情報科学 2006　久野クラス　＃5

久野　靖*

2006.11.17

はじめに

3 週間ぶりのごぶさたでした。なんですが、来週も駒場祭のため休講なので次回までも 2 週間あきになりますね (でも駒場祭は楽しみたいでしょうから、その分だけ課題を多くするなんてことは言いませんのでご安心を)。今回は新しい内容として次のものをやります。

- オブジェクト入門、文字列オブジェクト
- 簡単なオブジェクト (クラス) を自分で定義する

整列 (ソート) の話は習ったことがないので詳しく聞きたいという人がいましたので、後で計算量の話の仕上げも兼ねる程度で説明していきます。

1　演習問題解説

1.1　演習 2: 再帰関数

これは簡単に。関数の部分だけ示せばいいですよね。まず階乗。

```
static int fact(int x) {
  if(x == 0)     { return 1; }
  else           { return x * fact(x - 1); }
}
```

次はフィボナッチ数。

```
static int fib(int x) {
  if(x == 0 || x == 1) { return 1; }
  else                 { return fib(x-1) + fib(x-2); }
}
```

次は組合せの数。

```
static int comb(int n, int r) {
  if(r == 0 || r == n) { return 1; }
  else                 { return comb(n-1, r) + comb(n-1, r-1); }
}
```

最後に 2 進表現。

```
static String binary(int n) {
  if(n == 0)          { return "0"; }
  else if(n == 1)     { return "1"; }
  else if(n % 2 == 0) { return binary(n / 2) + "0"; }
  else                { return binary(n / 2) + "1"; }
}
```

まあこのあたりは、再帰に慣れて頂くということで簡単でよかったんじゃないでしょうか。

*筑波大学大学院経営システム科学専攻

1.2 演習 6/7 — 素数の計算

素数については前回もやったが、それを配列を使ってもっと効率よく、という話だった。これもプログラムだけ示しておく。まず「見つかった素数を覚えておいてそれだけで割ってみる」方法。

```
import java.io.*;

public class r3ex6 {
  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();
    int[] a = new int[m];
    int count = 1; a[0] = 2; 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 = 1; i < count && a[i] <= limit; ++i) {
        if((n % a[i]) == 0) { sosu = false; break; }
      }
      if(sosu) { a[count] = n; ++count; System.out.println(n); }
    }
  }
}
```

配列 a そのものは大きめに用意しておき、「現在いくつ値が入っているか」を変数 count で覚えておく。つまり実際に素数が入っているのは a[0]〜a[count-1] までの範囲ということ。

もう 1 つの方法は「エラトステネスのふるい」と呼ばれるアルゴリズム。

```
import java.io.*;

public class r3ex7 {
  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();
    boolean[] sosu = new boolean[m+1];
    for(int n = 2; n <= m; ++n) { sosu[n] = true; }
    for(int n = 2; n <= m; ++n) {
      if(sosu[n]) {
        System.out.println(n);
        for(int j = 2*n; j < sosu.length; j = j + n) { sosu[j] = false; }
      }
    }
  }
}
```

配列 sosu を sosu[0]〜sosu[m] が使えるように大きさ m+1 で用意し、最初はすべて true(素数である) にしておく。次に、2 から始めて順に n 番目が「素数である」なら、それは素数だから打ち出すとともに、n の倍数 (2 倍、3 倍、…) はすべて false(素数でない) に変更する。

1.3 演習 4: さまざまな図形の生成

とりあえず、演習 4 の a と b についてだけ。1 つのプログラムでまとめて示そう。さまざまな図形 (図 1) をすべて手続きによって実現しているので、main() ではそれらを順次呼び出すだけ。

```
public class r4ex4ab {
  public static void main(String[] args) throws Exception {
    Pixel[][] buf = new Pixel[200][300];
    for(int i = 0; i < buf.length; ++i) {
      for(int j = 0; j < buf[0].length; ++j) { buf[i][j] = new Pixel(); }
    }
    fillCircle(buf, 150, 30, 60, 255, 100, 70, 0.0);
```

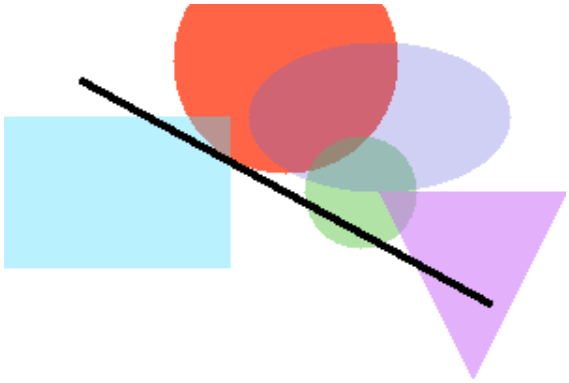

図 1: 生成されたさまざまな図形

```
    fillCircle(buf, 190, 100, 30, 100, 200, 80, 0.5);
    fillRect(buf, 60, 100, 80, 120, 80, 220, 255, 0.6);
    fillEllipse(buf, 200, 60, 70, 40, 100, 100, 220, 0.7);
    fillTriangle(buf, 200, 100, 300, 100, 250, 200, 200, 100,
    250, 0.5);
    fillLine(buf, 40, 40, 260, 160, 4, 0, 0, 0, 0.0);
    writeImage(buf);
  }
  static class Pixel { int r = 255, g = 255, b = 255; }
```

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。それを行う手続き `setColor()` を用意してあとはすべてこれを使うこととした。また、このとき指定した座標が配列の範囲を超えていたら「無視する」ことにした (縁に掛かった図形でも問題なく表示できるようにするため):

```
  static void setColor(Pixel[][] a, int x,
                       int y, int r, int g, int b, double alpha) {
    if(x < 0 || x >= a[0].length || y < 0 || y >= a.length) { return; }
    a[y][x].r = (int)(a[y][x].r * alpha + r * (1.0 - alpha));
    a[y][x].g = (int)(a[y][x].g * alpha + g * (1.0 - alpha));
    a[y][x].b = (int)(a[y][x].b * alpha + b * (1.0 - alpha));
  }
```

円を描く `fillCircle()` はこれを呼ぶように直した。あと、画面全体について調べるのでなく、XY 座標の最小〜最大の範囲内だけ調べるように直した (この方が大きい画面のときは効率がよいはず):

```
  static void fillCircle(Pixel[][] a, double x, double y, double rad,
                         int r, int g, int b, double alpha) {
    int i0 = (int)(y - rad), i1 = (int)(y + rad);
    int j0 = (int)(x - rad), j1 = (int)(x + rad);
    for(int i = i0; i <= i1; ++i) {
      for(int j = j0; j <= j1; ++j) {
        if((j-x)*(j-x)+(i-y)*(i-y) <= rad*rad) {
        setColor(a,j,i,r,g,b,alpha); }
      }
    }
  }
```

