情報科学 2006　久野クラス　＃2

久野　靖*

2005.10.14

はじめに

1 週間ぶりのごぶさたでしたが、演習とかやってみていかがでしたか? アンケートで色々感想を頂いていますが、まあそれなりという感じのようですので、今後とも同様に行きたいと思います。本日の目標は次の通り:

- 基本的な制御構造 (枝分かれ、繰り返し) について理解し、プログラムが書けるようになる。
- 基本的な制御構造を用いたアルゴリズム/プログラムが考えられるようになる。

1　前回の演習問題の解答例 (一部)

1.1　演習 3a

まず演習 3a であるが、疑似コードは次の通り。

- 実数 x、y を入力する。
- x+y を出力する。

で、Java では次のようになる。

```
import java.io.*;

public class r1ex3a {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = ");
    double x = (new Double(in.readLine())).doubleValue();
    System.out.print("y = ");
    double y = (new Double(in.readLine())).doubleValue();
    double z = x + y;
    System.out.println("x + y = " + z);
  }
}
```

ところで、演習でよくある間違いに

```
    System.out.print("x = ");
    System.out.print("y = ");
    double x = (new Double(in.readLine())).doubleValue();
    double y = (new Double(in.readLine())).doubleValue();
```

とやって動かして見ると

*筑波大学大学院経営システム科学専攻

```
% java r1ex3a
x = y =
...
```

となってしまう、というのがありますが、はまった人いませんか。「x のプロンプトを出す」「x を読む」「y のプロンプトを出す」「y を読む」というふうに**きっちりと順番に**やる必要があるわけです。

1.2 演習 3b

演習 3b は 3a とおんなじようにやればいいのだが、、、

- 実数 x、y を入力する。
- x+y、x-y、x*y、x/y を出力する。

で、Java では次のようにしてみた。

```
import java.io.*;

public class r1ex3b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = ");
    double x = (new Double(in.readLine())).doubleValue();
    System.out.print("y = ");
    double y = (new Double(in.readLine())).doubleValue();
    System.out.println("x + y = " + (x+y));
    System.out.println("x - y = " + (x-y));
    System.out.println("x * y = " + (x*y));
    System.out.println("x / y = " + (x/y));
  }
}
```

自分が作ったのと違う、と思いました? もちろん、それで当然であり、どっちが正解ということはない。ちょうど日本語で同じことを言うのにさまざまな言い方があるのと同じこと。ただし、スマートだとか短いとか分かりやすいとかそういった点で違いはあるかもしれない。美観の問題! だから自分の好みもとりまぜて、考えて選ぶこと。

上の解答例に戻ると、このコードでは和、差、商、積をいったん変数に入れていない。もちろん、前回の例題のように変数に入れてもよいが、上のようにいきなり出力してもよい。一般に変数が書けるところには代りに任意の計算式を書いてもよい。[1]

あと、字下げ (左側に空白を入れること) をしない人がいるけど、これは絶対やめた方がよい。たとえば、後で出て来る話だけど

```
if(...) {          if(...) {
    ....               ....
    ....               ....
}                  }
```

左のように書いてあれば、どこまでが「もし〜ならば」の範囲かすぐ分かる。しかし右のようにべったり揃えてしまうとそれが分からないし、たとえば間違って「}」をつけ忘れたり別の場所に入れてしまうともう大混乱になる。たかがスペースキーを打つ手間を惜しんで後で間違い探しに 1 時間も掛けるのは阿呆みたいでしょ? プログラムは「美しく」書こう。

1.3 演習 3c+3d

演習 3c と 3d は円錐の体積と表面積…表面積はけっこう面倒ですよね。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 で割るだけ。表面積は展開図を考えると、底面の面積はもう分かってるから側面は…側面の半径は $\sqrt{r^2+h^2}$。で、これを l と置くと、側面になる扇型の中心角は、全円周 (ラジアンで表した中

