平面の点の集合は次の式で表されます。

$$\{\ (x, y) \mid (x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0) \geq 0\ \}$$

なぜそうなるかというと、上の条件式は $\overrightarrow{(x_0, y_0) - (x_1, y_1)}$ と $\overrightarrow{(x_0, y_0) - (x, y)}$ のの外積が正であるという条件であり、一般に起点を共有する位置ベクトル $\vec{a}$ と $\vec{b}$ の外積 $\vec{a} \times \vec{b}$ の符号は $\vec{a}$ から見て $\vec{b}$ が左にある場合には正、右にある場合には負になるからです。

そして、半平面が定義できたら、3 つの半平面に「ともに」含まれる点 (つまり共通集合) が三角形となります (図 5.5)。ということは、条件で言えば上に記したような半平面の条件を 3 つ「すべて」満たす点、ということになります。

三角形に限らず、凸な (へこみの無い) 多角形についても同様の考え方で定義することができます。そして傾いた長方形 (太さのある線分は細長い長方形だと考えることができます) も、こちらの方法で定義することができます。

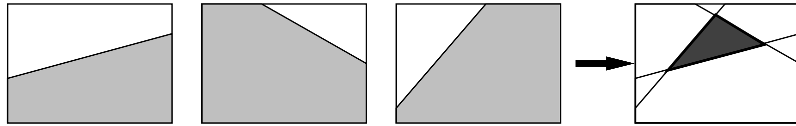


図 5.5: 三角形は3つの半平面の共通集合

5.2.3 おまけ 2: 楕円を塗るには

図形によっては、上述のように直接2次元座標上での条件を記述するのは厄介なものがあります。たとえば楕円を考えてみましょう。皆様は楕円の一般式を言えますか?

しかし、よい方法があります。楕円は円を「引き延ばして作る」ことができますね。ですから、原点を中心とした、たとえば横の半径が3、縦の半径が2の楕円があったとして、それを「横に $\frac{1}{3}$ 倍、縦に $\frac{1}{2}$ 倍に縮めると」半径1の円になるはずです。そして、半径1の円内にあるかどうかは簡単に判定できます。つまり、$p(x, y)$ を $p'(\frac{x}{3}, \frac{y}{2})$ に写像してから円内の判定をすればよいわけです (図 5.6)。原点にない楕円や軸の回転した楕円は? これらも、原点の移動や座標の回転をしてから判断すればいいわけですね。[5]

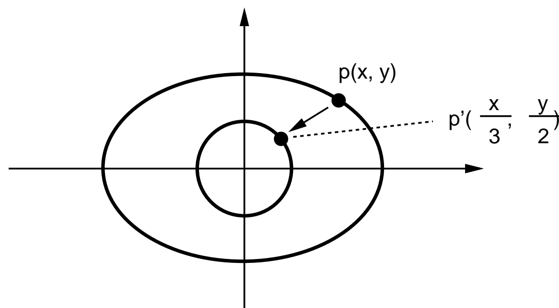


図 5.6: 楕円の内部かどうかの判定

5.2.4 おまけ 3: アニメーションを作るには

演習6のためにアニメーションの例を簡単に説明しましょう。以下は「赤い円が下に動いて行き、青に変わって小さくなる」というアニメーションを作成しています。毎回絵をまっさらにしてから描くために、clearimage というメソッドを追加しています。

```
def clearimage
  200.times do |y|
    300.times do |x|
      $img[y][x].r = 255; $img[y][x].g = 255; $img[y][x].b = 255
    end
  end
end
def myanim
  count = 100
  20.step(80, 4) do |y|
    clearimage
```

[5] こういう変換のことを総称してアフィン変換とか呼びますが、まあ用語はそれとして、適切な行列を作って掛けるとかでできます。

```
      fillcircle(110, y, 30, 255, 0, 0)
      writeimage("a#{count}.ppm"); count += 1
    end
    30.step(10, -1) do |r|
      clearimage
      fillcircle(110, 80, r, 0, 0, 255)
      writeimage("a#{count}.ppm"); count += 1
    end
  end
```