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 `setColor()` するだけ。回転させたい場合は後の「直線」を援用する。

```
  static void fillRect(Pixel[][] a, double x, double y, double w, double h,
                       int r, int g, int b, double alpha) {
    int i0 = (int)(y - 0.5*w), i1 = (int)(y + 0.5*w);
    int j0 = (int)(x - 0.5*h), j1 = (int)(x + 0.5*h);
    for(int i = i0; i <= i1; ++i) {
      for(int j = j0; j <= j1; ++j) { setColor(a,j,i,r,g,b,alpha); }
    }
  }
```

楕円を描く `fillEllipse()` は、円と同様でただし縦横に倍率を掛けてから半径 1 の円に入っているかどうかで判定すればよい:

```
static void fillEllipse(Pixel[][]a, double x, double y, double rx, double ry,
                        int r, int g, int b, double alpha) {
  int i0 = (int)(y - ry), i1 = (int)(y + ry);
  int j0 = (int)(x - rx), j1 = (int)(x + rx);
  for(int i = i0; i <= i1; ++i) {
    for(int j = j0; j <= j1; ++j) {
      if((j-x)*(j-x)/(rx*rx)+(i-y)*(i-y)/(ry*ry) <= 1.0) {
        setColor(a,j,i,r,g,b,alpha);
      }
    }
  }
}
```

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

```
static void fillTriangle(Pixel[][] a, double x0, double y0,
                         double x1, double y1, double x2, double y2,
                         int r, int g, int b, double alpha) {
  fillConvex(a, new double[]{x0,x1,x2,x0}, new double[]{y0,y1,y2,y0},
             r, g, b, alpha);
}
```

fillConvex() だが、各点についてそれが図形の内側にあるなら塗る。図形の内側にあるかどうかは isInside() で判定する。

```
static void fillConvex(Pixel[][] a, double[] x, double[] y,
                       int r, int g, int b, double alpha) {
  int xmin = (int)x[0], xmax = (int)x[0], ymin = (int)y[0], ymax = (int)y[0];
  for(int i = 1; i < x.length-1; ++i) {
    xmin = Math.min(xmin, (int)x[i]); xmax = Math.max(xmax, (int)x[i]);
    ymin = Math.min(ymin, (int)y[i]); ymax = Math.max(ymax, (int)y[i]);
  }
  for(int i = ymin; i <= ymax; ++i) {
    for(int j = xmin; j <= xmax; ++j) {
      if(isInside(j, i, x, y)) { setColor(a,j,i,r,g,b,alpha); }
    }
  }
}
```

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、と判断する。右側にあるかどうかは、辺の線分のベクトルと、線分の起点から調べるべき点までのベクトルの外積を計算して、負なら右側と判定する (このために左回りで周囲を指定するという条件が必要になっていた):

```
static boolean isInside(double x0, double y0, double[] x, double[] y) {
  for(int i = 0; i < x.length-1; ++i) {
    if(oprod(x[i+1]-x[i],y[i+1]-y[i],x0-x[i],y0-y[i])<0.0) {
    return false; }
  }
  return true;
}
static double oprod(double a, double b, double c, double d) {
  return a*d - b*c;
}
```

線を描く fillLine() だが、2 点の XY 座標と「線の幅」を指定する。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する。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るので、それを fillConvex() で塗ればよい:

```
static void fillLine(Pixel[][] a, double x0, double y0, double x1, double y1,
                     double w, int r, int g, int b, double alpha) {
  double dx = y1-y0, dy = x0-x1, n = 0.5 * w / Math.sqrt(dx*dx + dy*dy);
  dx = dx * n; dy = dy * n;
  fillConvex(a, new double[]{x0-dx, x0+dx, x1+dx, x1-dx, x0-dx},
             new double[]{y0-dy, y0+dy, y1+dy, y1-dy, y0-dy},
             r, g, b, alpha);
}
```

バッファを書き出す writeImage() は前回と変更なし。

```
  static void writeImage(Pixel[][] a) {
    System.out.println("P6 " + a[0].length + " " + a.length + " 255");
    for(int i = 0; i < a.length; ++i) {
      for(int j = 0; j < a[0].length; ++j) {
        System.out.write((byte)a[i][j].r);
        System.out.write((byte)a[i][j].g);
        System.out.write((byte)a[i][j].b);
      }
    }
    System.out.println();
  }
}
```

なかなか大変だったけど、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組み立てて行けることが納得できますね?

あと整列の計算量があるけど、それを先に喋っていると今回の演習をやる時間がなくていまいちなので、その話は後回しにする (レクチャーを聞いてもらうだけだから)。

2 オブジェクト

2.1 オブジェクト/抽象データ型/API

データ型とは、データの「種類」のことだった。また、データ型は基本データ型 (数値、論理値、文字など) と構造を持ったデータ型 (複合データ型) に分けられ、構造を持ったデータ型には、配列、レコード、オブジェクトなどがあることも学んだ。これまで 2 回で配列とレコードについてだいぶ分かったと思うので、今回はいよいよ**オブジェクト**を取り上げることにする。

オブジェクトについてはこれまで、「内部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」と説明してきた。もっと具体的に言うと、内部的には配列やレコードのように (どちらかというとレコードの方が近い)、複数の値を保持しているのだが (この内部で値を保持する変数を**インスタンス変数**と呼ぶ)、一般的にはそれを外部から直接読み書きせず、外部からは**操作** (**メソッド**) を呼び出すことで必要な処理をすべて行う (図 2)。

たとえば、これまでの例題では `in` という変数に `BufferedReader` オブジェクトを格納して扱っていたが、このオブジェクトの中身を直接参照することはなく、常に `in.readLine()` という形で操作 (メソッド) を呼び出し、その結果として次の入力行が返される、という形で利用してきたわけだ。このような、内部がどうなっているかは知らされず、外部から操作を呼び出すだけで扱うようなデータ型のことを**抽象データ型** (Abstract Data Types) と呼ぶ。

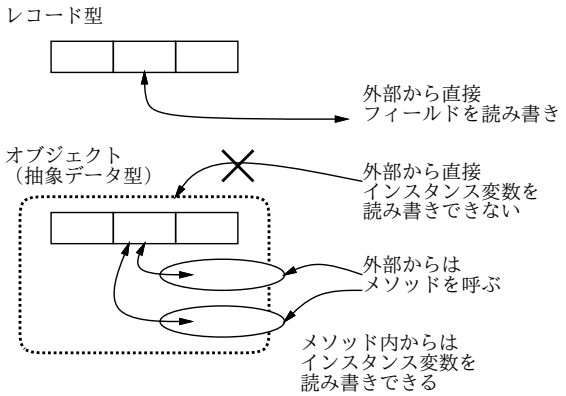

図 2: オブジェクト (抽象データ型)

オブジェクト指向言語のオブジェクトは、抽象データ型として扱えるような機能を提供するデータ型だと言える (それ以外にさまざまなプラスアルファもあるが、それは機会があれば説明していく)。

