情報科学 2006　久野クラス　＃4

久野　靖*

2006.10.27

はじめに

前回の感想で「難しくなってきてしんどい」というご意見をいくつか頂いていますが、今回が終わったら 11 月は授業が 1 回しかない (11/3 →祝日、11/10 →久野海外出張、11/24 →駒場祭) ので、一息つけると思います。なので今回とりあえず頑張りましょう。今回の主な内容は次の通りです。

- 手続き (関数) による抽象化、再帰関数
- レコード型、画像の生成
- アルゴリズムと計算量

関数が使えると自分のプログラムを見通しよく書けるようになりますので、最初は慣れないと思いますが意識して活用してください。そして、画像が生成できるとこれまでの数字ばっかりとは違ったものが扱えて楽しめると思いますので、それを次回までの課題にします。工夫して絵を作ってみてください。もう 1 つ、「プログラムの速さ」の問題は好きな人が多いようですので、計算量の話も課題にします。

1　演習問題解説 — $f(x) = 0$ の解

1.1　演習 2a — 区間 2 分法

区間 2 分法は自分でやってくれた人がほとんどだと思うけど一応。

- 実数 n を入力する。
- a ← 0。b ← n。
- $|a - b| > 0.000001$ である間繰り返し、
- 　c ← (a + b) / 2。
- 　もし c*c - n ＞ 0 なら、
- 　　b ← c。
- 　そうでなければ、
- 　　a ← c。
- 　枝分かれおわり。
- 繰り返しおわり。
- a を出力する。

Java では次の通り。

```
import java.io.*;

public class r3ex2a {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value n = ");
    double n = (new Double(in.readLine())).doubleValue();
    double a = 0.0, b = n;
```

*筑波大学大学院経営システム科学専攻

```
    while(Math.abs(a - b) > 0.000001) {
      double c = (a + b) / 2.0;
      if(c*c - n > 0) { b = c; }
      else            { a = c; }
    }
    System.out.println("squareroot: " + a);
  }
}
```

1.2 演習 2b — ニュートン法

ニュートン法の方が理屈は面倒だけど計算は枝分かれが無くて済むので簡単である。

- 実数 n を入力する。
- r ← n
- $|r^2 - n| > 0.000001$ である間繰り返し、
- $\quad$ r ← $\frac{r}{2} + \frac{n}{2r}$
- 繰り返しおわり。
- r を出力する。

これの Java 版は次のとおり。

```
import java.io.*;

public class r3ex2b {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("value n = ");
    double n = (new Double(in.readLine())).doubleValue();
    double r = n;
    while(Math.abs(r*r - n) > 0.000001) {
      r = 0.5*r + 0.5*n / r;
    }
    System.out.println("squareroot: " + r);
  }
}
```

1.3 演習 3 — 配列の演習

演習 3 は入力部分は配列の例題そのままで、計算だけ工夫すればよい。アルゴリズムは略して Java のコードだけ記そう。まず最大と最小。

```
import java.io.*;

public class r3ex3a {
  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 max = a[0], min = a[0];
    for(int i = 1; i < a.length; ++i) {
      if(a[i] > max) { max = a[i]; }
      if(a[i] < min) { min = a[i]; }
    }
    System.out.println("max: " + max + " min: " + min);
  }
}
```

このようなループと配列を使う場合は、ふつうは「とりあえず max に 1 番目の値を入れておき、より大きい値が出て来たら入れ替える」方法になる。
　以下は紙面節約のため、計算部分だけ示す。次は最大の値が何番目に出て来るかだが、最大は上の問題と同様に求め、その後もう 1 度ループで調べながら同じ値の番号を打ち出す。

```
double max = a[0], min = a[0];
for(int i = 1; i < a.length; ++i) {
  if(a[i] > max) { max = a[i]; }
}
for(int i = 0; i < a.length; ++i) {
  if(a[i] == max) { System.out.println(i); }
}
```

最後の問題は上のと似ているが平均を求めてそれ以下ということで、平均は例題と同様に合計を求めてデータ数で割ればよい。

```
double s = 0.0;
for(int i = 0; i < a.length; ++i) { s = s + a[i]; }
double avg = s / a.length;
for(int i = 0; i < a.length; ++i) {
  if(a[i] < avg) { System.out.println(a[i]); }
}
```

1.4　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。

2 手続き/関数

2.1 手続きと抽象化

プログラミング言語上の概念**手続き** (procedure) とは、ひとまとまりの動作に名前をつけたもの。そのひとまとまりを**呼び出す**ことによって何箇所からでも利用できる。また、呼び出すときに**パラメタ** (parameter) を渡すことで、呼び出す箇所ごとに動作の内容をいくらか調節できる。

たとえば、「整数 n が素数かどうか調べる」という手順はさんざんやったが、それに名前をつけておくと便利。疑似コードでは次のようにまず名前とパラメタを明示し、それから動作の内容を書く:

- isPrime(n) — 2 以上の整数 n が素数か否かを返す。
- 　i を 2 から n の手前まで変えながら繰り返し、
- 　　もし n が i で割り切れるなら、「いいえ」を返す。
- 　繰り返しおわり。
- 　「はい」を返す。

ここで「〜を返す」というのは、手続きから結果を返すことができるので、その値を返すという動作を意味する。値を返すと手続きの実行はそこで終わる (呼び出した側に戻る) ことに注意。だから上の手順では、割り切れる数が見つかったとたんに「いいえ」を返して終わることになる。また、パラメタ n を渡していることも重要。だって、ある数を指定して、その数が素数かどうか知りたいわけだから、パラメタがないと意味がないでしょう?