ファイル名は a100.ppm、a101.ppm、…という連番になります (ファイル名の長さが変わって欲しくないので 100 からにしています)。かなり時間が掛かりますが、生成が終わったらこれらの PPM ファイルをたばねてアニメーション GIF にします。

```
  convert -set delay 5 a*.ppm result.gif
```

これをブラウザなどで見ればよいわけです。

5.2.5 おまけ 4: フラクタル

フラクタルな図形というのは、再帰性のある図形 (その図形の一部分が、全体と相似になっているような図形) を言います。たとえば、図 5.7 を見ると、「正方形の 4 隅に小さい正方形がくっついている」という構造が繰り返されています。

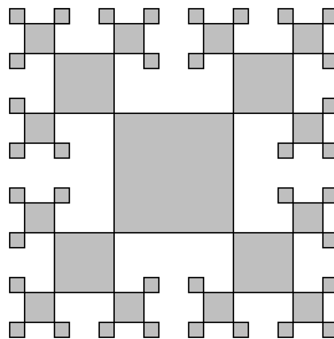


図 5.7: フラクタルな図形の例

こういうのは、再帰的なメソッドで作れます。どのみち、図形が小さくなりすぎたら描けないので、それが終了条件となります。たとえば次のような擬似コードを考えてみてください (正方形を描くメソッド fillsquare は別にあるものとします。)

- squares: 中心 x, y, 1 辺 2 × len の正方形フラクタルを描く
- 　　もし `len < 1` ならば 戻る。
- 　　fillsquare(x, y, len)。
- 　　half ← len / 2。
- 　　squares(x+len+half, y+len+half, half)。
- 　　squares(x+len+half, y-len-half, half)。
- 　　squares(x-len-half, y+len+half, half)。
- 　　squares(x-len-half, y-len-half, half)。

なお、このようにきっちり機械的にやると人工物ぽくなりますが、乱数を使って「大きさがランダムに変動したり」「子供が確率的にできたりできなかったり」すると、自然物ぽくなります (自然界はフラクタルだと言われている)。Ruby では乱数は次の 2 つの方法で使えます。

- rand() — 0 以上 1 未満の実数値の一様乱数が得られる。
- rand(N) — N は整数として、0 から $N-1$ までの整数の一様乱数が得られる。

5.3 演習問題解説 (一部)

5.3.1 2 次元配列

これは簡単なのでコードだけ示します。(c) や (d) は if 式を使うのが素直でしょう。

```
Array.new(5) do|i| Array.new(5) do|j| j-i end end   # (a)
Array.new(5) do|i| Array.new(5) do|j| i**j end end  # (b)
Array.new(5) do|i| Array.new(5) do|j| if i==j then 1 else 0 end end end
Array.new(5) do|i| Array.new(5) do|j|
               if (i-2).abs == (j-2).abs then 1 else 0 end
             end end      # (d)
```

5.3.2 様々な図形

細かい演習問題は省略して、今回は図 5.8 のようなさまざまな図形を描くプログラムを説明します (このために、透明度の機能も追加しました)。

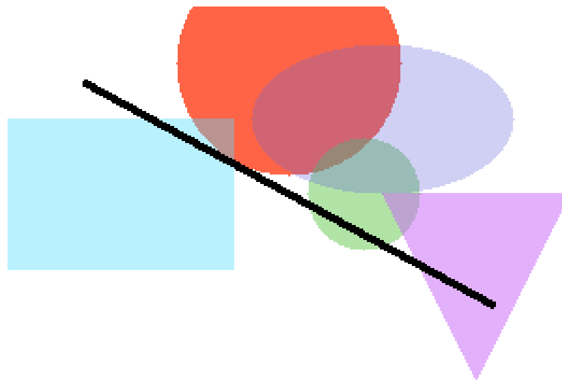


図 5.8: 生成されたさまざまな図形

レコード定義、画像の書き出しについては変更はありません。色に「透明度」をつけて塗れるようにするには、現在の色と塗りたい色を α (透明度) に応じて比例配分すればよいでしょう。それを行うように pset を変更し、あとはすべてこれを使っています。

```
def pset(x, y, r, g, b, a)
  if x < 0 || x >= 300 || y < 0 || y >= 200 then return end
  $img[y][x].r = ($img[y][x].r * a + r * (1.0 - a)).to_i
  $img[y][x].g = ($img[y][x].g * a + g * (1.0 - a)).to_i
  $img[y][x].b = ($img[y][x].b * a + b * (1.0 - a)).to_i
end
```