[1] ところで、式を丸かっこで囲んでいるのは、「"...." + x + y」とすると足し算は行われず全部文字列として連結されてしまうから。

心角 2π の場合に相当) の長さ $2\pi l$ と底面の円周の長さ $2\pi r$ の比率 $\frac{r}{l}$ に 2π を掛けたもの。で、扇型の面積は半径 l の円の面積の $\frac{r}{l}$ 倍となるわけだ。以上から表面積は

$$\pi r^2 + \pi l^2 \frac{r}{l} = \pi(r^2 + rl)$$

こういうのが得意な人むけにちょっと遊ぼうと思ったのだけど…大変でしたか?

```
import java.io.*;

public class r1ex3c {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("radius = ");
    double r = (new Double(in.readLine())).doubleValue();
    System.out.print("height = ");
    double h = (new Double(in.readLine())).doubleValue();
    double l = Math.sqrt(r*r + h*h);
    System.out.println("volume = " + (3.1416*r*r*h*0.333333));
    System.out.println("surface = " + (3.1416*(r*r + r*l)));
  }
}
```

ところで、「円周率は 3.1416 じゃないねん!」と思う人もいたようだが…計算機での計算は「電卓での計算」と同じであり、有限の桁数 (double だと 64 ビット) でしか行なえないのであり、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。もっとも、3.141592653589793 くらいまでは扱えるので、それをいちいち書くのは嫌だという人のために `Math.PI` と書いてもいい (ついでに e は `Math.E`)。

1.4 演習 4

このようなさまざまなプログラムを作ってみることが考えられるので、これはあくまでも一例。

```
import java.io.*;

public class r1ex4a {
  public static void main(String[] args) {
    int x = 1;
    System.out.println("x = " + x);
    x = x*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2*2;
    System.out.println("x*pow(2,30) = " + x);
    x = x*2;
    System.out.println("x*pow(2,31) = " + x);
    x = x*2;
    System.out.println("x*pow(2,32) = " + x);
    int y = -2147483648;
    System.out.println("y = " + y);
    y = -y;
    System.out.println("neg(y) = " + y);
    y = y + 1;
    System.out.println("y = " + y);
    y = -y;
    System.out.println("neg(y) = " + y);
  }
}
```

このプログラムはまず、2^{30}、2^{31}、2^{32} を打ち出す。続いて、「とある」非常に絶対値の大きいマイナスの数をまず表示し、次にそれを符号反転したものを打ち出す。そして 1 足して、また符号反転して打ち出す。結果はこうなる。

```
% java r1ex4a
x = 1                          ←確かに 1
x*pow(2,30) = 1073741824       ←確かに大きい数
x*pow(2,31) = -2147483648      ←マイナスに?
x*pow(2,32) = 0                ← 0 に?
y = -2147483648                ←マイナスの大きい数
neg(y) = -2147483648           ←符号反転しても同じ?
```

```
y = -2147483647              ← 1 つだけ絶対値を減らすと
neg(y) = 2147483647          ←符号反転できるようになる
%
```

前半については、2 ずつ掛けるということは、$1_{(2)}$、$10_{(2)}$、$100_{(2)}$、…のように (2 進表記で) 下に「0」がついて行くということ。そして 2^{30} だと「$1000000000000000000000000000000_{(2)}$ と 30 個 0 が並ぶので、桁数は 31 桁。もう 1 回 2 倍にすると「$10000000000000000000000000000000_{(2)}$」になるが、この「1」は最上位つまり符号桁にある「1」なので、この数はマイナスの大きな数になってしまう。さらにもう 1 回 2 倍にすると「$100000000000000000000000000000000_{(2)}$」になって、0 だけで 32 個あるので 32 ビットにはもう「1」が入り切らなくて「$000000000000000000000000000000000_{(2)}$」つまり 0 になってしまう。

後半であるが、また「$100000000000000000000000000000000_{(2)}$」から始めて、これを 2 の補数表現で符号反転するには「0 と 1 を入れ替えてから 1 足す」のだった。入れ替えた状態は「$011111111111111111111111111111111_{(2)}$」なので、これに 1 を足すと「$100000000000000000000000000000000_{(2)}$」に戻ってしまうので値が変わらない。しかし 1 つだけ大きい (0 に近い) 数だと…それは「$100000000000000000000000000000001_{(2)}$」なわけだけど…0 と 1 を入れ替えて「$011111111111111111111111111111110_{(2)}$」、これに 1 を足して「$011111111111111111111111111111111_{(2)}$」つまり 32 ビットの 2 の補数で表現できる最も大きな正の数になる。

