情報科学 2006　久野クラス　＃3

久野　靖*

2006.10.20

はじめに

今回はまず前回の課題の解説と併せて、数値積分や制御構造などで追加すべき点を説明します。また、本日の新しい内容として次のものを取り上げます。

- $f(x) = 0$ の求解
- 基本データ型、配列型とその利用

とくに配列を学ぶとデータを沢山保持しておけるようになるので、実際に「お仕事で計算する」時に役立つと思います。

1　演習問題解説 — 枝分かれ

1.1　演習 2a

演習 2a は 2 つの枝分かれですから例題とほとんど同じ。まず疑似コードを見よう。

- 実数 a、b を入力する。
- もし a > b であれば、
- 　　max ← a。
- そうでなければ、
- 　　max ← b。
- 枝分かれおわり。
- max を出力する。

Java では次の通り。

```
import java.io.*;

public class r2ex2a {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = ");
    double b = (new Double(in.readLine())).doubleValue();
    double max;
    if(a > b) {
      max = a;
    } else {
      max = b;
    }
    System.out.println("larger is: " + max);
  }
}
```

ところで、要点部分だけ示すとして、次のようなコードを書いてコンパイラに怒られていた人がいたが、どう思いますか?

*筑波大学大学院経営システム科学専攻

```
double max;
if(a > b) { max = a; }
if(a < b) { max = b; }
```

これをコンパイルすると「変数 max は初期化されていない可能性があります。」といわれる。つまり max は最初に値を入れずに宣言だけして、a と b の大小に応じてどちらかが入るから…もし等しかったら? その時は max は「何も入らないまま」になりますね? それはまずいでしょう、というのがこのメッセージの意味。しかし実は。

```
double max;
if(a > b) { max = a; }
if(a <= b) { max = b; }
```

これでも同じエラーが出る。実は Java コンパイラは「この 2 つの条件の少なくとも片方は必ず成り立つ」と推論してくれるほど賢くない。このような場合にエラーメッセージを消すには max に初期値を入れておけばいい…

```
double max = 0.0;
if(a > b) { max = a; }
if(a <= b) { max = b; }
```

けど、気持ち悪いでしょう? やっぱり「どちらか片方を実行」する場合はこういうのではなく「if-else」を使いましょう、ということ。しかし、次のような「別解」はどうだろう。

- 実数 a、b を入力する。
- max ← a
- もし b > max であれば、
- 　　max ← b
- 枝分かれおわり。
- max を出力する。

これの Java 版は次のとおり。

```
import java.io.*;

public class r2ex2a1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = ");
    double b = (new Double(in.readLine())).doubleValue();
    double max = a;
    if(b > max) {
      max = b;
    }
    System.out.println("larger is: " + max);
  }
}
```

どっちが好みですか? これもどちらが正解ということではない。

1.2 演習 2b — 枝分かれの入れ子

しかし演習 2b はもうちょっとややこしい。まず考えるのは、a と b の大きい方はどっちか決めて、それぞれの場合についてそれを c と比べるというもの。

- 実数 a、b、c を入力する。
- もし a > b であれば、
- 　　もし a > c であれば、
- 　　　　max ← a。
- 　　そうでなければ、

- max ← c。
- 枝分かれおわり。
- そうでなければ、
- もし b > c であれば、
- max ← b。
- そうでなければ、
- max ← c。
- 枝分かれおわり。
- 枝分かれおわり。
- max を出力する。

うーむ大変だ。これを Java にしておく。

```
import java.io.*;

public class r2ex2b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = ");
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("c = ");
    double c = (new Double(in.readLine())).doubleValue();
    double max;
    if(a > b) {
      if(a > c) {
        max = a;
      } else {
        max = c;
      }
    } else {
      if(b > c) {
        max = b;
      } else {
        max = c;
      }
    }
    System.out.println("largest : " + max);
  }
}
```

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。ときに、先の別解から発展させるとどうだろう?

- 実数 a、b、c を入力する。
- max ← a
- もし b > max であれば、max ← b。
- もし c > max であれば、max ← c。
- 枝分かれおわり。
- max を出力する。

この方がすっきりしているでしょう? Java でも次のとおり。

```
import java.io.*;

public class r2ex2b1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = ");
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("c = ");
    double c = (new Double(in.readLine())).doubleValue();
```

```
    double max = a;
    if(b > max) { max = b; }
    if(c > max) { max = c; }
    System.out.println("largest : " + max);
  }
}
```

今度はどちらが好みですか? 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方が分かりやすいといえる。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ね。なお、if 文や「もし」の疑似コードが 1 行に書かれているが、この場合はこちらのの方が見やすいと思ったので。