次に、円を描く (正確には円の形に色を塗る) メソッド fillcircle を示します。ただし、前回は「画像全部の点」に対して判定していましたが、今回は処理を軽くするため、必要にしてできるだけ少ない範囲の点だけを列挙して判定します。それには、円に含まれ得る点の X 座標、Y 座標の範囲 (中心 (x_c, y_c) 半径 r として $x_c \pm r$ と $y_c \pm r$) をまず考え、その範囲内の各点 (x, y) について $(x - x_c)^2 + (y - y_c)^2 \leq r^2$ を満たすなら円内にあるものとしてその点の色を設定します:

```
def fillcircle(x, y, rad, r, g, b, a)
  j0 = (y-rad).to_i; j1 = (y+rad).to_i
  i0 = (x-rad).to_i; i1 = (x+rad).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x)**2+(j-y)**2<rad**2 then pset(i,j,r,g,b,a) end
    end
  end
end
```

XY 座標や半径に小数点つきの値が入れられても動作するように、調べる範囲を計算した時に結果をメソッド to_i で整数にしています。

長方形を描く fillrect は、円よりもっと簡単で、単にその範囲全部を pset するだけです。[6]

```
def fillrect(x, y, w, h, r, g, b, a)
  j0 = (y-0.5*h).to_i; j1 = (y+0.5*h).to_i
  i0 = (x-0.5*w).to_i; i1 = (x+0.5*w).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i| pset(i, j, r, g, b, a) end
  end
end
```

楕円を描く fillellipse は、円と同様で、ただし縦横をそれぞれ縦横の半径で割ってから半径 1 の円に入っているかどうかで判定すればよいでしょう:

```
def fillellipse(x, y, rx, ry, r, g, b, a)
  j0 = (y-ry).to_i; j1 = (y+ry).to_i
  i0 = (x-rx).to_i; i1 = (x+rx).to_i
  j0.step(j1) do |j|
    i0.step(i1) do |i|
      if (i-x).to_f**2/rx**2 + (j-y).to_f**2/ry**2 < 1.0
        pset(i, j, r, g, b, a)
      end
    end
  end
end
```

三角形を描く filltriangle は、凸多角形を塗る fillconvex というのを作ってそれを呼ぶようにしました:

```
def filltriangle(x0, y0, x1, y1, x2, y2, r, g, b, a)
  fillconvex([x0, x1, x2, x0], [y0, y1, y2, y0], r, g, b, a)
end
```

fillconvex は X 座標、Y 座標をそれぞれ配列で渡し、最後には最初と同じ要素を重複して入れておくことにします。また、点を指定する順序は「左回り」である必要があります (これらの理由は後述します)。

fillconvex では、まず座標の範囲は配列に入っている X 座標や Y 座標の最大と最小を求め (最大と最小は前に演習でやったようなものですが、実は配列にはメソッド max と min があって最大と最小を計算してくれるのでそれを使っています)、その後各点についてそれが図形の内側にあるなら塗ります:

[6] 回転させたい場合は後の「直線」を援用してください。

```
def fillconvex(ax, ay, r, g, b, a)
  xmax = ax.max.to_i; xmin = ax.min.to_i
  ymax = ay.max.to_i; ymin = ay.min.to_i
  ymin.step(ymax) do |j|
    xmin.step(xmax) do |i|
     if isinside(i, j, ax, ay) then pset(i, j, r, g, b, a) end
    end
  end
end
```

図形の内側にあるかどうかは isinside で判定します。isinside は、与えられた点が「いずれかの辺の右側にある」なら図形の外にある、そうでなければ内側にあるか線上にある、と判断します。

```
def isinside(x, y, ax, ay)
  (ax.length-1).times do |i|
    if oprod(ax[i+1]-ax[i], ay[i+1]-ay[i], x-ax[i], y-ay[i]) < 0
      return false
    end
  end
  return true
end
```