Java ではすべてのオブジェクトはクラスに対応している。つまり、そのオブジェクトの構造や機能の定義をまとめたものがクラスということになる。あるクラスについて、そのクラスによって定義されているオブジェクトのことをその

クラスの**インスタンス** (実体) と呼ぶ。クラスが「型枠」で、その型枠を使って同じような形のモノ (実体) を沢山つくり出したものがインスタンスだと思うとよいだろう (図 3)。

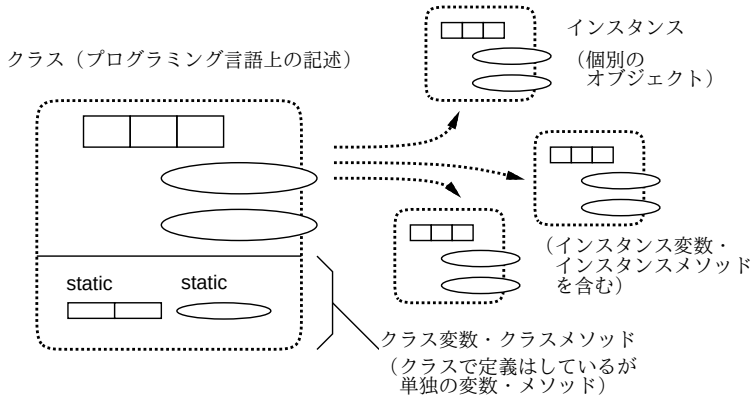

図 3: クラスとインスタンス

加えて、クラスは単独の変数や単独のメソッド (手続き) を置くのにも使える (図 3 下)。たとえばこれまで使って来た `Math.sqrt()` とか `Math.max()` などはクラス `Math` で定義されている単独のメソッドである。そして、Java では単独の変数やメソッドはキーワード `static` をつけて表す。ようやく、`static` というオマジナイの謎が少し解けましたね!

それで、どんなクラスにどんな変数やメソッドがあるのだろうか? それは Java の場合、**API ドキュメント**と呼ばれる文書にすべて記述されている。具体的には、本クラスの WWW ページから「JDK 1.5 日本語ドキュメント」の横にある「API ドキュメント」のリンクをたどると API ドキュメント が見られる。

API ドキュメントは「パッケージ」と呼ばれる単位ごとに分かれている。たとえば `java.lang.Math`(というのは `java.lang` パッケージに入っている `Math` というクラスを表す) のドキュメントを見ると、その中に `sqrt()` というメソッドがある。これが `Math.sqrt()` なわけである。そして冒頭にある `static` が、単独のメソッドを表している。

これまで散々お世話になっている**文字列** (`String`) も、オブジェクトの 1 種である。ただし特殊な性質として、文字列オブジェクトは「`"..."`」のように書くことでいきなりオブジェクトを作りだせるが、他のオブジェクトではそのような便利なものはない。

演習 1 API ドキュメント中の `java.lang.Math` の「メソッドの概要」の箇所を眺め、三角関数を計算するメソッドを探してみよ。また、`java.lang.Double` の定数の箇所を眺め、「-∞」の他にどのような定数があるか見てみよ。

演習 2 API ドキュメント中の `java.lang.String` の「メソッドの概要」の箇所を眺め、文字列に対して次のような操作を行なうメソッドの使い方 (名前とパラメタ) を手元にメモせよ (演習で役に立つ)。

a. i 文字目の文字を取り出す。
b. 文字列と別の文字列 (オブジェクト) が等しいかどうか調べる。
c. 文字列の長さ (文字がいくつ入っているか) を調べる。
d. 文字列の中で指定した文字/文字列が最初に現われる位置を得る。
e. 文字列中のある文字を別の文字に取り換えた文字列を得る。
f. 文字列中の i 文字目から後の部分の文字列を得る。
g. 文字列中の i 文字目から n 文字ぶんの文字列を得る。
h. 文字列中の小 (大) 文字を大 (小) 文字に取り換えた文字列を得る。

なお、「文字」と「文字列」は違うことに注意。「文字」は値の一種であり、型は「`char`」、定数は「`'a'`」などのように「`'`」で囲んで、文字 1 つだけを書く。「文字列」はオブジェクトであり、型/クラスは「`String`」、定数は「`"Abc"`」などのように「`"`」で囲んで、文字は 0 個以上何個でも書ける。

2.2 文字列オブジェクト

これからしばらく文字列オブジェクトを扱って行くが、「いきなり API ドキュメントを見ろと言われても見かたがわからん」という人が多いだろうから、少し解説を書いておく。API ドキュメントはとりあえず、クラス (=オブジェクトの種類) ごとに、そのオブジェクトがどのようなメソッドを持っているかを調べるのに使うことが多いはずである。

前節で説明したように、メソッドには、クラスメソッド (=単独のメソッド) とインスタンスメソッド (インスタンスに付属するメソッド) があり、その有無は API ドキュメントにおいても、「static」という指定がついているか否かで区別できる。

```
クラス名. メソッド (...) ← クラスメソッド (static と記されている)
式. メソッド (...)       ← インスタンスメソッド (static がない)
```

今後はインスタンスメソッドを使うことが多いと思うので、その場合についてもう少し説明する。インスタンスメソッドは「このオブジェクトについて〜する」という形の機能を表している。たとえば、文字列の中から指定した番号の文字を取り出すメソッド charAt() はインスタンスメソッドなので、必ず文字列オブジェクトを指定して呼び出す (配列と同様、先頭が 0 番なのに注意)。

```
"abcd".charAt(2) → 「'c'」
"XYZ".charAt(2) → 「'Z'」
```

つまり、同じ charAt(2) でも「どのオブジェクトに」対して呼び出すかによって結果が違う。それは、「お名前は」と尋ねて返って来る返事が相手によって違うのと同様だと思っていただければよい。従って、上の「式」というのはその「どのオブジェクト」を指定するわけだ。

ここで「式」がオブジェクトを表すのはいいとして、さらに「そのオブジェクトのクラスが指定したメソッドを持っている」ことも必要である。たとえば「"abcd"」という式は String のインスタンスを表すので、

```
"abcd".replace('a', '*')
```

は許されるが、「1」という式は int 型であり、これはクラス型ではないので (つまりインスタンスではなく値)、

```
1.replace('a', '*')
```

は許されない。また、たとえインスタンスであっても

```
System.out.replace('a', '*')
```

というのは、許されない。なぜなら、System.out は PrintStream クラスのインスタンスであって、PrintStream クラスは replace() というメソッドを用意していないから。たとえば犬に「おすわり!」と言えば分かるけど猫に「おすわり!」といっても分からないのと同様…おわかりかな?

