計算機プログラミング I (7)

プログラムの整理とアルゴリズム

2004 年 11 月 29 日　増原 英彦

1　クラスメソッドによるプログラムの整理

教科書ではクラスメソッドの定義方法は 6.6 章 (と練習問題 6.13) で簡単に説明されている。これを理解するためには、6 章の冒頭からのインスタンスメソッドに関する説明を読んだ上で、その変種としてクラスメソッドの定義方法を知ることになる。

ここではクラスメソッドを用いて複雑なプログラムを整理する手順を紹介する。まず、複雑なプログラムの例として、 morphing を行う図 1 のプログラムを見てみよう。各行でどのような操作を行っているからプログラム中のコメントを参照せよ。

1.1　内分点の計算

27〜28 行目の式は 0 番目と (`nTurtles-1`) 番目のタートルの X 座標を `i`:(`nTurtles-1-i`) に内分する座標を求めている:

```
i*hm[nTurtles-1].getX() / (nTurtles-1)
+ (nTurtles-1-i)*hm[0].getX() / (nTurtles-1)
```

この式を「内分を求める関数」を使って整理してみよう。「内分を求める関数」は (予め用意されてはいないので) 新たにクラスメソッドとして定義する。そして上の式を、新たに定義したクラスメソッドを呼び出すように書き直す。

1.1.1　内分を求めるクラスメソッドの定義

クラスメソッドを定義するには、次のようなことを順に考えてゆく:

1. まず新たに定義するクラスメソッドの**名前を決める**。自由な名前をつけてよいが、意味のある名前が良い。ここでは「内部点」(internally dividing point) を略して`divPoint` と付ける。
2. 次に `divPoint` を計算する際に**どのような値を用いているか**を考える。ここでは「2 つの座標値と、内分の比を表わす 2 つの数」である。
3. 上の**値にそれぞれ名前を付ける**。ここでは 2 つの座標を `x0`, `x1` と、また内分比を `m`, `n` とする。つまり`divPoint` は「座標値`x0`, `x1` を`m`:`n` に内分する座標値を求める」ものと言うことができる。
4. それぞれの**値の型**(値の種類) **を考える**。ここでは全て整数であるので、Java 言語では全て `int` 型となる。
5. 内分点を**計算する式を考え**、上の 3 で決めた値の名前で書く。ここでは `(n*x0 + m*x1)/(m+n)` である。
6. 上の計算式で求められる**答えの型**(値の種類) **を考える**。ここではやはり整数—`int` 型である。

以上をもとにクラスメソッドの定義を作る。上の 1〜6 で決めた名前・型・式を以下のように並べる。括弧内は上の番号に対応している:

```
1  // 円と星を描き、その間を補完する
2  public class Mix {
3      public static void main(String[] args) {
4          int radius = 100;     //円の半径
5          int nTurtles = 5;     //図形の数 (含円と星)
6          int starEdges = 5;    //星の角数
7          int steps = 30;       //分割数
8          //タートルを表示するフレームを作る
9          TurtleFrame f = new TurtleFrame(radius*2*nTurtles, radius*2);
10
11         Turtle[] hm = new Turtle[nTurtles];         //nTurtles 匹のタートルをしまう配列
12         for (int i = 0; i < nTurtles; i++) {        //nTurtles 回の繰り返しで
13             hm[i] = new Turtle(2*radius*i,radius,0); //i 番目のタートルを作り
14             f.add(hm[i]);                           //フレームに表示する
15         }
16         hm[nTurtles - 1].lt(360/starEdges); //星形の最初の向きを変えておく
17
18         for (int step = 0; step < steps; step++) { //steps 回の繰り返しで作図
19             hm[0].fd((int)(Math.PI*2*radius/steps)); //0 番に円の一部を描かせる
20             hm[0].rt(360/steps);
21             if (step % (steps/starEdges) == 0)      //nTurtles-1 番に
22                 hm[nTurtles - 1].rt(360*2/starEdges); //星形の一部を描かせる
23             hm[nTurtles - 1].fd(10*radius/steps);
24
25             for (int i = 1; i < nTurtles - 1; i++) {
26                 //i 番目のタートルを i:(nTurtles-1-i) の内分点に移動
27                 hm[i].moveTo(i*hm[nTurtles - 1].getX() / (nTurtles-1)
28                             + (nTurtles-1-i)*hm[0].getX() / (nTurtles-1),
29                             i*hm[nTurtles - 1].getY() / (nTurtles-1)
30                             + (nTurtles-1-i)*hm[0].getY() / (nTurtles-1),
31                             0);
32             }
33         }
34     }
35 }
```

図 1: Morphing を行うプログラム

```
static [型 (6)] [名前 (1)] ([1 番目の値の型 (4)] [1 番目の値の名前 (3)],
                           [2 番目の値の型 (4)] [2 番目の値の名前 (3)], ...)  {
  return [式 (5)];
}
```