右側にあるかどうかは、辺の線分の**ベクトル** (vector) と、線分の起点から調べたい点までのベクトルの**外積** (outer product) を計算して、負なら右側と判定します (このために左回りで周囲を指定するという条件が必要なのでした):

```
def oprod(a, b, c, d)
  return a*d - b*c;
end
```

このほか、線分が直交かどうか調べるにはベクトルの**内積** (inner product) が 0 かどうか調べればよいなど、図形処理においてベクトルの考え方はさまざまに活用できます。このような、プログラムで幾何学的な図形の計算を行うものを一般に**計算幾何学** (computational geometry) と呼びます。

線を描く fillline ですが、2 点の XY 座標と「線の幅」を指定します:

```
def fillline(x0, y0, x1, y1, w, r, g, b, a)
  dx = y1-y0; dy = x0-x1; n = 0.5*w / Math.sqrt(dx**2 + dy**2)
  dx = dx * n; dy = dy * n
  fillconvex([x0-dx, x0+dx, x1+dx, x1-dx, x0-dx],
             [y0-dy, y0+dy, y1+dy, y1-dy, y0-dy], r, g, b, a)
end
```

線分のベクトルからそれと直交するベクトルを計算し、その長さが線の幅の半分になるようにします。あとは線分の両端点と幅ベクトルを加減することで細長い長方形ができますから、それを fillconvex で塗ればよいわけです。

では最後に、さまざまな絵を描くメソッドを示します:

```
def mypicture
  fillcircle(150, 30, 60, 255, 100, 70, 0.0)
  fillcircle(190, 100, 30, 100, 200, 80, 0.5)
  fillrect(60, 100, 120, 80, 80, 220, 255, 0.6)
```

```
  fillellipse(200, 60, 70, 40, 100, 100, 220, 0.7)
  filltriangle(200, 100, 300, 100, 250, 200, 200, 100, 250, 0.5)
  fillline(40, 40, 260, 160, 4, 0, 0, 0, 0.0)
  writeimage("t.ppm")
end
```

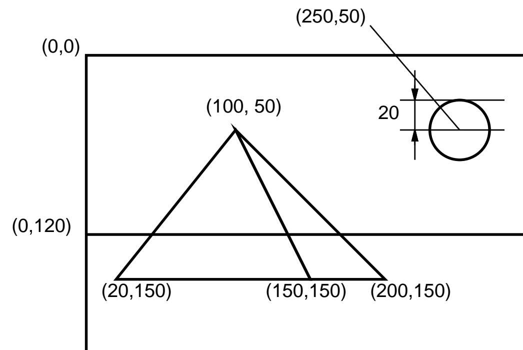


図 5.9: ピラミッドの絵の設計図

演習 5-7 の「美しい」については皆様にお任せしているのですが、たとえば風景みたいに構図のある絵を描くとしたら、やっぱり何らかの設計が必要ではと思います。たとえば「海にうかぶピラミッドと太陽」という絵を描くものとします。まず、図 5.9 のように方眼紙などで構図の設計をして、それからそれぞれの図形を色指定して入れていく、みたいにすればそれらしくなるのではないでしょうか (図 5.10)。

```
def mypicture1
  fillrect(150, 60, 300, 120, 180, 240, 250, 0.0);
  fillrect(150, 160, 300, 80, 20, 90, 200, 0.0);
  filltriangle(100, 50, 150, 150, 20, 150, 120, 70, 20, 0.0);
  filltriangle(100, 50, 200, 150, 150, 150, 160, 90, 80, 0.0);
  fillcircle(250, 50, 20, 255, 0, 0, 0.0);
  writeimage("t1.ppm")
end
```