1.3 多方向の枝分かれ

演習 7c は 3 通りに分かれるのだから、if の中にまた if が入るのはやむを得ない。しかし、ちょっと工夫すると分かりやすくなる。Java コードから見ていただく。

```
import java.io.*;

public class r2ex2c {
  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();
    if(x > 0.0) {
      System.out.println("positive.");
    } else if(x < 0.0) {
      System.out.println("negative.");
    } else {
      System.out.println("zero.");
    }
  }
}
```

これは実は最初の if の else のすぐ後ろに次の if がくっついた、という形をしているが、こういうパターンを利用するとプログラムが分かりやすくなる。順序が前後したが、疑似コードだと次のようになる。

- 実数 x を入力する。
- もし x > 0 ならば、
- 　　「positive.」と出力。
- そうでなくて x < 0 ならば、
- 　　「negative.」と出力。
- そうでなければ、
- 　　「zero.」と出力。
- 枝分かれおわり。

一般に「そうでなくて〜ならば、」は何回現われてもよい。またそのどれもが成り立たない場合は「そうでなければ」に来るが、この部分は不要ならなくてもよい。これを Java にする場合は次の形になる。

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

なお、これはあくまでもJavaのif文の「elseの後にすぐ次のifをくっつけた」というパターンなだけで、特別な文というわけではない。

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`が最大だと分かる。これを利用した3方向枝分かれで書くこともできる:

```
import java.io.*;

public class r2ex2b2 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    double a = (new Double(in.readLine())).doubleValue();
    System.out.print("b = ");
    double b = (new Double(in.readLine())).doubleValue();
    System.out.print("c = ");
    double c = (new Double(in.readLine())).doubleValue();
    double max;
    if(a > b && a > c) {
      max = a;
    } else if(b > c) {
      max = b;
    } else {
      max = c;
    }
    System.out.println("largest : " + max);
  }
}
```

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に比例してしまう。もっともだからいけないというわけではなく、Nの個数がいくつと決まっていればこのやり方を使ってもいいかも知れない。

1.4 文字列を変数に入れる

ところで、上の例解では枝分かれの中で出力することでメッセージを切替えていた。そうじゃなくて、これまでのように最後の1個所で出力したければどうしたらいいだろう? それは出力したい文字列を変数に入れておけばよい。Javaのコードだけ示す。

```
import java.io.*;

public class r2ex2c1 {
  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();
    String mesg;
    if(x > 0.0) {
      mesg = "positive.";
    } else if(x < 0.0) {
      mesg = "negative.";
    } else {
      mesg = "zero.";
    }
    System.out.println(x + " is " + mesg);
  }
}
```

このように、文字列は「`String`型」なわけだが、この型についてはもっと後で詳しくやるので今はこの程度で。

2 演習問題解説 — 繰り返し

2.1 演習4a〜4c

このあたりは簡単なのでさっさとJavaプログラムを示そう。2のN乗。

```
import java.io.*;

public class r2ex4a {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value n = ");
    int n = (new Integer(in.readLine())).intValue();
    int x = 1;
    for(int i = 0; i < n; ++i) { x = x * 2; }
    System.out.println("2^" + n + " = " + x);
  }
}
```

N の階乗。

```
import java.io.*;

public class r2ex4b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value n = ");
    int n = (new Integer(in.readLine())).intValue();
    int x = 1;
    for(int i = 1; i <= n; ++i) { x = x * i; }
    System.out.println(n + "! = " + x);
  }
}
```

組み合わせ。

```
import java.io.*;

public class r2ex4c {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value n = ");
    int n = (new Integer(in.readLine())).intValue();
    System.out.print("value r = ");
    int r = (new Integer(in.readLine())).intValue();
    int x = 1;
    for(int i = 1; i <= r; ++i) { x = x * (n-r+i) / i; }
    System.out.println(n + "C" + r + " = " + x);
  }
}
```

なお、組み合わせは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行かない。$\frac{4\times5\times6\times7}{1\times2\times3\times4}$ みたいに。この順序でやれば常に割算が割り切れるので誤差なしで計算できる。

2.2 演習 4d

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

```
import java.io.*;

public class r2ex4d {
  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();
    System.out.print("value n = ");
    int n = (new Integer(in.readLine())).intValue();
    double sign = 1.0, pow = 1.0, fact = 1.0, sin = 0.0, cos = 0.0;
    for(int i = 0; i < 2*n; i = i + 2) {
      cos = cos + sign * pow / fact;
      pow = pow * x; fact = fact * (i+1);
      sin = sin + sign * pow / fact;
```

```
      pow = pow * x; fact = fact * (i+2); sign = -sign;
    }
    System.out.println("x: " + x + "  sin: " + sin + " cos: " + cos);
  }
}
```

このプログラムでは、i 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く。一応、計算式を再録しておく。