さて、これを Java で書くと次のようになる。「返す」のは return 文。

```
static boolean isPrime(int n) {
  for(int i = 2; i < n; ++i) {
    if(n % i == 0) { return false; }
  }
  return true;
}
```

いくらか説明しておこう。

- static というおまじないは main と同じ。これは「単独の手続き」を表している。単独でない手続きについてはいずれ。
- boolean というのはもちろん論理型だが、つまりこの手続きは論理型の値を返す、ということを意味している。値を返す時はその返す値の種類はまず明示しておく必要があるので。
- 次の isPrime というのは手続きの名前。名前はクラスの名前や変数の名前なども同様だが、英字ではじまり、英数字が並んだもので、その規則の範囲で好きにつけてよい (-は使えない)。
- その後「(…)」の中にパラメタを書く。パラメタの指定は変数宣言と似ていて「型 名前, 型 名前, …」という形になる。
- その後「{ … }」で囲んだ中に関数の動作を書く。これは今までやっていた main() と同様だと思えばよい。

ではこれを使って、「指定した値までの範囲で、値が 2 だけ違う 2 つの素数がつ連続しているもの」を打ち出すという例を示そう。疑似コードは次の通り:

- 整数 m を入力。
- i を 4 から m まで変えながら繰り返し、

- もし i-2 が素数で、かつ、i が素数 なら、
- i-2 と i を出力。
- 枝分かれおわり。
- 繰り返しおわり。

これと手続きとを併せた Java プログラムを示す:

```
import java.io.*;

public class R4Sample1 {
  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 i = 4; i <= m; ++i) {
      if(isPrime(i-2) && isPrime(i)) {
        System.out.println("adjacent: " + (i-2) + " and " + i);
      }
    }
  }
  static boolean isPrime(int n) {
    for(int i = 2; i < n; ++i) {
      if(n % i == 0) { return false; }
    }
    return true;
  }
}
```

このように、手続きは `main()` とは並列にクラスの中に書いておけばよい。

しかしこれで何が便利なのだろう? それはもちろん、「もし i-2 が素数で、かつ、i が素数 なら」とひとことで言えること。つまり、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。たとえば区間 2 分法のコードも、解を求める関数 `f()` を別に定義しておいて「`if(f(c) > 0) ...`」と書いたらずっと読みやすいですね?

これを難しく言うと、手続きを使うことで**抽象化**して考えられる (つまり繰り返しとかごちゃごちゃしたものはさて置いて、ともかく「n が素数かどうか」を調べるものがあると思うことができる) のが利点と言える。

2.2 手続き/関数と副作用

先の例では値を返す手続きを扱ったが、値を返さない手続きというのも書くことができる。たとえば、「整数の配列 a の i 番目と j 番目に入っている値を交換する」という手続きを書いてみよう。

```
static void swapint(int[] a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
```

このように、返す型として `void`(「空」という意味の英語) を指定すると、値を返さない手続きになる。この場合は return 文がなくても動作の最後までくればそこで終わりということにしてよい。return 文は使いたければ使ってもいいが「`return;`」のように返す値は指定しないで使う。

ここで「関数」と「手続き」について整理しておく。

- 数学的には、「手続き」というものは出てこない。「関数」とは「入力空間 (定義域) から出力空間 (値域) への写像」として定義される。
- プログラミング的には、「手続き」とは「ひとまとまりの動作 (手順) に名前をつけたもの」。そのうち「値を返すもの」(Java の場合は返す型として `void` 以外のものが指定されている場合) を「関数」と呼ぶことがある。
- そして、特定の言語…C、C++などでは「手続き」のことをすべて「関数」と呼ぶ。Java などでもそう呼ぶことが多い。また、Java を含むオブジェクト指向言語では「メソッド」と呼ぶこともある。

そういうわけで呼び方が沢山あってややこしいが、ここでは当面「関数」「手続き」「メソッド」を適当に混ぜてつかう (適当ですみません)。

数学の「関数」とプログラミング言語の「関数」の最大の違いは、数学の関数では入力 (パラメタ) が同じなら結果は常に同じだが、プログラミング言語の関数はそうとは限らない。これは、実行する手順の中で変数などを書き換えてプログラム全体の状態を変化させることができるため。そのような動作のことを**副作用**と呼ぶ。たとえば次の例を見てみよう。

```
public class MySample {
  static int number = 1; // main と func から参照できる変数

  public static void main(String[] args) throws Exception {
  }
  static int func(int x) {
    number = number + 1; return x + number;
  }
}
```

ここで func() はパラメタ x に number を足した値を返す関数だが、1 度呼ばれるたびに number を増やしているので (副作用)、x の値が同じでも呼ばれるごとに返す結果は変わって来るわけだ。

ところで、変数 nubmer は手続き main() や func() の外側に (クラスのすぐ内側に) あるため、どちらの手続きからでもアクセスできる。こういう変数を**グローバル変数** (**広域変数**) と呼び、用途によっては便利だけれど下手に使うと副作用のため分かりにくい問題が起きることがあるので注意した方がよい。

なお、これまでに使って来た変数はすべて、手続きの中で宣言していたので**ローカル変数** (**局所変数**) ということになる。ローカル変数はその手続きが実行している間だけ存在していて、よその手続きからは参照できない。