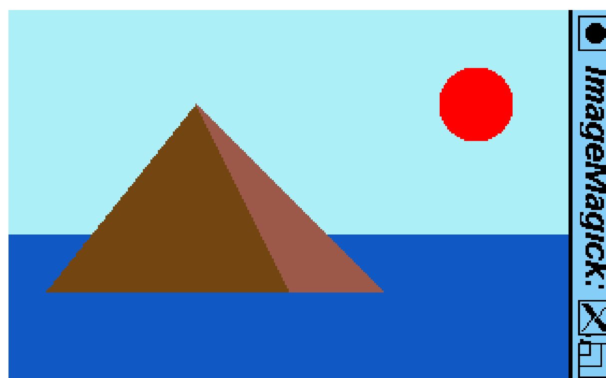


図 5.10: ピラミッドの絵の画像

なかなか大変でしたが、このように手続きを次々に作っていくことで、大きなプログラムでも見通しよく組み立てて行けることが納得いただけたかと思います。

5.4 検討 自分で考えたものを作ること

この部分が、このカリキュラムの非常に特徴的な部分です。つまり、2 次元配列とかレコードとかはとても一般的な話で、さらにこれらを組み合わせて画像が表現できますね、という説明があり、赤い点を 1 つ打つ、のようなプログラムまで説明されたあと「好きな絵を描きなさい」が課題になるわけです。

内容としてはそれ以外に、さまざまな図形を描くためのヒントが沢山出て来ますが (多くはそのための練習問題や解答例として提示)、これらは必須というわけではなく、最終的にどのような絵を作るかは各自に任されます。

このような設定にした意図は、「プログラムは結局、やりたいことを実現する手段として使えるようにならないと面白くないし意味もない」ということを体感してもらう、ということです。たとえば、基本情報処理技術者試験という国家試験がありますが、これを通過してプログラムが書けることになっているのに、実際にはプログラムを全く書くことができない人が多数います。つまり、試験のためだけに覚えて試験だけ通っても役に立たないのです。

肝心なのは「自分はこういうことをプログラムにさせたい」と考えた後、実際に「そういうことができるプログラムを組み立てる」力を持っているかどうか、だと考えます。そしてそのための題材として「絵の生成」が使われているわけです。絵であれば、「好きな絵を描きなさい」という課題は (幼稚園などで) ごく普通ですし、実際に描かせたものがどうなのかは「見れば分かる」からです。

そして、実際にプログラムを組む段階では、基本的な技能だけでもそれなりの絵は作れますし、基本図形については演習問題解説を見てそのメソッドを流用すれば自分で全部書かなくても使えます。そこに無いようなもっと複雑な絵の要素についても、頑張るとそれに応じて作れるようになるので、基礎的な技能を多く持っている学生さんにとっても、この課題はやりがいのあるものとなっているようです。

open question

- 「点の打ち方だけ教えて、好きな絵を描かせる」という課題は、レベルとして高度すぎるでしょうか?
- さまざまな基本図形を描くための手続きを別途学んでもらって、それらを組み合わせて絵が描けるところまでやった後であれば、どうでしょうか?
- ここで示したもののうち、どのような基本図形程度までであれば高校生にやってもらっても大丈夫そうでしょうか?
- このような課題は「すべての画像は個々の色のついた点 (ピクセル) の集まりである」ということの学習題材として有用でしょうか?
- アニメーションを生成できるということを、もっと大きく扱う方が、「お絵描きツールの方が楽でいい」という意見への反論として有用かも知れませんが、どうでしょうか?

第6章　計算量

6.1　さまざまな整列アルゴリズム

6.1.1　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 6-1 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。

6.1.2　基本的な整列アルゴリズム

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。

[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。

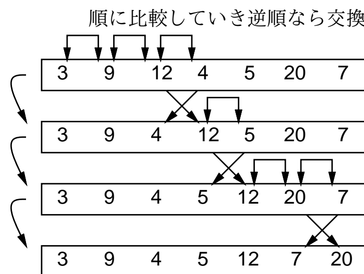


図 6.1: バブルソートによる整列

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 6.1)。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:

```
irb> a = [1, 9, 5, 4, 2]
```

[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。

```
=> [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。

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 6.2)。

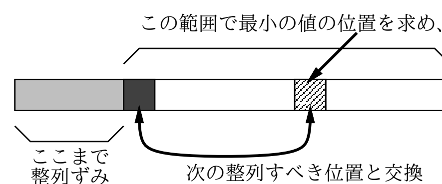


図 6.2: 単純選択法による整列

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
```