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

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

ところで、「何項まで計算するか」によって精度が代わって来るが、言い替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る。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る。これを**打ち切り誤差**といい、既に学んだ**丸め誤差**、**情報落ち**、**桁落ち**とならんで数値計算における誤差の要因の 1 つとなる。では実際に計算してみよう。

```
% java r2ex4d
value x = 1.04719755112  ← π/3 (60 度)
value n = 5                                    ↓確かに 0.5
x: 1.04719755112  sin: 0.8660254450614825 cos: 0.5000004334992502
% java r2ex4d
value x = 3.14159265359  ← π
value n = 5                      ↓いまいち…
x: 3.14159265359  sin: 0.006925270707302825 cos: -0.9760222126235919
% java r2ex4d
value x = 3.14159265359   ←もう 1 度π
value n = 10              ← 数を増やすと OK
x: 3.14159265359  sin: -5.291255517616602E-10 cos: -1.0000000035290801
% java r2ex4d
value x = 314.159265359   ← 100 π
value n = 10                      ↓グチャグチャ…
x: 314.159265359  sin: -2.286912714525164E30 cos: -1.3836026211765178E29
%
```

何がいけないのだろう? 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。ではどうすれば? sin とか cos は周期関数なので、上の方法で計算するのは絶対値の小さいところ、つまり $0 \le x \le \frac{\pi}{4}$ だけにしておくのがよい (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るから)。あとついでに、上のプログラムではやっていなかったが、加えて行く順序をテイラー級数の後ろの項から順にした方がよい。というのは、後ろの方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するため。それは 5 項とか決めておけば、その順で計算式を書けばよい。

2.3 演習 5

課題には「考え方」だけ書かれていたが、疑似コードを示す。

- 整数 x、y を入力する。
- 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。
- x を出力する。

Java のコードは次の通り。

```
import java.io.*;

public class r2ex5 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

```
    System.out.print("x = ");
    int x = (new Integer(in.readLine())).intValue();
    System.out.print("y = ");
    int y = (new Integer(in.readLine())).intValue();
    while(x != y) {
      if(x > y) {
        x = x - y;
      } else {
        y = y - x;
      }
    }
    System.out.println("GCD : " + x);
  }
}
```

なぜこれで最大公約数が求まるのだろう? 次のように考えてみるとよい。なお、x と y は正の整数であるものとします。

- $x = y$ であれば、最大公約数は x そのもの。当然ですね。
- $x > y$ であれば、x と y の最大公約数は $x - y$ と y の最大公約数に等しい。[1]
- したがって、$x - y$ を改めて x とおいて、x と y の最大公約数を求めればよい。
- $x < y$ 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。

どうですか、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、という意味が分かります?

3 数値積分 (つづき)

演習解説の途中だけれど、ここからが数値積分の話の本番なのであらためて。前回、数値積分の考え方を説明して、しかし区間の左端や右端の $f(x)$ の値では誤差があることまで述べた。

で、演習問題として「左端と右端の平均を取ったら」というのが演習 3a だった。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に等しい。このため、これを数値積分の「台形公式」と呼ぶ (区間の幅を d で表す)。

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

Java プログラムを示しておく。

```
import java.io.*;

public class r2ex3a {
  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;
```

[1] 証明: 最大公約数を G とおくと、x も y も G の整数倍なのだから、$x - y$ もまた G の整数倍である。ということは、G は $x - y$ と y の公約数である (最大かどうかはまだ分からない)。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。ということは、H は $x - y + y = x$ の約数でもある。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する。従って G は $x - y$ と y の最大公約数でもある。

```
    for(int i = 0; i < n; ++i) {
      double x = a + i * (b - a) / n;
      double y0 = x * x, y1 = (x+dx) * (x+dx);
      s = s + 0.5 * (y0 + y1) * dx;
    }
    System.out.println("integral: " + s);
  }
}
```

台形公式は直線による補間なので、関数の 2 階微分が 0 でない場合、つまり上や下に凸な場合は誤差が出る。具体的には、上に凸だと少なく、下に凸だと多くなる。ところで演習の問題文にも書いたように、左端/右端の代わりに区間の中央の x を使って長方形で計算すると (これを**中点公式**という)、逆に上に凸だと多く、下に凸だと少なくなる (図 1)。だからこれをちょうどよく混ぜたらいいのでは、というのが演習 3c になっていた。

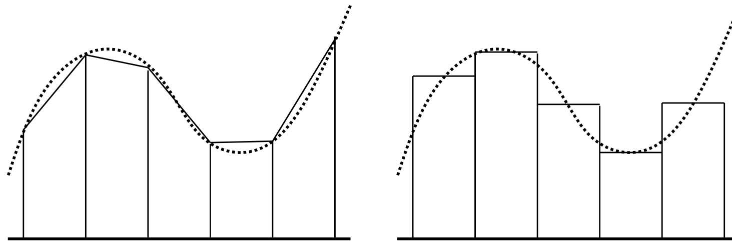

図 1: 台形公式と中点公式

実は、左端:中央:右端を 1:4:1 で混ぜると (つまり台形:中央を 1:2 で混ぜると) よい結果が得られる。そのプログラムも示しておく。