手続きが副作用を持つのは、グローバル変数に対する書き換えだけとは限らない。たとえば上の swapint() の例では、パラメタとして渡された配列 a の中身を書き換えて副作用を起こしている。もともと、値を返さない手続きは (値を返さないのだから) 何らかの副作用を起こすことを目的としている。

キーボードからの入力、画面への出力なども副作用の一種であることに注意。たとえば in.readLine() は呼ばれるごとに違う文字列を (つまりあなたがキーボードから読み込んだ文字列を) 返す。

2.3 再帰関数

関数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、というものがある。これを**再帰** (recursion) という。たとえば、正の整数 x、y について、その最大公約数は次のようにして定義できたことを思い出そう。

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

これにそのまま従って Java の関数を書くことができる。例題全体を示そう。

```
import java.io.*;

public class R4Sample2 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("a = ");
    int a = (new Integer(in.readLine())).intValue();
    System.out.print("b = ");
    int b = (new Integer(in.readLine())).intValue();
    System.out.println("gcd: " + gcd(a, b));
  }
  static int gcd(int x, int y) {
    if(x == y)     { return x; }
    else if(x > y) { return gcd(x-y, y); }
    else           { return gcd(x, y-x); }
  }
}
```

プログラムそのものは大変分かりやすいでしょう? しかしこれでなぜ「堂々めぐり」にならずに計算が終了するのだろうか。それは、図 1 のようなものを描いてみれば分かる。

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従う。

- 問題の「簡単で明らかな場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。
- それ以外の場合は、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、大きい方から小さい方を引くことで少し簡単にしている)。

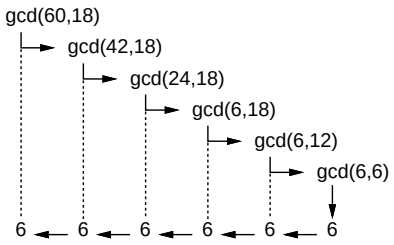

図 1: 再帰関数による最大公約数の計算

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。

演習 1 上の例題をそのまま打ち込んで動かせ。また、x や y に 0 や負の整数を入れるとどうなるかまず予測し、その後実際にやってみて予測を確認せよ。

演習 2 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 (関数部分以外は上の例題を直せば済む)。また、典型的な実行のようすを表す、図 1 のような図を描いてみよ (すみませんが何とか文字だけで工夫してみてください)。

a. 階乗の計算。

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

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。

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

この場合、関数の返す型は `String` であることに注意。また+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 (Java では整数どうしの「/」は自動的に切捨て除算になる)。

3 レコード型と画像

3.1 レコード型の利用

前回説明したように、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」ようなものだった。

Java ではレコード型はクラスを使って定義する。具体的には、次のようにすればよい。

```
static class 名前 {
  変数定義…
}
```

ここで「変数定義」は「int x = 10;」のようなこれまでの変数宣言 (+初期化) と同じもの。ただし、ここに書かれたものは単独の変数ではなく、そのレコード (クラス) に「付属した」もの (フィールド) になる。

たとえば、コンピュータ上で画像を扱うときは、多数の点の集まり (ピクセル) 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いことはご存じと思う。このピクセルの情報を表すレコード (クラス) を定義してみる:

```
static class Pixel {
  int r = 255, g = 255, b = 255;
}
```

なぜ RGB とも 255 を初期値にしたかというと、これで「白」になるから。実際にこのレコードを使うときは、次のようになる:

```
Pixel p = new Pixel();  // 変数宣言+レコードを生成
... p.r ...             // フィールド r を参照する
p.b = ...               // フィールド b を設定する
```

Java では配列のときと同様、レコードも変数を宣言するだけではなく、new を使ってそのレコード本体を作り出さないと使えないのに注意。

つまり、「Pixel p」までだと変数 p は何も指していない状態になっている (図 2 上)。そして、右辺の new でレコードを割り当ててそれを p に=で代入することではじめて利用できる状態 (図 2 下) になる。なお、この「何も指していない状態」はレコード、配列、オブジェクト型の変数すべてに存在し得る状態で、そのときはそれらの変数には null という特別な値が入っている。

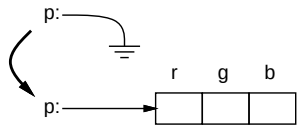

図 2: レコードの割り当て

3.2 2 次元配列

ピクセル 1 個ではつまらないので、ピクセルが 2 次元に (縦横に) 並んだ配列、つまり 2 次元配列 (添字が 2 個の配列) を作って画像を表すことにする。そこで最初に 2 次元配列について説明しておく。

```
int[] a = new int[100]; // 1 次元配列
```

だと、a[0]〜a[99] が使える、整数が並んだ配列ができるのだった。同様に、添字を 2 つ使いたい場合は

```
int[][] a = new int[100][20]; // 2 次元配列
```

のようにして作ることができる。この場合、添字の範囲は a[0][0]〜a[99][29]、ということになる。2 つの添字はここに書いたように、それぞれ別の [...] の中に入れて指定する。

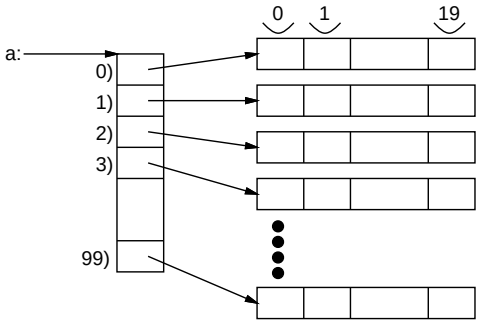

図 3: 2 次元配列

実際には、2 次元配列とは (Java など多くの言語では) 図 3 のように「配列の配列」になっている。つまり、上の例で a[0]、a[1] 等がそれぞれ 1 次元配列を指していて、そのため a[0][0]〜a[0][19]、a[1][0]〜a[1][19] 等が使えるというわけである。