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 6.3)。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 6.4)。

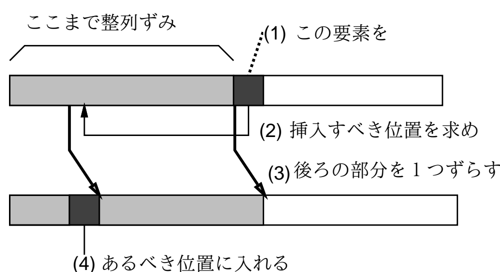


図 6.3: 単純挿入法による整列

ずらすコードと本体のコードは次のとおりです:

```
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
```

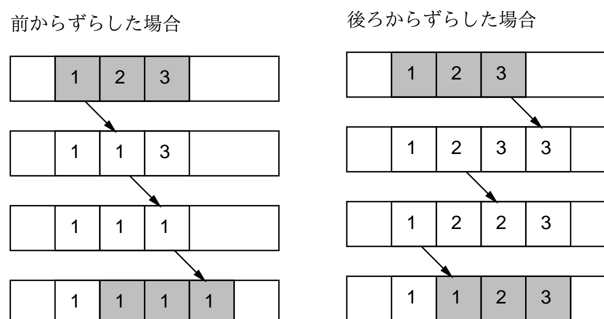


図 6.4: 配列をずらす

```
  end
end
def arrayshiftrange(a, i, j)
  j.step(i, -1) do |k| a[k+1] = a[k] end
end
```

6.1.3 整列アルゴリズムの計測

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 6-2 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

6.1.4 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 6.1 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 6.1: 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。

6.1.5 マージソート

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 6.5 のように

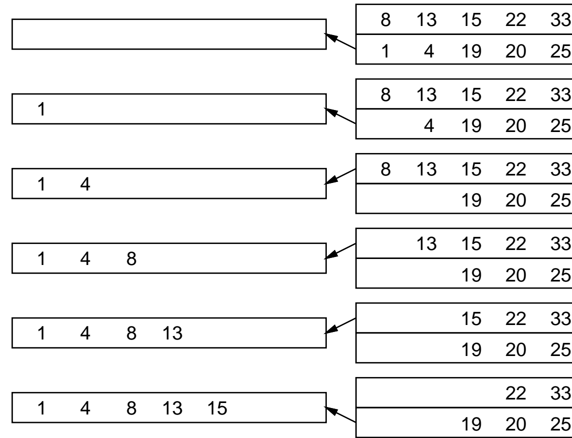


図 6.5: マージの処理

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 $j \leq i$ 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 6.6)。

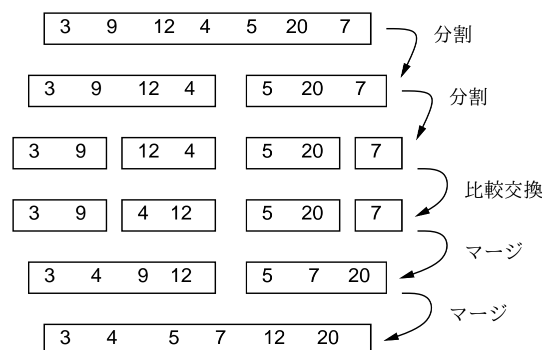


図 6.6: マージソートによる整列

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、
- 　　　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。