さて、この replace() の API ドキュメントを見ると次のように書いてある。

```
String replace(char oldChar, char newChar)
```

これの意味は次の通り。

- まずこのメソッドは String クラスのところに書かれていて、static と書かれていないので、String オブジェクトに対して呼び出すインスタンスメソッド。
- String — このメソッドは String(文字列) を返す。
- replace(…) — もちろん、メソッドの名前。
- char oldChar — 最初の引数はかりに oldChar と呼ぶことにするが、これは文字型である。
- char newChar — 2 番目の引数はかりに newChar と呼ぶことにするが、これは文字型である。

つまり oldChar とか newChar という名前は説明の都合上つけてあるわけで、重要ではない (ちょうど自分のプログラムで好きな変数名をつけて構わないのと同じこと)。また、char というのは「文字ですよ」という情報を表しているだけであり、呼び出す時に本当に「char」と書いてはいけない (そうじゃなく、文字の値を渡す)。以上を (説明の文章とつき合わせて) 総合すると、replace() というメソッドは

```
"abcd".replace('a', '*') --- 文字「a」を「*」に置き換える
```

のように使うと判るわけである。まあ最初は慣れなくてしんどいと思うけど、API ドキュメントは調べられないといろいろ困るので気長に練習してください。

さて、お話ばかりではつまらないので、とりあえず上で調べた String クラスを題材にちょっと遊んでみよう。まずは「左右ひっくり返し」の例題を見てみよう。疑似コードを示す。

- 無限に繰り返し:
- 　文字列 str を入力。
- 　str が空文字列なら、ループを抜け出す。
- 　res ← 空文字列。
- 　i を 0 から長さ-1 まで変えながら繰り返し:
- 　　res の先頭に str の i 番目の文字を連結。
- 　ここまで繰り返し。
- 　res を出力。
- ここまで繰り返し。

先にも述べたように、計算機では数は 0 から数えることが多いので、文字列も長さ L の文字列であれば、その中には 0 番目、1 番目、…、$L-1$ 番目の文字が含まれると考える。Java のコードは次の通り。

```
import java.io.*;

public class R5Sample1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    while(true) {
      System.out.print("String> ");
      String str = in.readLine();
      if(str.equals("")) { break; }
      String res = "";
      for(int i = 0; i < str.length(); ++i) {
        res = str.charAt(i) + res;
      }
      System.out.println(res);
    }
  }
}
```

では説明を。

- 1 回入力したらおしまいではつまらないので、このプログラムは空行 ([RET] のみ) を入力するまで繰り返し処理を行う。
- 「`while(true) ...`　」で、条件が「真」という定数なので無限に繰り返すことになる。
- 文字列はオブジェクトなので「`==`」では比較できない。メソッド `equals()` を使うこと。ここでは空文字列と比較している。空文字列だったら前回学んだ `break;` を使ってループから抜ける。
- 文字列の長さはメソッド `length()` で調べられる。

では動かしてみよう。

```
% javac R5Sample1.java
% java R5Sample1
String> hello
olleh
String> aho
oha
String>    ← [RET] だけを打った
%
```

なかなか面白いでしょう?

演習 3 上の例題をそのまま打ち込んで動かせ。

演習 4 次の Java プログラムを作れ。`String` クラスのメソッドを活用すること。

a. 入力文字列をすべて大文字にして表示する。(ヒント: `toUpperCase()` を使う)

b. 入力文字列に現われる「`a`」をすべて「`*`」にする。(ヒント: `replace()` を使う)

c. 入力文字列に現われる「`a`、`e`、`i`、`o`、`u`」をすべて「`*`」にする。(ヒント: `replace()` を 5 回使う)

d〜f. 次のような三角形や巡回を表示する。(ヒント: `substring()` を使う)

```
d: String> abcd  e: String> abcd  f: String> abcd
   abcd             abcd             bcda
   abc               bcd             cdab
   ab                 cd             dabc
   a                   d             abcd
```

2.3 コマンド引数

これまで、プログラムにデータを与える時は BufferedReader オブジェクトから 1 行ずつ読み込む方法だけを使って来た。しかし、「ちょっとだけ」データを与える時にはプログラムを起動する java コマンドの行で一緒に指定する、という方法がある。具体的には、これまでは

```
java クラス名
```

でプログラムを起動しますよ、ということだけ説明してあったが、

```
java クラス名 文字列 文字列 … 文字列
```

のようにデータを書くことでそれがそのまま渡せる。なお、この「文字列」の部分はデータなので「"..."」や「'...'」で囲んでもよいが、中に空白文字などが入っていなければ何も囲まないままでもよい。

さて、この渡したデータを利用するには? それは、プログラムの冒頭に

```
public static void main(String[] args) ...
```

というところがあったが、実はこの「args」というのが (そのすぐ前に書かれているように)「文字列の配列」であり、そこに上のようにして渡したデータが入った状態でプログラムが実行開始される。簡単な例題を示そう。

```
import java.io.*;

public class R5Sample2 {
  public static void main(String[] args) throws Exception {
    for(int i = args.length-1; i >= 0; --i) {
      System.out.println(args[i]);
    }
  }
}
```

このプログラムはもうほとんど説明の余地はないというか、実行例だけ見ていただこう。

```
% java R5Sample2 This is a pen.
pen.
a
is
This
```

演習 5 文字列を扱う次のようなプログラムを作れ。ただし入力はコマンド引数で指定するものとする。

a. 文字列 s の大文字を小文字に、小文字を大文字にそれぞれ入れ替えた文字列をうち出す。(例: 「java XXX This」→「tHIS」)

b. 文字列 s が「回文」(前から読んでも後ろから読んでも同じ) かどうか判定する。回文の定義はまかせる。(例: 「java ABBA」→「true」、「java XXX SAY」→「false」)

c. 文字列 s の中に部分文字列として回文が含まれているかどうか調べ、それらをすべて打ち出す。さすがに 1 文字や 2 文字はつまらないので、回文の長さは 3 文字以上であるものとする。(例:「java XXX abassaba」→「aba、assa、aba、abassaba」) (ヒント: 文字列の長さを L として、長さ 3〜L のすべての部分文字列を取り出してそれが回文かどうか判定すればよい。)

d. 文字列 s と文字列 t について s の中に t が (連続した) 部分文字列として含まれているかどうか判定する。(例: 「java XXX akasaka asa」→「true」、「java XXX akasaka sasa」→「false」)(ヒント: t の長さを L として、s から長さ L の部分文字列をすべて取り出して t と equals() かどうか調べる。)

e. 文字列 s と文字列 t を読み込み、s の中に t が (とびとびにでも) (その順番で) 含まれているかどうか判定する。(ヒント: t の先頭の文字が s の中で最初に現れる場所を調べる。なければ NO。あれば、t の残りの文字列が s の残りの文字列の中にとびとびに現れるかどうか調べればよいので、再帰が使える。堂々めぐりにならないためにどうするかは考えてみること。)

f. 文字列 s と文字列 t を読み込み、s と t がアナグラムの関係にある (s の文字の並び順を交換したものが t になっている) かどうか判定する。(ヒント: t の 1 文字目が s の中にあるかを調べる。なければ NO。あれば、その文字を s から取り除いてしまった文字列について、それが t の 2 文字以降の文字列のアナグラムになっているかどうか調べればよいので、再帰が使える。堂々めぐりにならないためにどうするかは考えてみること。)