1.5 演習 5

以下のもあくまでも 1 例。有名な例として「0.1」という定数は 10 進数では大変キリがいい数だが、2 進数では無限小数になってしまう (つまり誤差が避けられない)、というのを取り上げてみた。

```
import java.io.*;

public class r1ex5a {
  public static void main(String[] args) throws Exception {
    double x = 7.0;
    double y = 0.1 * x;
    System.out.println("y = " + y);
    double e1 = 0.000000000000001;
    double e2 = 0.0000000000000005;
    double e3 = 0.00000000000000025;
    System.out.println("x + e1 = " + (x + e1) + ", e1 =" + e1);
    System.out.println("x + e2 = " + (x + e2) + ", e2 =" + e2);
    System.out.println("x + e3 = " + (x + e3) + ", e3 =" + e3);
  }
}
```

実行したようすは次の通り:

```
% java r1ex5a
y = 0.7000000000000001
x + e1 = 7.000000000000001, e1 =1.0E-15
x + e2 = 7.000000000000001, e2 =5.0E-16
x + e3 = 7.0, e3 =2.5E-16
%
```

なお、「E-15」というのは「$\times 10^{-15}$」の意味で、この書き方はプログラミング言語で指数つきの数を書くときの標準的なものである。「3.0×10^8」は「3E8」になる (分かりにくいですかね?)。

話を戻すと、つまり 7 を 0.1 倍すると誤差のついた数になり、それを 10 倍するとその誤差のため 7 より「わずかに」大きい数になる。では 7 以外ではどうかというと、計算のとき「丸め」が起きるため、数によってこのような現象が見られるかどうかは変わってくる。

次に、この「微妙に大きい数」とその 1/2、1/4 を変数 e1、e2、e3 に入れて 7 と加えてみる。そうすると、1/2 までは値が増えるが 1/4 だと値が増えない。このように、絶対値が大きく違う値の加減算は浮動小数点の「有効数字の一番下のあたりだけ」で計算することになり、精度が悪くなる。(これとは逆に、ほとんど同じだが微妙に異る数どうしの引き算なども精度が悪くなる。)

演習 6 は略しますが、NaN とか∞とかが現れることが分かっていればとりあえずいいんではないでしょうか。

2　アルゴリズムと制御構造

2.1　基本的な制御構造

ここまでに出て来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、一番下まで来たらそれでおしまい、というものだった。それでも単純な計算なら問題なくできるが、手順が複雑になってくると「枝分かれ」や「繰り返し」などが必要になってくる。このような、実行の流れを表す構造のことを一般に制御構造と呼ぶ (動作を順番に書くと順番に実行される、というのも実は「連接」という制御構造ということになっている)。

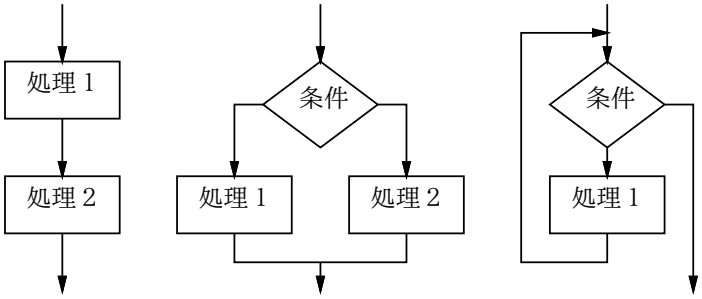

図 1: 3 つの基本的な制御構造

この 3 種類の構造 (連接、枝分かれ、繰り返し) は図 1 のような流れ図に対応している。それで、なぜこの 3 つが基本的かというと、「どんなにごちゃごちゃの流れ図でもこの 3 つの組み合わせに書き換えられる」という定理があり、そのためにこの 3 つだけあればどのような処理の流れでも表現可能だからである。

たとえば、図 2 左のような「入り組んだ」流れ図があるとする。これをこのまま「何をするか」理解するのは大変だが、同じ動作だが基本的な制御構造の組み合わせになるような流れ図に (中) に書き換え、さらに条件の組み合わせを見て簡単化する (右) と、何をするかがずっとよく分かるようになる。[2]