6.1.6 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 6.7)。

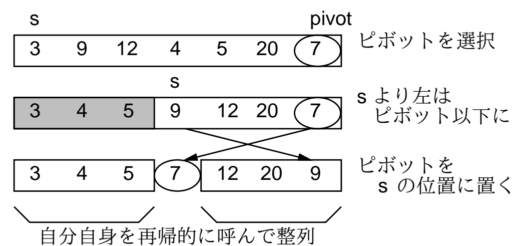


図 6.7: クイックソートによる整列

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。

演習 6-3 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。

[3] `j` 番はピボットが入っているので保留します。

6.2　時間計算量

6.2.1　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 6.8 に示します。

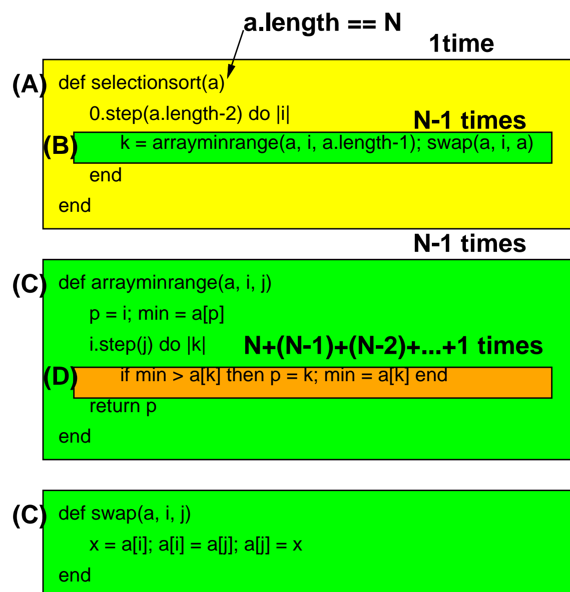


図 6.8: 選択ソートのコード実行回数の検討

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。

6.2.2 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 6.1 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 6-4 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。

6.3 整数値のための整列アルゴリズム

6.3.1 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)。

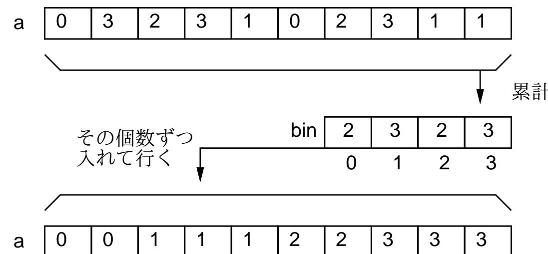


図 6.9: ビンソートによる整列

それなら図 6.9 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]

演習 6-5 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]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。

6.3.2 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 6.10 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 6.10: 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 6.11)。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。

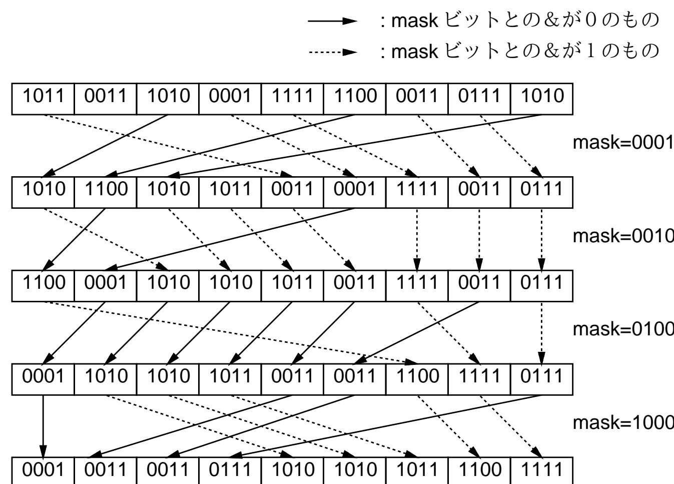


図 6.11: 基数ソートによる整列

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

[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。

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 6-6 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 6-7 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。

6.4　演習問題解説 (一部)