```
import java.io.*;

public class r2ex3c {
  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 y0 = x*x, y1 = (x+0.5*dx)*(x+0.5*dx), y2 = (x+dx)*(x+dx);
      s = s + (y0 + 4.0*y1 + y2) * dx / 6;
    }
    System.out.println("integral: " + s);
  }
}
```

実際にやってみよう:

```
% java r2ex3c
value a = 1
value b = 10
value n = 100
integral: 332.99999999999994  ←すごくいい!
% java r2ex3c
value a = 1
value b = 10
value n = 10
integral: 333.0  ←ぴったし…?
%
```

実は、この計算式は**シンプソンの公式**と言われ、数値積分では標準的な方法である (下の式では見やすくするため区間の半分を d としている、そのためプログラムの 6 で割る代わりに 3 で割っている):

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

なぜこれがいいのかというと、シンプソンの公式では当該区間を 2 次曲線で補間していることになるから。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) よかったわけである。

ではなぜ 2 次式の補間になるのか、その理由を説明する (やりたくないけど)。当該区間の曲線を 2 次式

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

で表せるとする。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 とおく。上の 2 次式の不定積分は $\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ だから、面積 (定積分) は

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

となる。これを整理すると

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

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 (そんなの見比べて分かるか?!)、

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

になる。というわけで、上の式が出て来るわけである。

さて、ではシンプソンの公式が一番いいのかというと、必ずしもそうとは言えない。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はとっておいて、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よい。

このような計算方法を**漸近的**という。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 つの方法である。

4 $f(x) = 0$ の求解

4.1 数え上げによる求解

こんどは、関数 $f(x)$ について、$f(x) = 0$ を満たす x を求めるという問題、つまり求解を取り上げる。これも解析的に解けなくても次のような条件があればプログラムで解を求めることができる:

- ある区間 $[a, b]$ において、$f(x)$ が単調増大、連続、かつ $f(a) < 0$、$f(b) > 0$ となるような a、b が分かっている

ひらたく言えば、a ではマイナス、そこからずっと増えていって、b ではプラスになっていて、途中で「飛んでいる」ところがないならその間のどこかに解があるからそれを求める、ということ。

たとえば、「$N(> 1)$ の平方根を求める」としよう。$f(x) = x^2 - N$ とすれば、$f(0) < 0$、$f(N) > 0$ だから上の条件を満たしているので、解を求めることができる (そしてそれが N の平方根なわけだ)。

では次の疑似コードを見てみよう:

- 数値 n を入力する。
- d ← n / 1000000。
- i を 0 から 1000000 の手前まで変えながら繰り返し、

- 　r ← i * d。
- 　もし $r^2 - n \geq 0$ なら、繰り返しを抜け出す。
- 繰り返し終わり。
- r を出力する。

つまり、$f(0) < 0$ なのだから、十分小さい d を用意し、$d, 2d, 3d, 4d, \cdots$ について順に $f(x)$ を計算し、最初に 0 以上になったところでやめれば精度 d で解が求まるわけだ。これを**数え上げ**による方法という。Java のコードは次の通り（「ループから抜け出す」のには **break 文**というものを使えばよい)。

```
import java.io.*;