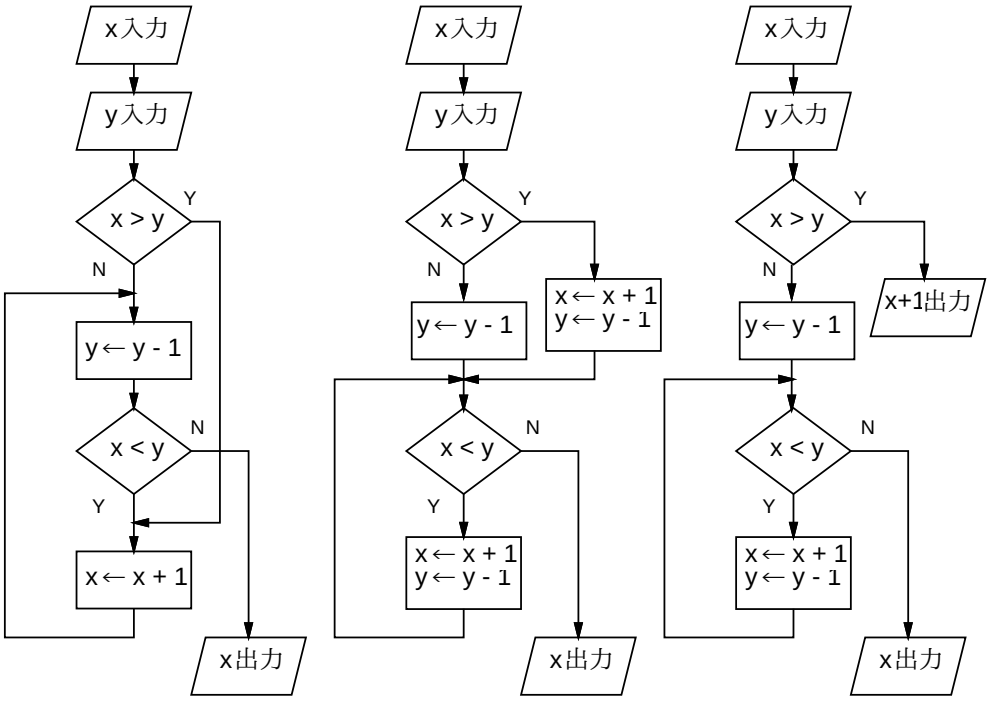

図 2: 分かりにくい流れ図と構造に従った流れ図

[2]右側の流れ図を見ると、「x > y なら x+1 を出力」はすぐわかる。繰り返しの中では、x が y より小さい間 x を 1 増やし、y を 1 減らすので、x と y が同じ値に近付くことになる。x+y が偶数ならちょうど等しくなって終わるが、奇数なら x が y より 1 多くなる。このため、「x ≤ y なら $\lfloor \frac{x+y}{2} \rfloor$ を出力」と分かる。

このため、現在ではアルゴリズムを 3 種類の構造の組み合わせで表すのがコンピュータサイエンス分野では標準的となってきていて、そのため「入り組んだ」構造が簡単に書けてしまう流れ図の代わりに疑似コードが使われるようになっている。

2.2 枝分かれと if 文

さて、ここまでに説明した機能だけでは、上から順に動作を実行していくようなプログラムしか作れない。それでは不便なので、枝分かれを学ぶ。枝分かれの疑似コードは次のような形で書く:

- 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。

なお、「動作 2」が不要なら「そうでなければ」も書かなくてよい。これを Java では次のように「if 文」と呼ばれる文を使って書く (右は「動作 2」のない場合):

```
if( 条件 ) {
  ... 動作 1 ...
} else {
  ... 動作 2 ...
}
```

```
if( 条件 ) {
  ... 動作 1 ...
}
```

「条件」については、当面は次のものがあると思っておいて欲しい。

- 比較演算 — たとえば「`x > 10`」のようなもの。>(より大)、>=(以上)、<(より小)、<=(以下)、==(等しい)、!=(等しくない) がある。[3]
- 条件の組み合わせ — 「条件 1 `&&` 条件 2」(A かつ B)、「条件 1 `||` 条件 2」(A または B)、「`!`条件 1」(A でない) が使える。複数のかつ、またはを組み合わせることもでき、適宜かっこを使ってもよい。