3 クラス定義

3.1 Java の文法

Java でだいぶ色々書いてもらったが、まだ全体の文法を示していなかった。以下に拡張 BNF と呼ばれる記法による、Java の文法 (簡略化してあります) を示す:

```
プログラム = クラス定義 …
クラス定義 = 修飾… class クラス名 { 定義… }
修飾 = public | private | static | …(以下略)
定義 = クラス定義 | 変数定義 | メソッド定義 | コンストラクタ
変数定義 = 型指定 ( 変数名 [ = 式 ] ),..  ;
型指定 = 型名 | 型指定 []
メソッド定義 = 修飾… 型指定 メソッド名 ( 引数,..  )  ブロック
コンストラクタ = 修飾… クラス名 ( 引数,..  )  ブロック
引数 = 型指定 引数名
文 = ; | 式 ; | ブロック | if 文 | while 文 | …(以下略)
ブロック = { ( 変数定義 | 文 )… }
if 文 = if ( 式 ) 文 [ else 文 ]
while 文 = while ( 式 ) 文
式 = 定数 | 呼び出し | 配列 | 演算 | ( 式 )
呼び出し = ( 式 | クラス名 ) .  メソッド名 ( 式,..  )
配列 = 式 [ 式 ]
演算 = 式 演算子 式 | 式 演算子 | 演算子 式
```

すべての Java プログラムはこの構文規則に従っていなければならない。構文エラーの場合は、この書き方に合っていない箇所がある、ということになる。拡張 BNF の書き方について少し説明しておく。

- α = β — 左の内容を右辺で定義。
- α | β — α または β。
- [α] — α があってもなくてもよい。
- α… — α の繰り返し。
- α ,.. — α をカンマで区切った並び。

ちなみに **BNF**(Backus Naur Form、または Backus Normal Form) とはプログラミング言語などの規則性のあるものの「規則」を表現するために考案された記法であり、本来の BNF では上記のうち冒頭の 2 つ (「定義」と「または」) だけを使っていた。しかしそれだけだと規則が長くなって読みづらいのでいくつか書き方を拡張したものを必要に応じて定義することが多い。これを拡張 BNF と呼ぶわけだ。そして、これらの拡張なくても元の BNF だけでも同じものが書き表せる。たとえば上の記法の残り 3 つについては次の通り (「ε」は「空」をあらわす):

$$\alpha_{opt} = \alpha \mid \varepsilon$$
$$\alpha_{list} = \alpha \ \ \alpha_{list} \mid \varepsilon$$
$$\alpha_{commalist} = \alpha \ \ , \ \ \alpha_{commalist} \mid \alpha$$

3.2 自前のオブジェクト定義

前にレコードの作り方をやったが、Java ではレコードをクラスで表すので、あれで半分は学んだことになる。オブジェクトを定義するのにクラスを使う場合は、さらに**メソッド**を追加すればよい。例として、有理数型オブジェクト ($\frac{a}{b}$ をそのまま扱うオブジェクト) を自前で作ってみよう。その定義は後で見るとして、とりあえずそれを使うメインから:

```
import java.io.*;

public class R5Sample3 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a1 = ");
    int a1 = new Integer(in.readLine()).intValue();
    System.out.print("b1 = ");
    int b1 = new Integer(in.readLine()).intValue();
    System.out.print("a2 = ");
    int a2 = new Integer(in.readLine()).intValue();
    System.out.print("b2 = ");
    int b2 = new Integer(in.readLine()).intValue();
    Rational r1 = new Rational(a1, b1);
    Rational r2 = new Rational(a2, b2);
    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
  }
```

この例題では、2 つの有理数を足してその結果を表示している。このように有理数オブジェクトは 2 つの値 a、b を指定して `new` で作り出すものとした。

では、これを実現するクラスを示す。変数 `a`、`b` がそれぞれ分子と分母を表す整数である。その次に `Rational()` という名前のメソッド「のようなもの」があるが、これは**コンストラクタ**と言い、オブジェクトを初期化するための専用のメソッドである。ここでは 1 引数と 2 引数の 2 つのものを用意している (Java では引数の個数や型で区別がつけられる限り、1 つの名前でいくつでもメソッドを定義してよい)。1 引数の場合は分母を 1、分子を渡された数にするので簡単だが、2 引数の場合は (1) 分母が 0 なら分子は 1、(2) 分母は 0 でないなら常に正、(3) 分子と分母は互いに素、という条件を満たすようにしているのでちょっと厄介である。

```
  static class Rational {
    private int a, b;
    public Rational(int x) { a = x; b = 1; }
    public Rational(int x, int y) {
      if(y == 0) { a = 1; b = 0; return; }
      if(y < 0)  { x = -x; y = -y; }
      if(x == 0) {
        a = 0; b = 1;
      } else if(x > 0) {
        a = x / gcd(x,y); b = y / gcd(x,y);
      } else {
        a = x / gcd(-x,y); b = y / gcd(-x,y);
      }
    }
    public Rational add(Rational r) {
      return new Rational(a*r.b + r.a*b, r.b*b);
    }
    public String toString() {
      if(b == 0) { return "Nan"; } else { return a + "/" + b; }
    }
    private static int gcd(int x, int y) {
      while(x != y) {
        if(x > y) { x = x - y; } else { y = y - x; }
      }
      return x;
    }
  }
}
```

メソッドやコンストラクタや変数の冒頭に `public` とついている場合、それはそのクラスの外部から参照できることを意味する。また `private` とついている場合、それはそのクラスの中からのみ参照できる。何もつけないと「同じファイル (正確にはパッケージ) 中から参照できる」になるが、はっきりさせるため以下では必ず `public` か `private` を指定することにする。

ここではコンストラクタと、足し算をするメソッド add、文字列に変換するメソッド toString() については、外部から呼び出してよいものとした。一方、通分に必要な最大公約数のメソッド gcd() はクラス内からだけ呼び出すこととした。そして gcd() は特定のオブジェクトに付随していないので static も指定してある。足し算 add については、分母を両方の分母の積にして通分して足し、約分はコンストラクタに任せるようにした。メソッド toString() は分母と分子を間に「/」をはさんで連結した文字列を返すだけである (ただし分母が 0 のときは"Nan"を返す)。なお、Java では文字列とオブジェクトを連結するなどで文字列が必要になったときは、このメソッドを呼んでオブジェクトを文字列に変換してくれる。このおかげで、main() では単に計算結果を文字列と連結するだけで表示が行えたわけである。

3.3 再び抽象データ型について

さて、ここまででクラス定義の作り方について学んだが、なぜこんな面倒なことをしているのだろうか? 分数を扱うのなら分子と分母の 2 つの値を整数型の変数に入れて計算すればいいのではないだろうか? これに対する反論が思いつきますか? 少し考えて見ていただきたい。