ちょっと蛇足だが、1 次元配列、2 次元配列等で要素の数が少ない場合は、配列の生成と初期値の指定を一緒に行うことがきる。たとえば、要素数 5 ですべてに 1 が入った実数配列を作るには次のようにすればよい:

```
double[] vec = new double[]{1.0, 1.0, 1.0, 1.0, 1.0};
```

この右辺の書き方は**配列イニシャライザ**と呼ばれ、普通の式なので上のように変数に代入しなくても、たとえば関数のパラメタなどに直接書いてもよい。また、初期値は上のように定数でなくても一般の式でよい。2 次元の例として、3x3 の単位行列を示しておく:

```
double[][] mat = new double[][]{{1.0,0.0,0.0},{0.0,1.0,0.0},{0.0,0.0,1.0}};
```

要素数が多い場合はこれを沢山書くのでは大変すぎるので、普通にループを使おう。

さて、画像をピクセルの 2 次元配列として扱うには、次のようにすればよいだろうか:

```
int[][] buf = new Pixel[200][300]; // 300x200 の画像
```

実はこれではまだ「途中」である。つまり、2 次元配列そのものはこれで作れるが、その各要素はまだ null が入ったままであるので、個別に Pixel オブジェクトを作る必要がある。その具体例は例題を見た方が速いので、先に進もう。

3.3 例題: 画像を生成し書き出す

以下に示す例題は、色のついた円が 2 個重なっている画像を生成し書き出すものである (図 4)。説明の都合上、3 つに分けて示しているが、もちろんこれらをくっつけた形でファイルに書いてコンパイルして動かす必要がある。

図 4: プログラムで生成した画像

```
public class R4Sample3 {
  static class Pixel { int r = 255, g = 255, b = 255; }
  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);
    fillCircle(buf, 190, 100, 30, 100, 200, 80);
    writeImage(buf);
  }
```

クラスの冒頭に、レコード Pixel の定義がある。次は main() だが、その先頭で buf に Pixel の 2 次元配列を用意し、続いて 2 重の for ループでその各要素にレコード本体を入れている。これをやらないとレコードが使えるようにならないのは上で説明した通り。その後は、以下に説明する手続き fillCircle() を 2 回呼んで円を 2 つ描き、画像を書き出す。これで全体の動作は終わりである (やはり手続きに分けてあると分かりやすいでしょう?)。

では次に、fillCircle() を見てみよう。渡すパラメタは Pixel の 2 次元配列、塗りつぶす円の XY 座標と半径、そして塗りつぶす円の色を R/G/B 値で指定する。

```
  static void fillCircle(Pixel[][] a, double x, double y, double rad,
                         int r, int g, int b) {
    for(int i = 0; i < a.length; ++i) {
      for(int j = 0; j < a[0].length; ++j) {
        if((j-x)*(j-x)+(i-y)*(i-y) <= rad*rad) {
          a[i][j].r = r; a[i][j].g = g; a[i][j].b = b;
        }
      }
    }
  }
```

アルゴリズムは簡単で、「画像中のすべての a[i][j] について、点 (j, i) が中心 (x, y)、半径 r の円内に入っているかどうか調べ、入っている時だけ指定された色をその点の R/G/B 値として書き込む」だけである。円の中に入っているかどうかはもちろん、ピタゴラスの定理により $(j-x)^2 + (i-y)^2 \leq r^2$ で判断できる。

なお、ここで使っている表現は「配列の 2 つの添字のうち 1 番目が Y 座標、2 番目が X 座標で、しかも画像の下の方ほど Y が大きい」という奇妙なものだが、コンピュータグラフィクスではなぜかこうすることが多い (図 3 もこれに合わせて描いてある)。

最後に画像を書き出す手続きだが、これはできるだけ出力が簡単なフォーマットとして **PPM 形式**の画像を選んだ。

```
  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();
  }
}   // class R4Sample3 のおわり
```

PPM 形式の画像は、先頭に「P6 幅 高さ 255」という特別な行を置き、その後はひたすら各ピクセルの R/G/B 値をバイトとして書き出せばよい。数値を (数字の列ではなく) バイトとして書き出すには System.out.write() を使う。そしてその引数は byte 型である必要があるので、整数からの変換のためにキャストを使っている。

このプログラムの出力は、画面に出すのではなくファイルに書く必要があるので、次のように操作する (ファイル名の最後は「.ppm」としておくこと):

```
% javac R4Sample3.java       ←コンパイル
% java R4Sample3 >test.ppm   ←実行 (結果をファイルに)
% display test.ppm &         ←表示 (「&」はつけた方が便利)
```

PPM 形式の画像は ImageMagick というツールパッケージに含まれているコマンド display で表示できる。ECC の Mac にはこれが用意されているが (あと Mac では Preview でも PPM を表示可能)、自宅で Windows でやる場合には次のところの説明を読んで Windows 版を取りよせてください:

```
http://mechanics.civil.tohoku.ac.jp/soft/node43.html
```