2.3 例題: 絶対値の計算

では具体的な例題として、「入力 x の絶対値を計算する」ことを考えよう。まず疑似コードを示す:

- 数値 x を入力する。
- もし $x < 0$ ならば、
- 　$abs \leftarrow -x$。
- そうでなければ、
- 　$abs \leftarrow x$。
- 枝分かれ終わり。
- 数値 x とその絶対値 abs を出力する。

考え方としては簡単ですね? これを Java にしてみよう (混同しないようにクラス名の冒頭に授業の回をつけるようにした):

```
import java.io.*;

public class R2Sample1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value X = ");
    double x = (new Double(in.readLine())).doubleValue();
    double abs;
    if(x < 0.0) {
      abs = -x;
    } else {
      abs = x;
```

[3] Java では「`!`」は「否定」を表すのに使っている。階乗ではないので注意。

```
    }
    System.out.println("value: " + x + " absolute value: " + abs);
  }
}
```

これまで、変数は使うところの最初で「double 名前 = 初期値;」のようにして宣言 (使いますよとコンパイラに教える) と初期化を一緒にしてきたが、abs は宣言だけで初期化がない。これは、変数の宣言を { ... } の中に書くとその範囲内でしか有効にならないので、if の外側に書かなければいけないのでしかたがない。

ところで、同じプログラムだがこう書いたらどうだろう?

- 数値 x を入力する。
- $abs \leftarrow x$。
- もし $x < 0$ ならば、
- $\quad abs \leftarrow -x$。
- 枝分かれ終わり。
- 数値 x とその絶対値 abs を出力する。

このように、「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。Java プログラムでも同様。

```
import java.io.*;

public class R2Sample1b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value X = ");
    double x = (new Double(in.readLine())).doubleValue();
    double abs = x;
    if(x < 0.0) {
      abs = -x;
    }
    System.out.println("value: " + x + " absolute value: " + abs);
  }
}
```

先のプログラム例と、どちらが好みですか?

ところで、「わざわざ変数 abs を用意しなくても、変数 x の符号を反転させたら?」と思ったかも知れない。確かに、このプログラムの場合はそれでもよいけれど、場合によっては後で「元の値」と「絶対値」の両方が必要になるかも知れない。一般に、1 つの変数を複数の意味 (「x」と「x の絶対値」) で使うのは混乱する (書く人も、読む人も) からやめた方がよい。ただし、x はすぐに絶対値に直して、あとは絶対値だけしか使わない、ということなら x を直接書き換えてもよい。

要するに、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くるし、人によっても基準が違うところがある。早く自分の基準を見つけよう!

演習 1 上の絶対値計算プログラムの好きな方のバージョンを打ち込んでそのまま動かせ。

演習 2 枝分かれを用いて、次の動作をする Java プログラムを作成せよ。

a. 2 つの異なる実数 a、b を入力し、より大きい方を表示する。

b. 3 つの異なる実数 a、b、c を入力し、最も大きいものを表示する。やる気があったら 4 つでやってみてもよい。

c. 実数を 1 つ入力し、それが正なら「positive」、負なら「negative」、零なら「zero」と表示する。

2.4 繰り返しと while 文

枝分かれができても、まだプログラムで大したことができる気がしない。計算機の特徴は、どんなつまらない単純作業でもいくらでも繰り返してくれることにあるので、繰り返しが使えるとプログラムもぐっと役に立つ。

繰り返しを疑似コードで記述するときは、次のように書く:

- ～ である間繰り返し、

- 　動作 1。
- 繰り返し終わり。

「〜」のところには条件を記述する。書けるものは if 文の条件とまったく同じである。この繰り返しを Java で書く場合は「while 文」を使う:

```
while( 条件 ) {
  ... 動作 1 ...
}
```

while 文は形だけなら if 文より簡単だが、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かも知れない。たとえば次のような感じで考えれば分かりやすいだろう:

- 「〜」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 (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そこで終わるわけである。