確かに上の Rational 型でもやっていることは 2 つの整数型変数に値を保持するだけだった。ただし違うのは、**2 つの変数とそれを扱う手続き (メソッド) が 1 つにまとめられていた**という点である。これによって、「r1.add(r2)」のようにコンパクトな形で操作を呼び出せる。

しかしもっと大事なのは、内部の 2 つの整数変数に対する制約をコントロールできるということ。具体的には、2 つの整数は常に次のような条件を満たすようになっていた。

- 分母は 0 または正である。0 のときは分子は 1 で、全体の値として NaN を表す。
- 常に約分された形になっている。整数の場合は分母は 1。

これらの条件を常に満たすことで、データの扱いが容易になる。もしも単に 2 つの変数を自前で扱うようにすると、その値はどこでも書き換えられるので、このような条件を維持するのは簡単ではない。これに対し、クラスの private 変数の値は同じクラスのメソッドを通じてのみ変更できるので、これらのメソッドをチェックすれば条件が維持できているかどうかは確認できる。これが抽象データ型の利点なわけである。

演習 6 上の例題を打ち込んで動かせ。うまく動いたら、減算、乗算、除算のメソッド (名前は sub()、mul()、div() とする) も作り、それらの演算も行ってみせるように直せ。

演習 7 復素数の値を表すクラス Complex(どのようなメソッドを用意するかは自分で決めてよい) と、その演算を使って計算を行う main() を作れ。

演習 8 int 型や long 型のビット数で収まらない大きい値を計算したければ、配列を使ってその要素に各桁の値を格納すればよい。配列の大きさが 10000 あれば、その各要素に 0〜9 を格納して計算を実行することで、10000 桁の計算ができる。このようなクラス Bigint を作り、それを利用して次のように 100 の階乗を求めてみよ。

```
Bigint n = new Bigint(1); // 整数型の値を渡して Bigint を作る
for(int i = 2; i <= 100; ++i) {
  n = n.mul(new Bigint(i));  // 順次掛けて階乗を作る
}
System.out.println("fact(100) = " + n); // toString() が使われる
```

ちなみに、100 の階乗の値は正しく計算すれば次の値になるはず:

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

なお、配列の各要素に 0〜9 ではなく整数で入り切る切りのいい範囲である 0〜9,999,999,999 を格納する方法 (百億進法) などを使ってもよい。配列も固定サイズでなく必要に応じて大きさを決めるようにしてもよい (この場合、下の桁から順に格納する方がやりやすいと思う)。

4　さまざまなアルゴリズムの計算量

4.1　復習: $O(N^2)$ と $O(N \log N)$ の整列アルゴリズム

では整列アルゴリズムの時間計算量の話題の続きをやることにしよう。しかし 3 週間たっているので、まずは前回説明した整列アルゴリズム ($O(N^2)$ と $O(N \log N)$ のもの 2 つずつ) がどんなものだったかの復習から (コードは再掲しないので前回資料を参照のこと)。

単純選択法 ($O(N^2)$)

単純選択法とは、まず 0〜N-1 番の要素から一番小さいもののある場所を見つけ、その一番小さいものを 0 番の位置に持って来て (それには 0 番の位置にあったものを一番小さいものがあった場所に移す、つまり交換する)、次に 1〜N-1 番から一番小さいもの (つまり全体としては 2 番目に小さいもののある場所を見つけ、それを 1 番の位置に、…のように続けていって全体を整列する (図 4)

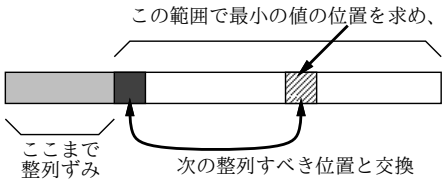

図 4: 単純選択法による整列

このアルゴリズムでは、最初に N 個、次に N-1 個、次に N-2 個、…の要素から最小を探すので、比較回数は $\frac{N(N+1)}{2}$ 回となり、低い次数を無視すると $O(N^2)$ の時間計算量になる。

バブルソート ($O(N^2)$)

バブルソートとは、端から順に隣接する 2 要素を比較し、逆順に並んでいたら交換する (図 5) ことを繰り返す。いつまで繰り返すかというと、もはや交換が起きなくなる (つまり完全に並び終わる) まで。

その手間は何回全体を操作するかによるが、その回数は運がよければ 1 回 (最初から全部並んでいた場合)、悪ければ N-1 回 (完全に逆順に並んでいた場合)、平均して $\frac{N}{2}$ 回と考えると、比較交換は 1 回につき N-1 必要だから全体として $\frac{N(N-1)}{2}$ となり、やはり低い次数を無視すると $O(N^2)$ の時間計算量になる。

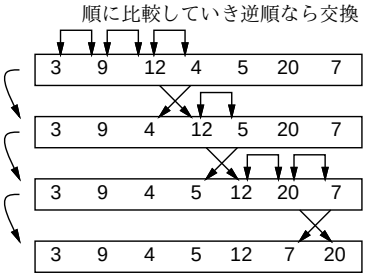

図 5: バブルソートによる整列

マージソート ($O(N \log N)$)

マージソートでは、再帰手続きを活用する。まず整列すべき列の長さが 1 なら何もせず、2 なら比較して逆順のとき交換を行うだけでよい。長さが 3 以上のときは、まず列を 2 つに分けて、それぞれを自分自身を再帰的に呼ぶことで整列する。続いて、整列し終わった 2 つの列をマージ (併合) して 1 つの整列ずみの列にする (図 6)。

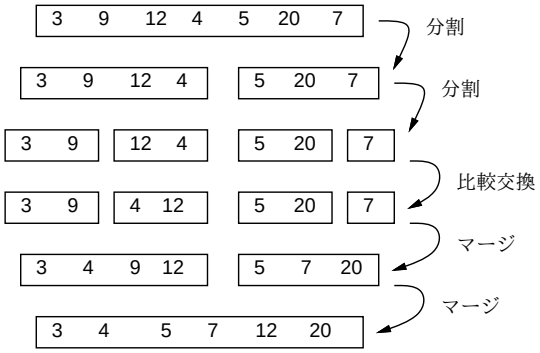

図 6: マージソートによる整列

この計算量であるが、1 つの手続きの中だけ (下請けを呼び出さない範囲) で考えると、長さ N のときに整列し終わった 2 つの列をマージする手間が問題である。マージ自体は 2 の列の先頭を比較して、より小さい方を取って行くだけなので、(最終的に N 個の要素を取るので)N 回の比較で済む。次に再帰呼び出しの方を考えると、N 個の列を 2 つに分けてそれぞれ自分自身を呼ぶのだから、結局 1 段呼ぶごとに合計 N 個の比較が増える。何段の呼び出しが起きるかというと、$\log_2 N$ 段。なので、計算量は $O(N \log N)$ になる。

クイックソート ($O(N \log N)$)