public class R3Sample1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value x = ");
    double n = (new Double(in.readLine())).doubleValue();
    double r = 0.0, d = n / 100000;
    for(int i = 0; i < 100000; ++i) {
      r = i * d;
      if(r*r - n >= 0) { break; }
    }
    System.out.println("n: " + n + "  root: " + r);
  }
}
```

演習 1 このプログラムを打ち込んでそのまま動かせ。また、精度を上げたときに何桁くらいなら実用になるか試せ。(終わらなくて止めたいときは「Control+C」で中止。)

演習 2 もっとましな方法を実現してみる (説明は下記。どちらも、分割数 n は不要で、代わりに許容誤差 e を指定する。e は 0.000001 とか固定してもいいし、入力させてもよい)。

a. 区間 2 分法の考え方によって平方根を求めるようにしてみよ。

b. ニュートン法の考え方によって平方根を求めるようにしてみよ。

4.2 区間 2 分法

先の前提では、区間 $[a, b]$ おいて、$f(a) < 0$、$f(b) > 0$ であり、この区間中に根があるという前提だった。ところで、$c = \frac{a+b}{2}$ を求め、$f(c)$ を計算してみたらどうだろうか。もしも $f(c) < 0$ であれば、根がある範囲は区間 $[c, b]$ に狭められる。そうでなければ、根がある範囲は区間 $[a, c]$ に狭められる。つまり a か b のどちらかを c で置き換えられるわけだ。その後また同様に繰り返すことで、区間の幅を半分ずつにして行ける。$2^1 0$〜1000 だから、40 回も繰り返せば区間の幅を 10^{12} 分の 1 にできる、言い替えればその精度で解が求まったとも言えるわけだ。

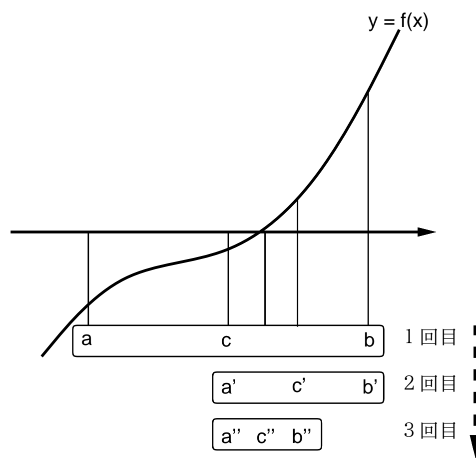

図 2: 区間 2 分法による求根

4.3 ニュートン法

ニュートン法はかの万有引力の発見者ニュートンに由来する方法で (彼は微積分学の発明者でもある)、適当な近似値 r から始めて、その近似値を改良していくことで解に到達する。具体的には、$f(x)$ の $x = r$ における接線を求め、接線と X 軸が交わる点の X 座標を新たな r とし、これを反復していく。これを r_i のような数列と考える。

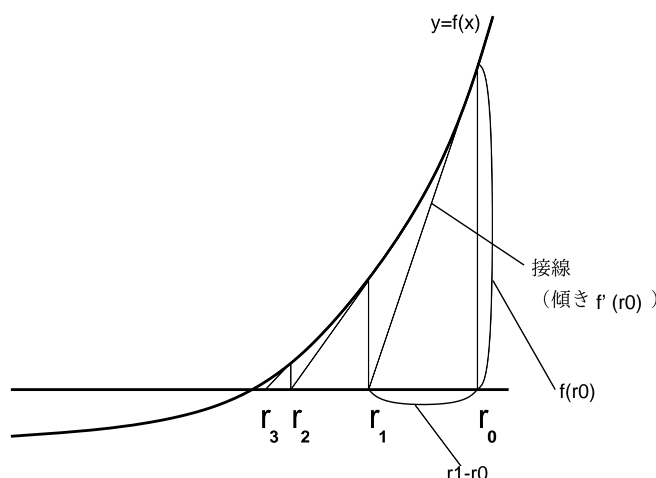

図 3: ニュートン法による求解

具体的に求めてみよう。$x = r_i$ のときの接点の座標は $(r_i, f(r_i))$、そこでの接線の傾きは $f'(r_i)$ (もちろん関数は微分可能でないといけません)。

$$\frac{f(r_i)}{r_{i+1} - r_i} = f'(r_i)$$

より、

$$r_{i+1} = r_i - \frac{f(r_i)}{f'(r_i)}$$

となる。いちおう親切までに $f(x) = x^2 - n$ の場合、$f'(x) = 2x$ より、

$$r_{i+1} = r_i - \frac{{r_i}^2 - n}{2r_i} = \frac{r_i}{2} + \frac{n}{2r_i}$$

一般にこのような、近似値を反復によって改良していく方法を**反復解法**と呼ぶ。反復の結果、値がほとんど変化しなくなったら**収束**したこととし、そこでの近似値を解とする。つまり収束条件は

$$\left| \frac{r_{i+1} - r_i}{r_i} \right| < \epsilon$$

しかし今回は平方根と分かっているので $|r^2 - n|$ で見てもよい。なお、絶対値は前回の絶対値のプログラムを書いてもいいけど面倒なので、よかったら `Math.abs(`x`)` を使ってください。

ニュートン法は収束すれば高速なことで知られるが、収束しないケースもある (その条件とかは難しいので説明しない、というか久野には説明できません `^_^;`)。ともかく、平方根の場合は近似値として n から始めれば問題ない。

5 さまざまなデータ型

5.1 基本データ型

＃1 でもやったように、コンピュータではさまざまなデータを 2 進表現して扱っている。もとのデータが何であるかによって、どのような 2 進表現を使うかを適宜選択する。実際には、プログラムを書くとき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する。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) と呼ぶ。

以下では Java 言語の場合を例にとり、主要なデータ型を見て行くことにする。そのうちでも、単一の (内部に構造を持たない — もちろんビットの列という意味での構造はあるけど、それ以上に込み入ったものはない) データ型のことを**基本データ型**という。Java では次のものがある:

- 整数のデータ型 — 2 進数、2 の補数表現
  - `int` — 整数型。32 ビットの 2 の補数表現。「`123`」
  - `long` — 倍長整数型。64 ビットの 2 の補数表現。「`123L`」
  - `short` —- 16 ビット整数型。16 ビットの 2 の補数表現。
  - `byte` — 8 ビット整数型。8 ビットの 2 の補数表現。
- 実数のデータ型 — 2 進数、浮動小数点
  - `double` — 64 ビット IEEE754 形式浮動小数点。「`3.14`」
  - `float` — 32 ビット IEEE754 形式浮動小数点。「`3.14f`」
- `char` — 文字型。16 ビットの符号なし 2 進数。「`'a'`」
- `boolean` — 論理型。`true`(はい)、`false`(いいえ) の 2 値。

なぜ文字が数のデータ型かというと、文字はコンピュータ内部では文字コード (文字の番号) として表現されているから。一方、論理型は「はい」「いいえ」の 2 値で、これは数ではない。比較演算の結果はこの型。

なお、Java では数のデータ型どうしでは、ビット数の少ないものから多いものへ、また整数から浮動小数点への変換は自動で行われる (逆向きはキャストが必要)。

5.2 構造を持ったデータ型

構造を持ったデータ型ないし**複合データ型**とは、その中に基本データ型を複数個含み得るような種類のデータ型をいう (図 4)。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 Java の場合を説明する。

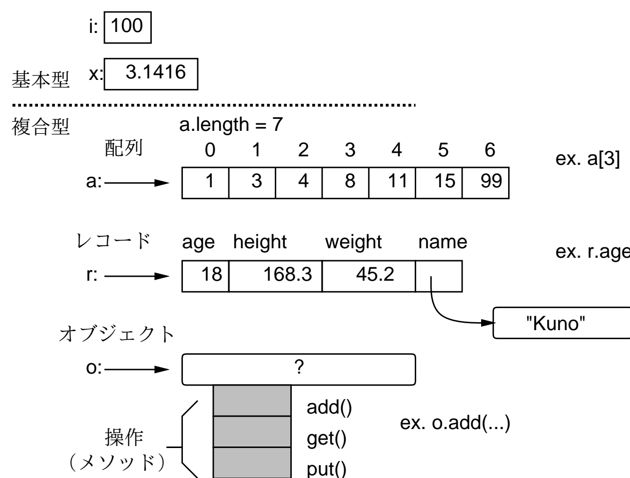

図 4: さまざまなデータ型

- **配列型** — 同じ型の値が並んだもの。数学の x_i(添字つき変数) みたいなもの。添字は「`[…]`」の中に書いて表す言語が多い。たとえば `a` が配列なら、`a[0]`、`a[i+1]` などのようにして個々の要素を指定する。
- **レコード型** — 複数の型の値を組みにしたもの。それぞれの (中に含まれている) 値を**フィールド**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。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、という風に使う。

- **オブジェクト型** — 内部的に (レコード型のように) データを保持しているが、外部からはさまざまな操作を呼び出して利用するようなもの。たとえば `in` というのが入力用のオブジェクトで、`in.readLine()` で 1 行読み込みその結果を文字列として返す。

Java は「基本データ型以外はすべてオブジェクト型」という設計の言語なので、レコード型はない (そのような機能はオブジェクト型に包含されている)。配列もオブジェクトの一種なのだが、ただし上で説明した添字による参照は配列だけに可能など、特別扱いの部分がある (次節で説明)。なお、先に出て来た `String` 型もオブジェクト型の一種である。

図 4 を見て不思議に思ったことはないだろうか。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るが、オブジェクト型 (配列等) では少しはな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る。実はこの矢印はデータのありかを示す**参照** (実体はメモリ上の番地だと思ってよい) である。で、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、フィールドにはそこの参照が入っている。

この「値と参照の区分」はまた後でもやるが、とりあえず「`a = b;`」のようにして変数間で代入をしたとき、基本型では値 (箱の中身) がコピーされるが、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) なので注意が必要であることは覚えておいて欲しい。

5.3 配列

上述のように「配列」とは、「同じ種類 (型) のデータを沢山ならべたもの」という意味である。並べる型は何でもよい。Java では配列の型は元の型の後ろに「`[]`」をつけて表す。たとえば「`int[]`」は整数の並んだ配列、「`String[]`」は文字列の並んだ配列ということになる。

配列を使うにはその配列型の変数を宣言した上で、大きさを指定して配列オブジェクトを用意しなければならない (他のオブジェクトの機能としてできあがった配列を返して貰う場合はそちらに任せればよいが)。具体的には次のような感じになる。