取り寄せるバージョンは「6.3.3.0-Q16-windows-static」でいいと思います。

演習 3 画像ファイルを生成する例題を打ち込んでそのまま動かせ。動いたら円の位置や色を変更したり、円をもっと増やしてみよ。

演習 4 画像ファイルを生成する例題に、次のような手続きを追加してみよ。追加したものを活用した画像を生成してみること。

a. 長方形、楕円、三角形、直線を描く (指定した色で塗り潰す) ような手続き。長方形や楕円は回転できると嬉しいかも知れない。

b. 上記 a. や円の手続きについて、色を「重ね塗り」できるようにする。つまり透明度 $0 \leq p < 1$ 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 $p\, r_{old} + (1-p)\, r_{new}$ のように混ぜ合わせた値にする。

c. 上記 a. や円の手続きについて、全範囲を均一な色で塗るのではなく、徐々に色調が変わって行くようにする。

d. その他、美しい絵を描くのにあるといいと思うもの。

演習 5 「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。

4 アルゴリズムと計算量

4.1 演習問題解説: 単純選択法による整列

以下は前回演習 5 の解説を兼ねて計算量について説明する。プログラムの形としては、時間計測を行う形とする。まず、プログラム全体の形と併せて単純選択法のコードを示す (以下では整列する部分はすべて手続きの形にしてある)。

```
import java.io.*;

public class r3ex5a {
  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();
    double[] a = new double[n];
    for(int i = 0; i < a.length; ++i) { a[i] = Math.random(); }
    long t1 = System.currentTimeMillis();
    selectionsort(a);
    long t2 = System.currentTimeMillis();
    System.out.println(check(a) + ": " + (t2-t1));
  }
  static void selectionsort(double[] a) {
    for(int i = 0; i < a.length-1; ++i) {
      int k = selectmin(a, i, a.length-1); swapdouble(a, i, k);
    }
  }
  static int selectmin(double[] a, int i, int j) {
    int k = i; double min = a[i];
    for(int l = i+1; l <= j; ++l) {
      if(min > a[l]) { k = l; min = a[l]; }
    }
    return k;
  }
  static void swapdouble(double[] a, int i, int j) {
    double x = a[i]; a[i] = a[j]; a[j] = x;
  }
  static String check(double[] a) {
    for(int i = 0; i < a.length-1; ++i) {
      if(a[i] > a[i+1]) { return "NG"; }
    }
    return "OK";
  }
}
```

最後に呼んでいる `check()` は、1 箇所でもちゃんと並んでいないところがあれば`"NG"`、すべて大丈夫なら`"OK"`を返す。ちゃんと並べていないプログラムの時間を測ってもしかたないですから (「正しい結果を出さなくていいのなら、プログラムはいくらでも速くできる」という名言があります)。

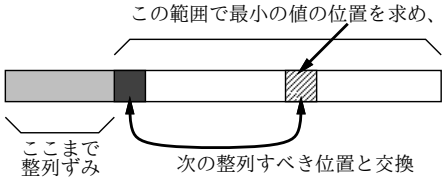

図 5: 単純選択法による整列

プログラムが先になったが、アルゴリズムは次の通り (手続き部分のみ示す):

- selectionsort(a) — 配列 a を単純挿入法で整列
- 　i を 0 から a.length の手前まで変えながら繰り返し、
- 　　k ← {a 中の i 番〜a.length-1 番の間の最小値の添字 }
- 　　{a 中の i 番と k 番の内容を交換 }

- 繰り返しおわり。

つまり、左から順に「残っているものの最小」をみつけてそれを「(交換を使って) 次の場所に置く」ことで左から小〜大の順に並べる (図 5)。なお、{…} の間は「これからさらに詳しく書く」ことを意味するものとしよう。で、詳しく書く方法として、その部分にさらに詳しい内容を埋め込むこともできるが、手続きとして分けて書くこともできる。ここでは手続きに分けるものとする。

- selectmin(a, i, j) — 配列 a の i〜j 番までで最小値の番号を返す
- k ← i。min ← a[i]。 // とりあえず最初を最小とする
- l を i+1 から j まで変えながら繰り返し、
- もし min > a[l] なら、
- k ← l。min ← a[l]。
- 枝分かれおわり。
- 繰り返しおわり。
- k を返す。

「a 中の i 番と j 番の内容を交換」は既に出ているので説明は省略。

4.2 時間計算量

一般に、「アルゴリズム (やプログラム) がどれくらい時間がかかるかを表す量」のことを**時間計算量**と呼ぶ。

`selectionsort()` を例題にして、これがどれくらい「速い」(遅い?) かを見積もってみよう。データの数を N とすると、`selectionsort()` の中からは `selectmin()` を N 回、`swapdouble()` を N 回呼び出す。ループ回数も N 回。なので、ここで消費される時間はたとえば $T_1 = C_1 N$ と表すことができる。

次に、`selectmin()` の処理を見てみよう。これは、1 回目には N 個の値を調べ、2 回目は $N-1$ 個の値を調べ、3 回目は $N-2$ 個の値を調べ、…のようになるので、1 回調べるのについて時間が C_2 掛かるものとすると、合計では次のようになる。

$$T_2 = C_2 N + C_2(N-1) + \cdots + C_2 1 = \frac{C_2}{2} N(N+1)$$

最後に、`swapdouble()` は N 回呼ばれて 1 回あたり C_3 とすると $T_3 = C_3 N$。これらを合計すると

$$T = T_1 + T_2 + T_3 = \frac{C_2}{2} N^2 + (C_1 + \frac{C_2}{2} + C_3) N$$

ここで、仮に C_1、C_3 が C_2 の 100 倍くらいあったとしよう (そんなに差があるとはとても思えないが)。だとしても、N として 1000 とか 10000 とかを入れて計算するわけだから、第 2 項 (N の 1 次の項) はほとんど無視できる。