クイックソートもやはり再帰手続きを活用するが、マージソートのようにいきなり自分自身を呼ぶのでなく、まずランダムに「ピボット」と呼ばれる値 p を選び、列全体を「p より小さい」「p」「p より大きい」の 3 部分に分かれるように分け、「小さい」「大きい」部分についてそれぞれ自分自身を呼ぶ (図 7)。

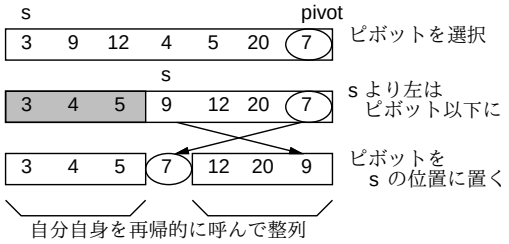

図 7: クイックソートによる整列

クイックソートでは、ピボットを選んだあとの分割に N 個の要素を全部見ていくのでそこで N 回の比較があり、段数は「平均して分割されると思えば」やはり $\log_2 N$ に比例する段数になるので、計算量は $O(N \log N)$ となる。この点はマージソートと同じだが、マージソートに比べて記憶領域が少なくて済み、また操作も効率よくしやすいのでこの名前がある。ただし、ピボット選択時に偏りがあると計算量が大きくなってしまうという弱点があるのだった。

4.2 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 を変えて計測してみる (計測は海外出張中の機上でノート PC でやりました)。

$O(N^2)$ 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る。一方、$O(N \log N)$ のマージソートとクイックソートは百万くらいのデータであれば十分実用になる。

ところで、ビンソートと基数ソートはどうだろうか。ビンソートが異常に速いことは分かるが…(もっと大きい N で測ってみると、基数ソートもクイックソートやマージソートより勝っているのがわかるはず)。これらを再度読んで原理を解説しよう。

表 1: 単純挿入法とバブルソートの所要時間

データ数 (10^3)	10	20	30	50	100	200	300	500	1000
単純挿入法	853	3,390	7,606	21,145	-	-	-	-	-
バブルソート	2,687	10,561	23,739	66,068	-	-	-	-	-
マージソート	31	45	61	78	164	358	581	909	1,730
クイックソート	11	20	27	41	89	170	263	447	936
ビンソート	4	5	5	6	7	10	14	20	36
基数ソート	15	25	35	64	110	241	341	543	1,049

4.3 ビンソート

ビンソートとは、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るのだった。コードを再掲する:

```
static void binsort(int[] a, int size) {
  int[] bin = new int[size]; // initially, all zero
  for(int i = 0; i < a.length; ++i) { bin[a[i]] += 1; }
  for(int i = 0, k = 0; k < bin.length; ++k) {
    for(int j = 0; j < bin[k]; ++j) { a[i] = k; ++i; }
  }
}
```

「範囲が広くない」という制約の理由は明らかで、その範囲全部をカバーする配列を作っている。たとえば値が 0〜1,000,000 だったら大きさ百万の配列が必要なわけで、それより大きいとちょっと実用上使えない。

その替わり整列作業は簡単である。というのは、すべての要素を順に見ながら、「その値が何個あったか」を数えていく (それですべての値に対応する配列が必要だったわけだ)。そして、最後まで数え終わったら、今度は先頭から順にその個数ずつ値を作り出して元の配列に書き込んで行けばいい (図 8)。

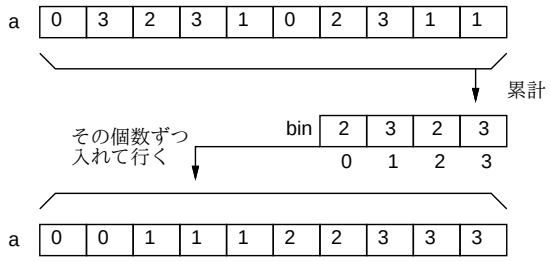

図 8: ビンソートによる整列

とすると、このアルゴリズムの時間計算量はどうなるだろうか。まずすべてのデータを順に操作するから $O(N)$ だが、それだけではない。最後に値の数を E とすると、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る。なので、時間計算量は $O(N+E)$ になる。もっとも、値の範囲が 1 万でデータ数が百万ならほとんど $O(N)$ といってよい。

ただしもう 1 つ、E 個ぶんの配列を必要とすることも忘れてはいけない。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るが、ビンソートはまさにその例である。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」という。キー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ある。

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 $O(N)$ だが、マージソートだけは「別の場所にマージして戻す」ので $O(2N)$ になっている。

4.4 基数ソート

では、基数ソートはどうだろうか。コードを再掲する:

```
static void radixsort(int[] a, int size) {
  int[] b = new int[a.length], c = new int[a.length];
  for(int mask = 1; mask < size; mask = mask * 2) {
    int bc = 0, cc = 0;
    for(int i = 0; i < a.length; ++i) {
      if((a[i] & mask) == 0) { b[bc] = a[i]; ++bc; }
      else                   { c[cc] = a[i]; ++cc; }
    }
    for(int i = 0; i < bc; ++i) { a[i] = b[i]; }
    for(int i = 0; i < cc; ++i) { a[bc+i] = c[i]; }
  }
}
```

何をやっているか分かりましたか。変数 mask は最初 1 で、毎回 2 倍になるから、2 進表現でいうと「0000…1」「000…10」「00…100」のように毎回「1」の場所が右へ移動していく。そして「&」はビット単位の and 演算なので、その「1」の箇所に a[i] 側でも 1 があるかどうかで if 文の枝分かれをしている。枝分かれ内の処理は、そこが 0 なら配列 b、1 なら配列 c に値を順番に詰めて行き、最後にまず b、続いて c の値を a にコピーし戻している。そうすると、外側の for 文の 1 周目では最下位ビットが 0 のものが前半、最下位ビットが 1 のものが後半に移動する (図 9 では 4 ビットの 2 進数のデータを例にこの様子を示している)。

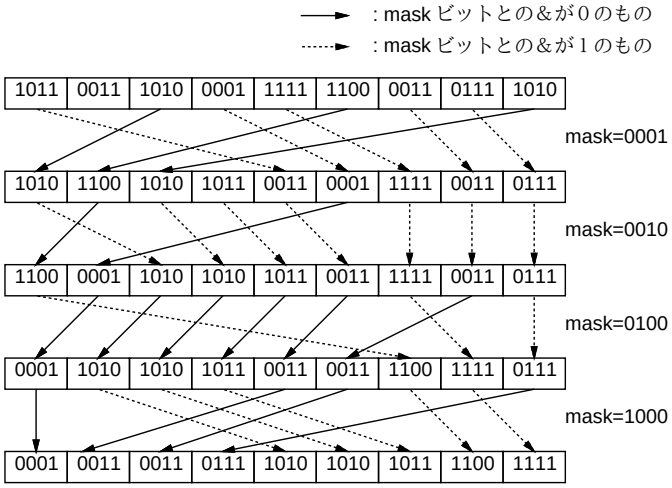

図 9: 基数ソートによる整列

2 周目では下から 2 ビット目が 0 のものが前半、1 のものが後半に移動するが、そのとき 1 周目の処理つまり最下位ビットが 0 のものが先という状態はそれぞれの中では維持されている。3 周目では下から 3 ビット目の 0、1 で前半、後半に分かれるが、そのとき下 2 ビットについての処理の結果は維持されている。このようにして、下から 1 ビットずつ処理し、値の範囲から見てもはや「1」が出てこないところまで来たら終わる。