```
int[] a = new int[100];   ←要素数 100 の配列を用意
int a[] = new int[100];   ←変数定義ではこの書き方も可 (C 言語から伝承)
```

上述のように、配列も一種のオブジェクトである (ただし書き方がやや特別な形になっている)。そして、オブジェクトはレコードのようなものでもあり、たとえば配列の長さ (要素の数) は `a.length` で取得できる。上の例では `a.length` は 100 である。

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る。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する (これを添字と呼ぶ)。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る (例によって 0 番目から数えるのに注意)。

では、配列に実数をいくつか読み込み、その合計を表示するという例題を示そう。合計は積分とかで散々やったので簡単ですね。ただし、in.readLine() で読み込めるのは文字列なので、それをもとに配列のデータを作るところがちょっと新しい。

- data に文字列の配列を読み込む。
- a ← data と同じ要素数の実数配列。
- 変数 i を 0 から data.length の手前まで変えながら繰り返し、
-     a[i] ← data[i] を数値にしたもの。
- 繰り返し終わり。
- s ← 0。
- 変数 i を 0 から a.length の手前まで変えながら繰り返し、
-     s ← s + a[i]。
- 繰り返し終わり。
- s を出力する。

最初の「文字列の配列を読み込む」ところは、「`in.readLine()`」で 1 行読み込み、それを「`.split(" +")`」で空白の箇所で分割することで、文字列の配列が得られる。文字列から数値に変換するのは、その後別途行う (難しいですよね、またそのうち説明しますので)。Java コードは次の通り。