このように、アルゴリズム/プログラムの実行時間を評価するときには、「入力 N や要求する誤差 δ に対してどれくらい時間が掛かるか」を見積もるが、その時もっとも次数の高い項が支配的になるので、低い次数の項は通常無視できる。

さらに、2 つのアルゴリズムがあって、片方が $T = C_1 N^2$、他方が $= C_2 N^3$ だったとすると、たとえ C_1 が C_2 の 1000 倍だったとしても (そんなこともあまりありそうにないが)、N に 10000 を入れるとそんな差はすぐに逆転してしまう。このため、定数も無視して、N の次数 (オーダー) だけを問題にして「このアルゴリズムの時間計算量は $O(N^2)$ である」などと言う。

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 $O(N)$ は最低必要である。これを線形時間と呼ぶ。たとえば最大や最小を求めるなど、線形時間のアルゴリズムがあるような問題はコンピュータで簡単に処理できると思ってよい。なお、N 個の値といっても、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がある。

少し込み入ったアルゴリズムは、$O(N^2)$、$O(N^3)$ などの計算量になる。これを**多項式時間**と呼ぶ。さらに時間計算量の大きなものとしては、$O(e^N)$ や $O(N!)$ などもあり、これらだとコンピュータで実用的に扱えるのは小さい N に限られてしまう。

4.3 演習問題解説: 比較交換法 (バブルソート)

前回資料で 2 番目の方法として例示されていた比較交換法 (バブルソート) に進もう。`main()` や下請け手続きは同様なので本体だけ:

```
static void bubblesort(double[] a) {
  boolean done = false;
  while(!done) {
    done = true;
    for(int i = 0; i < a.length-1; ++i) {
      if(a[i] > a[i+1]) { swapdouble(a, i, i+1); done = false; }
    }
  }
}
```

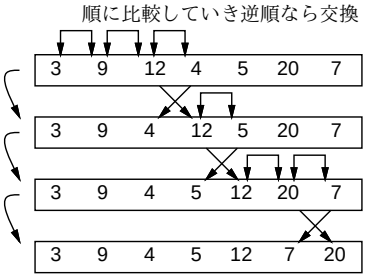

図 6: バブルソートによる整列

この方法では、左端から順に隣接する要素を比較していき、大小が逆のところで交換を行う。このため、一巡するともっとも大きい要素が右端に来ている。再度一巡すると、2 番目に大きい要素が右端から 2 番目に来る。これを「一巡してまったく交換が必要なかった」状態になるまで繰り返す。

さて、この時間計算量はどうだろうか。内側のループでは $N-1$ 回の比較を行う。そして、**最善の場合**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ある。つまり $O(N)$。しかし**最悪の場合**、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来るだけで終わってしまう。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 回必要になる。つまり $O(N^2)$。では平均的にはどうだろうか。平均的には、内側のループの繰り返しは N 回は必要ないだろうが、N に比例する回数 (たとえば $0.1N$ とか) が必要になりそうである。ということは、平均でも (定数倍は無視するので) やはり $O(N)$ になるわけである。

ここで単純挿入法とバブルソートの所用時間 (ミリ秒) を手元のマシンで実測してみた (図 1)。これを見ると、いずれも絶対値 (定数倍) の違いはあるが、N が 2 倍、3 倍になったとき所用時間が 4 倍、9 倍になっているので、確かに $O(N^2)$ の時間計算量らしいと言える。

表 1: 単純挿入法とバブルソートの所要時間

アルゴリズム	10,000	20,000	30,000
単純挿入法	540	2,142	4,805
バブルソート	1,847	7,360	16,665

4.4 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だろうか。次の方法 (**マージソート**) を見てみよう。これは呼び出し方として次のように、「配列のどこからどこまでを整列する」かを指定する:

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

その疑似コードを示そう。

- mergesort(a, i, j) — 配列 a の i 番から j 番の範囲を整列
- 　もし j ≤ i なら、
- 　　何もしない。 //長さが 0 や 1 ならもう並んでいる
- 　そうでなくてもし j = i+1 なら、
- 　　もし a[i] > a[j] なら、{a の i 番と j 番を交換 }。
- 　そうでなければ、
- 　　k ← (i + j) / 2。
- 　　mergesort(a, i, k)。mergesort(a, k+1, j)。
- 　　{2 つの整列された列をマージ (併合) する }
- 　枝分かれおわり。

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行き、長さ 1 や 2 のときはすぐ整列できるので、その後戻って来たら 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する (図 7)。

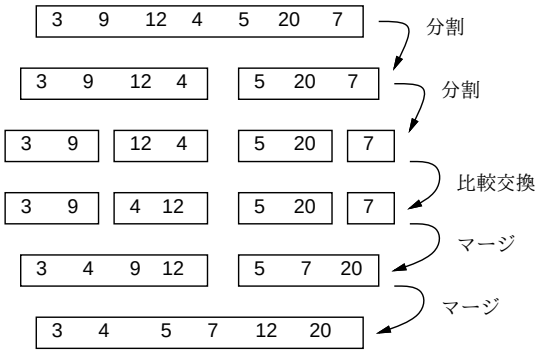

図 7: マージソートによる整列

なお、**マージ** (併合) とは、$(1,3,4,8)$ と $(2,5,6,7)$ のように 2 つの整列ずみの列があったときに、それを併せて $(1,2,3,4,5,6,7,8)$ のように 1 つの整列ずみの列にすることを言う。疑似コードにすると長いのでそれは略。