2.5 例題: 数値積分

繰り返しの具体的な題材として**数値積分**を取り上げて見よう。定積分を求めるのって、積分の公式を覚えたりあてはめ変形に苦労したり、大変でしたよね? もうそんな心配はない、コンピュータがあれば元の関数から直接計算してしまえる (実はそんなにいいものでもないけど)。

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは図 3 のように、その $[a, b]$ 部分の関数の下側の面積、ですよね? (なぜだかは聞かないように、私は理系だけど数学は大嫌いです。)

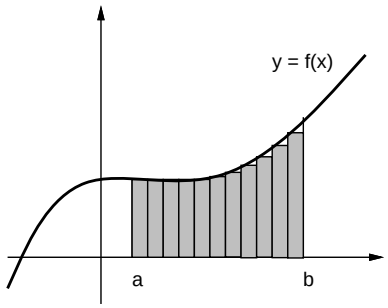

図 3: 数値積分の原理

だったら、この図 3 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いいわけだ。幅は区間を n 当分した値 dx にするものとして、高さは $f(x)$ を計算すれば済むので簡単である。この方法だったら $f(x)$ が解析的に不定積分が求められないヘンテコな関数だろうと計算するだけだからへっちゃらである。

とはいえ、今はプログラムを作って正しい値が求まるかどうかチェックしたいので、ひじょーに簡単な関数 $y = x^2$ でやってみる。不定積分は $\frac{1}{3}x^3$ だから、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ ということになる。たとえば $[1, 10]$ だったら $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ ということになる。

ではアルゴリズムを作って見る。

- a、b、n を入力する。

- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- $x \leftarrow a$。
- $x < b$ が成り立つ間繰り返し:
- 　　$y \leftarrow x^2$。 //関数 $f(x)$ の計算
- 　　$s \leftarrow s + y \times dx$。
- 　　$x \leftarrow x + dx$。
- 繰り返しおわり。
- s を出力する。

要するに、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) 動かして行き、b まで来たら繰り返しを終わる。このように、繰り返しは「こういう条件で変数を動かして行きこうなったら終わる」という考え方が必要なわけである。面積の方は、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行くことで、合計が求まるわけだ。
　なお、「//」の後ろに書いたのは注記 (コメント) で、Java 言語でも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る。では Java プログラムを示そう。

```
  import java.io.*;

  public class R2Sample2 {
    public static void main(String[] args) throws Exception {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("value a = ");
      double a = (new Double(in.readLine())).doubleValue();
      System.out.print("value b = ");
      double b = (new Double(in.readLine())).doubleValue();
      System.out.print("value n = ");
      int n = (new Integer(in.readLine())).intValue();
      double dx = (b - a) / n;
      double s = 0.0;
      double x = a;
//    int count = 0;
      while(x < b) {
        double y = x * x; // f(x) の計算
        s = s + y * dx;
        x = x + dx;
//      count = count + 1;
//      System.out.println("count: " + count + " x: " + x);
      }
      System.out.println("integral: " + s);
    }
  }
```

3 箇所ほど行頭に「//」があって行全体がコメントになっているが、これは後で使う。とりあえずないものと思って読んで欲しいが、やっていることは上の疑似コードそのままである。なお、n は「回数」を入れるので double(実数型) ではなく int(整数型) を使っていることに注意。
　さて、333 が求まるだろうか? 実行させてみる:

```
  % javac R2Sample2.java
  % java R2Sample2
  value a = 1
  value b = 10
  value n = 100
  integral: 337.5571499999994     ←ふーん?
  % java R2Sample2
  value a = 1
  value b = 10
  value n = 1000
  integral: 332.55462150000733     ←小さい
  % java R2Sample2
  value a = 1
  value b = 10
  value n = 10000
  integral: 333.04545121491196     ←大きい…
  %
```

なんだかヘンである。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 (変数 count に回数を数えつつ x を表示するようにして) 動かし直してみた:

```
% java R2Sample2
value a = 1
value b = 10
value n = 100                    ← 100 分割のつもり
count: 1 x: 1.09
count: 2 x: 1.1800000000000002 ←誤差が…
count: 3 x: 1.2700000000000002
count: 4 x: 1.3600000000000003
...
count: 98 x: 9.81999999999999
count: 99 x: 9.90999999999999
count: 100 x: 9.99999999999999
count: 101 x: 10.08999999999999 ← 101 回目が…
integral: 337.5571499999994      ←このため多い
%
```