そういうわけで、キーの範囲を E とすると、キーのビット数は $\log E$ なので、基数ソートの時間計算量は $O(N \log E)$ ということになる。もっとも、ビット数は最大でも 32 とか 64 程度なので、これを定数と思えば $O(N)$ の時間計算量と思ってもよい。

4.5 その他のアルゴリズム

4.5.1 最大公約数

2 つの数 $M, N (M < N$ とする) の最大公約数のベタなバージョン (M からカウンタを 1 ずつ減らして行き、それで両者が割り切れることをチェックする方法) は当然 $O(M)$ になる。ではおなじみ「大きい方から小さい方を引いて行く」バージョンはどうなのだろう。最善の場合は $M = N$ のときで、すぐ終わる。最悪の場合は $M = 1$ のときで、$N - 1$ 回引き算をしないと終わらない。平均は…? そこで、実験プログラムを作ってみた。

M と N の最大値をそれぞれ指定し、その範囲内で乱数を生成して最大公約数を計算し、これを百万回実行する所要時間を計測表示している。

```
java ...
m n> 100 100
time = 77
m n> 1000 1000
time = 81
m n> 10000 10000
time = 101
m n> 100000 100000
time = 121
m n> 1000000 1000000
time = 143
```

これをみると、値が 10 倍ずつ大きくなるときにほぼ一定ずつ時間が増えているので、$O(\log M)$ に見える。それは、両者がほぼ同じ値のときはループ 1 回について一定比率で大きい方が減って行くだろうと考えるとうなずける。しかし、M と N の大きさが違うとどうだろう?

```
m n> 10 100
time = 72
m n> 10 1000
time = 193
m n> 10 10000
time = 1414
m n> 10 100000
time = 13575
```

おやおや、すごく遅く…それは当然で、N が M の 10000 倍なら 10000 回引く必要があるわけだから。ということは M と N の比率が問題、つまり $O(\frac{N}{M})$ だろうか? しかし比率が一定でも値が大きくなると変わるのは上記の通り:

```
m n> 10 100000
time = 13575
m n> 100 1000000
time = 24331
m n> 1000 10000000
time = 33914
```

確かに 10 倍ごとに一定ずつ増えている。とすると、$O(\frac{N}{M} \times C \log M)$ だろうか (実際は定数倍は無視するので C は不要だが説明の都合上入れてある)。しかしその「一定」はさっきの 20msec に対して 9000msec とずっと大きい。つまり C は定数ではなくまだ何かの関数。そこでさらに実験して、$\frac{N}{M}$ と「一定」の関係を調べてみると次のようになった。

$\frac{N}{M}$	10^0	10^1	10^2	10^3
時間の差 (msec)	20	110	1000	9000

なるほど、比率が 10 倍ごとに時間差が 0.9 倍ですか。10 倍ごとに 10 倍なら非常にすっきりと $O((\frac{N}{M})^2 \times \log M)$ になるのだが、微妙に違うので $O((\frac{N}{M})^C \times \log M), C \sim 1.9$ くらいということか。そうなる理由は上記に加えて、M, N が大きくなるほどループ回数が増えて、その結果途中で引き算したとき運悪く片方がすごく小さくなってループがぐっと増えてしまう可能性も増えるから、ということだろうか。まあここまでやればすばらしいが、いくらかでもこういう雰囲気が味わえれば課題としては大成功だと思います。どうでしたか?

4.5.2 フィボナッチ数

やってみればすぐ分かるが、再帰的定義そのままのフィボナッチ数の計算は、$fib(N)$ の計算に $fib(N-1)$ と $fib(N-2)$ を実行し、それがさらに $fib(N-2)$ と $fib(N-3)$、$fib(N-3)$ と $fib(N-4)$ をそれぞれ呼び、というふうに「倍々」になるので、$O(2^N)$。これは非常にのろい。ループで計算する場合は当然 $O(N)$。最後に説明した、行列計算を使ってなおかつ行列積の計算を工夫する方法だとどうだろう。その「工夫」の漸化式を再掲する。(注記: これは漸化式なのだから、Q^{n-1} とか $Q^{\frac{n}{2}}$ のところをさらに展開する必要がある。でないとちっとも速くならないので注意。)

$$Q^n = \begin{cases} E & (n = 0) \\ QQ^{n-1} & (n \text{ が正の奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が正の偶数}) \end{cases}$$

この場合、「正の奇数」が選ばれるのは N を 2 進表現したときに「1」が現れる回数と等しい。「正の偶数」が選ばれるのは N を (奇数なら 1 を引きながら) 半分ずつにしていき 0 になるまでやるので、2 進表現の桁数つまり $\log N$。上記「1」の数は平均すると桁数の半分くらいだから $\frac{\log N}{2}$、すると全体として $O(\log N)$ となる。実験して確かめてくれましたか?

4.5.3 組み合わせの数

再帰的定義はフィボナッチと同様、倍々の呼び出しになるので、$O(2^N)$。掛け算での計算はループの回数が R 回だから $O(R)$。「パスカルの三角形」の場合はどうだろうか。再掲すると次のものを計算するわけだ。

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 ...
```

これを N 段目まで作るとなると、要素数が $\frac{N(N+1)}{2}$ あるから、計算量としては $O(N^2)$ ということになるはず。実験してみましたか?

演習 9 「抽象データ型の機能を活用してみる」または「計算量について考察してみる」のに適すると思われるプログラムを設計・制作し、その結果について報告せよ。(たとえば画像生成が好きなら画像関係を抽象データ型と組み合わせることもできるし、もっと別の題材でもよい。計算量について今回の資料を見てもっとやりたいと思うならそちらの方向で探求してみてもよくて、その場合アルゴリズム本体は既に出ているもので、それを呼び出す計測用のプログラムを作ったということでもよい。つまり各自の腕前に合わせて自分で課題を設定していただいて構いませんということ。)

A 本日の課題 5A

「演習 4」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 5A」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. 文字列オブジェクトについて学びましたが、納得しましたか。

   Q2. 自分でも抽象データ型を作って使えそうですか。

   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 5B

次回までの課題は「演習 5」「演習 7」「演習 8」「演習 9」の (小) 課題から 2 つ以上選んで報告することです。「演習 9」の内容を含め、実力相応に選んでください。あまり無理はしないで結構ですので。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは授業開始時刻の **10 分前** までに久野までメールで送付してください。

1. Subject: は「Report 5B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 文字列が操作できるようになりましたか。

   Q2. 自分なりのクラスが作れるようになりましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。