では計算量はどうだろうか。1 つの `mergesort()` の呼び出しを見ると、単純な場合 (長さが 0、1、2) は一定時間で済む (当然)。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、最後に併合する。自分自身に掛かる時間は分けて考えるとして、併合は両方の列の先頭を見て小さい方を取ることを繰り返せばいいので、$O(N)$ で済む。さて、再帰呼び出しの方はどうだろうか。長さ N の列を半分にしてそれぞれ `mergesort()` を呼ぶのだから、2 段目の呼び出しは $O(N/2)+O(N/2)=O(N)$。3 段目は 4 分の 1 の列について 4 つ呼ぶのでやはり $O(N)$。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 $log_2 N$。なので、全体では $O(NlogN)$ の計算量となる (計算量では log の底が何かも省略するのが通例)。では実際にマージソートのコードを示す:

```
static void mergesort(double[] a, int i, int j) {
  if(j <= i) {
    // nothing to do.
  } else if(j == i+1) {
    if(a[i] > a[j]) { swapdouble(a, i, j); }
  } else {
    int k = (i + j) / 2;
    mergesort(a, i, k); mergesort(a, k+1, j);
    int i1 = i, k1 = k+1, c = 0;
    double[] b = new double[j-i+1];
    while(i1 <= k || k1 <= j) {
      if(i1 > k)             { b[c] = a[k1]; ++c; ++k1; }
      else if(k1 > j)        { b[c] = a[i1]; ++c; ++i1; }
      else if(a[i1] > a[k1]) { b[c] = a[k1]; ++c; ++k1; }
      else                   { b[c] = a[i1]; ++c; ++i1; }
    }
    for(int l = 0; l < b.length; ++l) { a[i+l] = b[l]; }
  }
}
```

併合のところがちょっと面倒だが、要するに併合に使う配列 b を用意し、そこに 2 つの整列ずみ部分列からコピーしながら併合し、終わったら元の配列 a にコピーし戻すようになっている。

これを実行してみると、N が 10000、20000、30000 で 27msec、38msec、51msec と、比べものにならないくらい速い。つまり、定数倍の違いで頑張るよりも、時間計算量のオーダーが優れたアルゴリズムを見つけるほうが、はるかに効果的だというわけだ。

4.5 クイックソート

もう 1 つ別のアルゴリズムを示そう。これは**クイックソート**といういかにも速そうな名前がついている。

```
static void quicksort(double[] a, int i, int j) {
  if(j <= i) { return; }
  double pivot = a[j]; int s = i;
  for(int k = i; k <= j-1; ++k) {
    if(a[k] <= pivot) { swapdouble(a, s, k); ++s; }
  }
  swapdouble(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j);
}
```

非常に短いけど、説明されないと分からないですよね。まず長さ 0、1 なら何もしないのはマージソートと同じ。次に、マージソートと同じく列を 2 つに分けるが、こちらはある値 p(ピボットと呼ぶ) を選んで「左半分は p 以下、続いて p の値、右半分は p より大きい」状態にしてから、左半分と右半分をそれぞれ整列する。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」になるのでこれで完了なわけだ (図 8)。

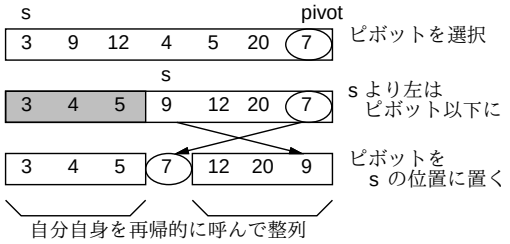

図 8: クイックソートによる整列

p としては「ちょうど列を半分ずつに分ける値」を使えるとベストだが、そんなものは分からないのでランダムに選ぶことにし、上のコードでは右端 (j 番目) の値を p にしている。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」番号を表している。そこで、k を i から j-1 まで (j 番はピボットが入っているので保留) 左から順に調べて、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ることがおこなえる。

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く。その後、自分自身を呼ぶわけだが、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べばよい。

では、クイックソートの計算量はどうだろうか。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 $log_2 N$ 回だが、ランダムに選んでいるのでその定数倍と考える。定数倍は無視するので、これも計算量は $O(NlogN)$ になる。

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 $O(N^2)$ ということになる。そんな運が悪いことはないだろうと思うかも知れないが、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。

演習 6 マージソート、クイックソートを打ち込んで動かし、所要時間を計測してみよ。また、クイックソートで「並んでいると最悪の場合になる」ことを確認し、この問題を緩和する方法を考えてみよ。

4.6 ビンソート、基数ソート

整列アルゴリズムにはだいぶ食傷したと思うが、「整列するデータが整数であり、かつ範囲があまり広くない」場合に使えるアルゴリズムである**ビンソート**と、上記の「かつ」以下の制約がない**基数ソート**をプログラムだけ載せておこう。まずビンソートから (データが整数になるので全体を示す):

```
import java.io.*;

public class R4Sample4 {
  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();
    int[] a = new int[n];
    for(int i = 0; i < a.length; ++i) { a[i] = (int)(10000*Math.random()); }
    long t1 = System.currentTimeMillis();
    binsort(a, 10000);
    long t2 = System.currentTimeMillis();
    System.out.println(icheck(a) + ": " + (t2-t1));
  }
  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; }
    }
  }
  static String icheck(int[] a) {
    for(int i = 0; i < a.length-1; ++i) {
      if(a[i] > a[i+1]) { return "NG"; }
    }
    return "OK";
  }
}
```