区間を 100 個でやってみると、長方形を 1 個余計に加えてしまってそれで多すぎるようだ。しかしなぜこんなことが起きるのだろうか? それは、$x \leftarrow x + dx$ で x を増やしていって b になったらやめる、というアルゴリズムの問題である。そもそもコンピュータでの計算は近似値なので、最初に dx を区間長の $\frac{1}{100}$ に計算したとしても、そこには誤差がある。誤差のため、100 回足した後でもわずかに b より小さい場合には、さらにもう 1 回繰り返しを実行してしまうわけだ。

2.6 計数ループと for 文

ではどうすればいいのだろうか。繰り返しが 100 回と決めているのだから、回数を数えるのは整数型の変数で行い、それをもとに各回の x を計算するのがよい。つまり、次のようなループを書くことになる:

```
int i = 0;       // i は「カウンタ」(数を数える変数)
while(i < n) {   // 「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
}
```

このようなループを「計数ループ」と呼ぶ。このようなループは良く使うので、読みやすさのために「for 文」という構文を使って次のように書くのが普通である:

```
for(int i = 0; i < n; ++i) {
  ...            // ここでループ内側の動作
}
```

「++i」というのは「i を 1 増やす」という演算で、つまり「i = i + 1」と同じ。だからこの記述は上の while 文を使った記述と同じなのだが、カウンタの初期化、条件、カウンタを増やす、というループの構成要素がまとめて書かれているので読みやすい。疑似コードでもこのような計数ループは次のように書く:

- 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
- 繰り返しおわり。

2.7 例題: 数値積分 (つづき)

余談が終わったので、では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る:

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

- 繰り返し終わり。
- s を出力する。

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ある。では、これを Java プログラムにしたものを示す。

```
import java.io.*;

public class R2Sample2b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("value b = ");
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("value n = ");
    int n = (new Integer(in.readLine())).intValue();
    double dx = (b - a) / n;
    double s = 0.0;
    for(int i = 0; i < n; ++i) {
      double x = a + i * (b - a) / n;
      double f = x * x;
      s = s + f * dx;
    }
    System.out.println("integral: " + s);
  }
}
```

これを動かしてみる:

```
% javac R2Sample2b.java
% java R2Sample2b
value a = 1
value b = 10
value n = 100
integral: 328.55715000000004
% java R2Sample2b
value a = 1
value b = 10
value n = 1000
integral: 332.5546214999998
% java R2Sample2b
value a = 1
value b = 10
value n = 10000
integral: 332.9554512149995
% java R2Sample2b
value a = 1
value b = 10
value n = 100000
integral: 332.9955450121489
%
```

今度はきざみを小さくすると順当に誤差が減少して行くことが分かる。しかし、常に正しい面積である 333 より小さいのはなぜだろう?

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、増大する関数では図 4 のように微小な三角形のぶんだけ面積が小さめに計算されてしまうからである (逆に減少する関数だと大きめに計算される)。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欲しい。

演習 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。

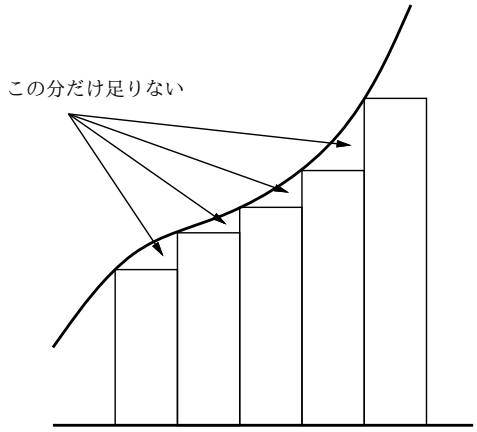

図 4: 区間の左端を使う場合の誤差

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 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 を入力し、2^n を計算する。

b. 整数 n を入力し、n の階乗 $(n \times (n-1) \times \cdots \times 2 \times 1)$ を計算する。

c. 整数 n と整数 $r(\leq n)$ を入力し、${}_nC_r$ 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せよ。

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

$$ cosx = \frac{1}{0!}x^0 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots $$

実際に値の分かる x を入れて精度を確認してみること。$\pm 100\ \pi$ とかだとどうか? n はいくつくらいが必要か?