```
import java.io.*;

public class R3Sample2 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("data a0 a1 ... : ");
    String[] data = in.readLine().split(" +");
    double[] a = new double[data.length];
    for(int i = 0; i < data.length; ++i) {
      a[i] = (new Double(data[i])).doubleValue();
    }
    double s = 0.0;
    for(int i = 0; i < a.length; ++i) { s = s + a[i]; }
    System.out.println("total: " + s);
  }
}
```

一応、動かすところのようす:

```
% java R3Sample2
data a0 a1 ... : 3 5 7 9 20
total: 44.0
%
```

なお、配列では「添字が許される範囲を超える」とエラーになり止まるようになっているので、エラーが出たらよく見ること (その範囲外の値がいくつなのかがメッセージに含まれている)。

演習 3 上記のプログラム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 Java プログラムを作れ。

a. 配列に実数いくつかを読み込み、その最大値と最小値を出力する。
b. 配列に実数いくつかを読み込み、最大値が何番目か (複数あればそれら全部の番号) を出力する。なお先頭を 0 番目とする。
c. 配列に整数をいくつか読み込み、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。

演習 4 配列に実数をいくつか読み込み、それを「小さい順に並べて」出力するプログラムを作ってみよ。アルゴリズムの例としては、次のような方法が考えられる。

a. 単純選択法 — たとえば添字番号で言って 0〜9 番があったとする。まず、0〜9 番のうちで最大のものを選び、それを 0 番に入れる (これまで 0 番に入っていたものはその最大のものが入っていた場所に代わりに入れる)。つぎに 1〜9 番で最大のものを選び、それを 1 番に入れる (これまで 1 番に入っていたものはその最大のものが入っていた場所に代わりに入れる)。以下同様にしていく。
b. 比較交換法 — 隣接する 2 つの要素の大小を比べ、「大-小」の順で並んでいたらそれを交換して「小-大」の順にする。これを「それぞれの位置について十分な回数だけ」繰り返す。

これ以外の他の方法を自分で考えてもいいです。

演習 5 演習 4 のような方法で「並べ換え」をテストしていると、データの量が少ないので性能の差が分からない。そこで、入力する代りに「`Math.random()`」(区間 $[0, 1)$ の一様乱数生成) を使って十分大きな配列 (1 万とか 10 万とか) にデータを入れ、作成した整列アルゴリズムで所要時間を計測してみよ。また、もっと速くする工夫があれば試してみよ。

なお、正確な時間計測のためには、`System.currentTimeMillis()` を使うとよいでしょう。これは 1970 年 1 月 1 日午前 0 時から現在までの「ミリ秒数」を `long`(64 ビット整数) で返すので、次のようにして 2 回呼んで差を測ることで所要時間が計測できる。

```
long t1 = System.currentTimeMillis();
// 何か時間を測りたい処理
long t2 = System.currentTimeMillis();
int dt = (int)(t2 - t1);  // 所要時間 (ミリ秒単位)
```

最後に `int` にキャストしているのは、所要時間差 (ミリ秒単位) は 32 ビットに入るだろうから 32 ビット整数に入れるのが便利だが、64 ビットから 32 ビットにするのにはキャストが必要だから。

6 アルゴリズムの改良

6.1 演習 6 — 論理型

すごーくお待たせしたが、演習問題解説の続き。素数、やってみましたか? 疑似コードは次の通り。

- N を入力する。
- 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」
- 繰り返しおわり。
- sosu を出力。

この sosu は先に説明した真偽値 (真/偽のいずれかだけを表す) なので、Java では `boolean` 型を使い、また「真」は `true`、「偽」は `false` で表す。もっとも、皆様は習ってないから整数の 1/0 で表したりしたと思うけれど、それはそれで結構。なお、この変数 sosu は最初に「真」を入れておいて、どこかで素数でないと思ったら「偽」にして、最後に結果がどちらか見る。このような使い方の変数のことを「旗」(flag) と呼ぶ。では Java コードを見てみよう。