データは Math.random() による $[0,1)$ の一様乱数に 10000 を掛けて整数に切捨てることで 0〜9999 の整数の一様乱数を作っている。次に基数ソート (整列する手続きだけ):

```
  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]; }
    }
  }
```

こちらは上記の 10000 が 1 億でも 1 兆でも大丈夫である。[1]

演習 7 ビンソートと基数ソートの仕組みを解明し、計算量を検討してみよ。

5 既出アルゴリズムの別バージョン

5.1 最大公約数

上で出てきた最大公約数は引き算を使っていたが、替わりに剰余演算を使えば演算回数はずっと少なくなる (ユークリッドの互除法)。逆に、もっとベタなアルゴリズムとして、次のようなものも考えられる。

- i を $min(x,y)-1$ から 1 まで 1 ずつ減らしながら繰り返し、
- 　x も y も i で割り切れるなら、ループから抜け出す。
- 繰り返しおわり。
- i を打ち出す。

[1] なお「&」という演算子は「ビット単位での and 演算」を表す。変数 mask は 1、2、4、…等「2 進表現で 1 つのビットだけが 1 の数」になっているので、それと and を取って 0 かどうか見ることで「そのビットが 0 かどうか」が分かる。

なお、この方法を Java プログラムにする場合は、for 文のループに使う変数をループが終わった後でも使いたいわけだから、次のように分けて宣言しなければならないことに注意:

```
int i;
for(i = ... ) {
  ...
}
// ここで i が利用可能
```

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すごく遅い。別の方法として、たとえば x0 と x1 に 1 を入れておき、それからループで x0 にはこれまでの x1、x1 にはこれまでの x0+x1 を入れることを繰り返して計算することが考えられる。この方法だと計算量は $O(n)$ になると予想される。

もう 1 つこういうのはどうだろうか。

$$\begin{pmatrix} x_{i+1} \\ x_i \end{pmatrix} = \begin{pmatrix} x_{i-1} + x_i \\ x_i \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_i \\ x_{i-1} \end{pmatrix}$$

だから、$x_0 = x_1 = 1$ とおいてあとは上の漸化式で x_i を計算すれば i 番目のフィボナッチ数が求まる。漸化式といっても次々に同じ行列を掛けるだけだから、次の Q、v について $Q^n v$ を求めればよい:

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},\ v = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

この方法の Java プログラムを示そう (大半は 2x2 行列の積、2x2 行列とベクトルの積の手続き):

```
import java.io.*;

public class R4Sample4 {
  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();
    int[][] Q = new int[][]{{1,1},{1,0}}, M = new int[][]{{1,0},{0,1}};
    for(int i = 0; i < n; ++i) { M = mulMatMat(M, Q); }
    int[] F = mulMatVec(M, new int[]{1,1});
    System.out.println(F[1]);
  }
  static int[] mulMatVec(int[][] a, int[]b) {
    return new int[]{a[0][0]*b[0]+a[0][1]*b[1],a[1][0]*b[0]+a[1][1]*b[1]};
  }
  static int[][] mulMatMat(int[][] a, int[][] b) {
    return new int[][]{
      {a[0][0]*b[0][0]+a[0][1]*b[1][0], a[0][0]*b[0][1]+a[0][1]*b[1][1]},
      {a[1][0]*b[0][0]+a[1][1]*b[1][0], a[1][0]*b[0][1]+a[1][1]*b[1][1]}};
  }
}
```

これでも行列 Q を n 回掛けるから $O(n)$ だろうって? そうだけど、実は次の漸化式を利用すれば掛け算の回数をぐっと減らすことができ、従って計算量をもっと小さくできる (E は単位行列を表す)。

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

5.3 組み合わせの数

組合せの数も再帰的定義そのままでは非常に遅い。別の方法として、前回やった掛け算を使う方法がまずある。また、「パスカルの三角形」を作る方法がある:

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

演習 8 何か 1 つ、ある計算に複数のアルゴリズムが存在するようなものを選び、その複数 (3 つ以上あればよい) のアルゴリズムの時間計算量をそれぞれ見積もってみよ。また、実際にそのようになっているかどうか、プログラムを動かして計測し確かめてみよ。予想と合わない場合はその理由も検討すること。

これまでに出てきた題材としては「素数の判定」「素数の列挙」「最大公約数」「組合せの数」「フィボナッチ数」「整列」「平方根」などがあったが、これ以外に自分で考えてもよい。短時間で終わってしまう計算の場合は、その計算を何回も繰り返して実行させて時間を測り、割り算で 1 回あたりの時間を求めればよい。

A　本日の課題 4A

「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 4A」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. 手続き/関数について学びましたが、納得しましたか。

   Q2. 画像の生成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 4B

次回までの課題は「演習 5」と「演習 8」の 2 つとします。いずれも、それをやるに当たって他の課題も結果的にやることになる「かも」知れませんがそのときはよろしく。ただし、実力相応で構いませんのであまり無理はしないでください。

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。また、今回は分析/考察が重視されますのでよろしく。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 4B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 演習 5 で作成しだプログラムのソース。
4. その説明と分析/考察。
5. 演習 8 で作成した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. 手続きが使いこなせるようになりましたか。

   Q2. 時間計算量について納得しましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。