3 整数と実数

先の例題で `int` 型が出て来たついでにもうちょっと。数学の世界では整数とは実数の部分集合だが、計算機の世界では全然違う! 整数型は次のような特徴を持っている。

- 誤差がない (扱える最大の値と最小の値はある)。
- 実数より計算の効率がよい。
- 「N 番目」を指定するような場所で使うことができる。

だから、計算するといっても「ここは整数がいいか?」「ここは実数がいいか?」を絶えず意識しておく必要がある。あと、実数型が必要なところで整数型を使うと自動的に実数に変換してくれる (整数を実数に変換しても情報が失われることはないから)。

```
int i = 12345;
double f = i; ←自動的に変換
.... i + 3.14 ... ←こういうのも自動変換
```

逆に実数を (小数点以下を切捨てて) 整数に変換するには「`(int)` 式」という形で指定しないといけない。これを「キャスト」(強制型変換) と呼ぶ。

```
double g = 3.1416;
int k = (int)g; ←こうしないと javac でエラーに
.... 2 * (int)(g / 7.0) ... ←こういうのもキャスト
```

覚えておいてください。

4　制御構造の組み合わせ

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る。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ある。

```
if( ... ) {
    while( ... ) {
        ...
    }
}
```

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る。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) 作れるのと同じである。

演習 5 2 数 a、b の最大公約数を $gcd(a,b)$ と記すことにする。正の整数 x、y について $gcd(x,y)$ を求めることを考える。以下 (☆) に注意。

- $x = y$ のとき、$gdc(x,y) = x = y$。
- $x > y$ のとき、$gcd(x,y) = gdc(x-y,y)$。
- $x < y$ のとき、$gcd(x,y) = gdc(x,y-x)$。

これを利用して、2 つの正の整数 x、y を読み込み、その最大公約数を出力するアルゴリズムの疑似コードを書き、それを実現した Java プログラムを作成せよ (☆の簡単な証明も書くこと)。

演習 6 「正の整数 N を読み込み、N が素数か否かを教えてくれる Java プログラム」を書け。まず疑似コードを書き、それから Java に直すこと。(ヒント: N が素数ということは、N を $2 \sim N-1$ のいずれで割っても余りが出るということ。剰余は演算子「%」で計算できる。たとえば「7 % 4」は 3。)

演習 7 「正の整数 N を読み込み、N 以下の素数をすべて打ち出す Java 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。(もちろん N が大きくなるように工夫してくれるとなおよい。)

A　本日の課題 2A

今日は「演習 2」で動かしたプログラムを含む小レポートを久野まで電子メールで送ってください。具体的な内容は次の通り。

1. Subject: は「Report 2A」とする。[4]
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答 (簡単でよい)。

   Q1. プログラムを打ち込んで動かすのに慣れましたか?

   Q2. 自分にとって次の「難しいポイント」は何だと思いますか?

   Q3. 本日の全体的な感想と今後の要望をお書きください。

[4]注意! 1 バイトコード (いわゆる半角) で、大文字小文字もこの通りに、符号化なしで! プログラムで処理するので、この通りでないと間違って処理される可能性がいくらか高くなります。もちろんチェックはしますが。

B　次回までの課題 2B

次回までの課題は「演習 3」〜「演習 7」までの (小) 課題からプログラムを 2 つ以上作ること。ただし「演習 4」から選ぶのは最大 1 個とする。

レポートは授業開始 10 分前 (10:30) までに、上記と同様に久野までメールで送付してください。具体的な内容は次の通り。

1. Subject: は「Report 2B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. 説明と考察。
5. 選んだプログラムもう 1 つのソース。
6. 説明と考察。
7. 下記のアンケートの回答。

   Q1. 枝分かれや繰り返しの動き方が納得できましたか?
   Q2. 枝分かれと繰り返しのどっちが難しいですか? それはなぜ?
   Q3. 課題に対する感想と今後の要望をお書きください。