```
import java.io.*;

public class r2ex6 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("n = ");
    int n = (new Integer(in.readLine())).intValue();
    boolean sosu = true;
    for(int i = 2; i < n; ++i) {
      if((n % i) == 0) { sosu = false; }
    }
    System.out.println("prime number : " + sosu);
  }
}
```

前にも説明したが、「`%`」は剰余演算子 (割った余りを返す) なので「`(n % i) == 0`」で「`n` が `i` で割り切れる」という意味になる。

6.2 演習 7 — プログラムの改良

さて、演習 7 は演習 6 の部分を「ループの中に取り込んで」作ることができる。疑似コードは略して Java の方を見てみよう。

```
import java.io.*;

public class r2ex7 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("m = ");
    int m = (new Integer(in.readLine())).intValue();
    for(int n = 1; n <= m; ++n) {
      boolean sosu = true;
      for(int i = 2; i < n; ++i) {
        if((n % i) == 0) { sosu = false; }
      }
      if(sosu) { System.out.println(n); }
    }
  }
}
```

変数 sosu の使い方が「sosu だったらその数を打ち出す」に変っているのに注意。ところで、これでやると私の手元では 17000 までやるのに 10 秒くらい掛かっている。これを工夫して速くするにはどうしたらいいだろう? たとえば次のようなことが考えられる。

- 2 は別に打ち出すことにして、候補として 3 以上の奇数だけ調べる (候補の数が半分になる!)。

- 割ってみる数も 3 以上の奇数だけ試す (割ってみる数も半分に!)。
- $N-1$ まで試さなくても、$\sqrt{N}$ まで試せば十分 ($\sqrt{N}$ より大きい因数があるなら、必ず $\sqrt{N}$ より小さい因数もあるわけだから)。
- 割り切れると分かったところでループを止めてその先は調べないようにする。

これらの改良を施した版を次に示す。

```
import java.io.*;

public class r2ex71 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("m = ");
    int m = (new Integer(in.readLine())).intValue();
    System.out.println(2);
    for(int n = 3; n <= m; n += 2) {
      boolean sosu = true;
      int limit = (int)(Math.sqrt(n) + 1);
      for(int i = 3; i <= limit; i += 2) {
        if((n % i) == 0) { sosu = false; break; }
      }
      if(sosu) { System.out.println(n); }
    }
  }
}
```

break 文も今回出て来たので、使えなかった人はすみません (代わりにループ条件の変数にすぐ終わるような値を入れれば止められますね)。で、これだと、17000 までやるのに私の手元で約 1 秒だった。つまり、10 倍のスピードアップ! ところで、さらに次のような改良もあり得ますね?

- 3 からの奇数全部で割ってみる代りに、これまでに見つかった素数でだけ割ってみれば十分。

素数は数が大きくなるとかなりまばらにしかないので、これで割ってみる数が減らせる。ただし残念なことに、これまでに学んだやり方では「見つかった素数を覚えておいて利用する」ことができない! 実は、後で学ぶ「配列」を使うとこれができるので、ぜひこの改良にチャレンジしてみて欲しい。

演習 6 配列を使って「N 未満の素数を全部打ち出す」プログラムを「これまでに分かっている素数でだけ割って見る」ように改良し、どれくらい速くなったか調べよ。

演習 7 次のような構想 (これはまだ疑似コードではない!) に従って「N 未満の素数を全部打ち出す」プログラムを作り、速さを評価せよ。

- 論理値 (boolean) 型が並んだ要素数 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。

なお、`boolean` の配列は作った時点で自動的に「すべて偽」に初期化されることになっています。

A　本日の課題 3A

「演習 2」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 3A」とする。
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。

   Q1. 繰り返しを使ったプログラムに慣れましたか。

   Q2. 配列について学びましたが、使えそうですか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　次回までの課題 3B

次回までの課題は「演習 3」～「演習 7」の (小) 課題からプログラムを 2 つ以上作り、報告すること。レポートは授業開始時刻の **10 分前**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。

1. Subject: は「Report 3B」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. その簡単な説明。
5. もう 1 つのプログラムのソース。
6. その簡単な説明。あれば考察等も。
7. 下記のアンケートの回答。

Q1. 配列が使いこなせるようになりましたか。

Q2. 今回もまた、疑似コードを書くのと、Java に直すのと、打ち込んで動かすのとで掛かった手間の比率を教えてください。

Q3. 課題に対する感想と今後の要望をお書きください。