6.4.1　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。

6.4.2 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
```

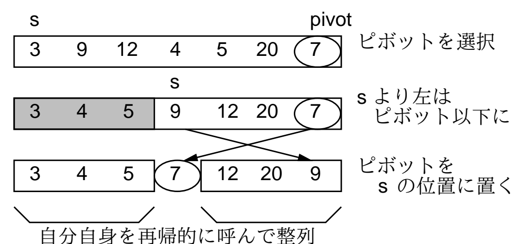


図 6.12: クイックソートによる整列 (再掲)

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 6.12)。

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 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。

---

[7] `j` 番はピボットが入っているので保留します。

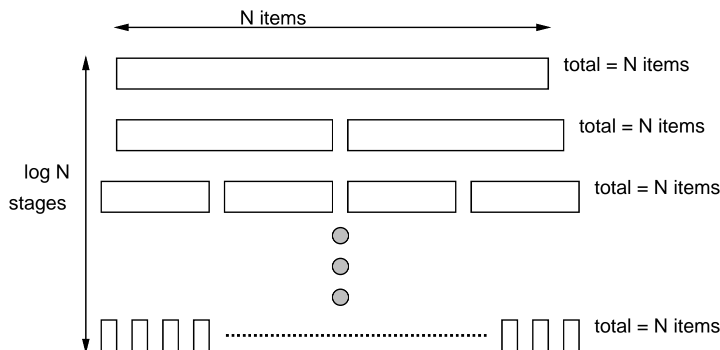


図 6.13: クイックソートのコード実行回数の検討

6.4.3 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 6.14 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。

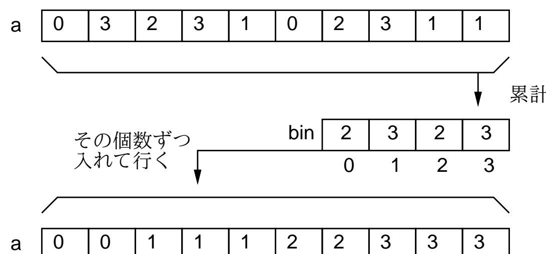


図 6.14: ビンソートによる整列 (再掲)

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
```

6.4.4 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 6.15)。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。

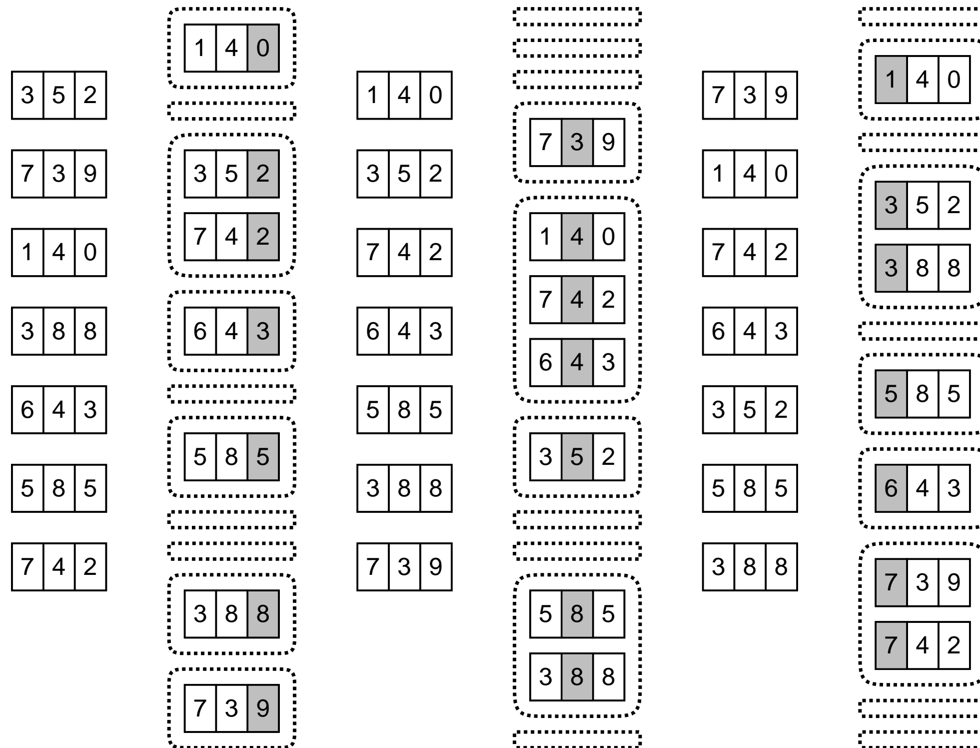


図 6.15: 十進法の場合の基数ソートの例

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

6.4.5 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 6.2 に示します。

表 6.2: 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 6.2 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 6.2 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。

6.4.6 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:

[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。

```
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。

6.5 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。

> 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。

[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。

- $O(1)$ — 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 7-1 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
    # 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
```

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

```
                end
            end
        end
    end
```

6.6 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?

6.7 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。