具体的には次のようになる:

```
static int divPoint(int x0, int x1, int m, int n) { //x0 と x1 を m:n に内分する
  return (x0*n + x1*m)/(m+n);                        //点を計算する
}
```

1.1.2 クラスメソッドを「使う」

divPoint(式 1, 式 2, 式 3, 式 4) という式は、式 1 と式 2 の値の間を、(式 3 の値) 対 (式 4 の値) で内分する値を計算する。27〜28 行目の式は「hm[0].getX() とhm[nTurtles-1].getX() をi:(nTurtles-1-i) に内分する点を求め」ていたので、divPoint を使って次のように書き直すことができる:

```
divPoint(hm[0].getX(), hm[nTurtles-1].getX(), i, nTurtles-1-i)
```

同様に 29〜30 行目は「Y 座標の内分点を求め」ているので、同じdivPoint を使って書き直すことができる。

結果として全体のプログラムは図 2 のようになる。このとき、**クラスメソッド**divPoint **の 定義 は、図 1 のプログラムの 34 行目と 35 行目の間に書かれる**ことに注意せよ。(実は 2 行目と 3 行目の間でも構わない。)

```
1  // 円と星を描き、その間を補完する
2  public class Mix2 {
3      public static void main(String[] args) {
4          int radius = 100;     //円の半径
5          int nTurtles = 5;     //図形の数 (含円と星)
6          int starEdges = 5;    //星の角数
7          int steps = 30;       //分割数
8
9          TurtleFrame f = new TurtleFrame(radius*2*nTurtles, radius*2);
10
11         Turtle[] hm = new Turtle[nTurtles]; //nTurtles 匹のタートルを作って表示
12         for (int i = 0; i < nTurtles; i++) {
13             hm[i] = new Turtle(2*radius*i,radius,0);
14             f.add(hm[i]);
15         }
16         hm[nTurtles - 1].lt(360/starEdges); //星形の最初の向きを変えておく
17
18         for (int step = 0; step < steps; step++) {
19             hm[0].fd((int)(Math.PI*2*radius/steps)); //円の一部を描く
20             hm[0].rt(360/steps);
21             if (step % (steps/starEdges) == 0)      //星形の一部を描く
22                 hm[nTurtles - 1].rt(360*2/starEdges);
23             hm[nTurtles - 1].fd(10*radius/steps);
24
25             for (int i = 1; i < nTurtles - 1; i++) {
26                 //i 番目のタートルを i:(nTurtles-1-i) の内部点に移動
27                 hm[i].moveTo(divPoint(hm[0].getX(), hm[nTurtles-1].getX(),
28                                       i, nTurtles-1-i),
29                              divPoint(hm[0].getY(),hm[nTurtles-1].getY(),
30                                       i,nTurtles-1-i),
31                              0);
32             }
33         }
34     }
35
36     // x0 と x1 を m:n に内分した値を求める
37     static int divPoint(int x0, int x1, int m, int n) {
38         return (x0*n+x1*m)/(m+n);
39     }
40
41 }
```

図 2: 内分点を計算するメソッドを用いたプログラム

1.2 内分点への移動

図 2 の 27〜31 行目の文は、0 番目と (nTurtles-1) 番目のタートルの位置をi:(nTurtles-1-i) に内分する点を求め、i 番目のタートルをそこへ移動させている。このような **処理** もクラスメソッドを用いて整理することができる。前と同様の手順で考える:

1. **名前**はmoveToDivPoint とする。
2. **用いる値**は「位置を参照する 2 つのタートルと、内分の比、移動させるタートル」である。
3. 上の**値の名前**を順に turtle0, turtle1, m, n, turtle2 とする。
4. **値の型**は、turtle0,1, 2 はタートルオブジェクトなのでTurtle 型で、内分比はint 型である。

5. (計算式のかわりに) **処理内容**を考える。ここでは、X 成分、Y 成分ごとに内分点を求め、それぞれを内分する位置にmoveTo メソッドを使って移動するので、次のようになる:

```
turtle2.moveTo(divPoint(turtle0.getX(),turtle1.getX(),m,n),
               divPoint(turtle0.getY(),turtle1.getY(),m,n),
               0);
```

6. **答えの型**はvoid である。このメソッドのように「処理をする」だけで計算した答えを求めるものではない場合には、「答えが無い」ことを示す型としてvoid を使う。

これらをまとめて、メソッドを定義して整理し直したのが図 3 である。31〜37 行目がmoveToDivPoint の定義、27 行目をメソッドの使用である。moveToDivPoint はdivPoint と違って答えを返さないので、「return **式**;」のようには**書いていない**ことに注意せよ。

```
1  // 円と星を描き、その間を補完する
2  public class Mix3 {
3      public static void main(String[] args) {
4          int radius = 100;     //円の半径
5          int nTurtles = 5;     //図形の数 (含円と星)
6          int starEdges = 5;    //星の角数
7          int steps = 30;       //分割数
8
9          TurtleFrame f = new TurtleFrame(radius*2*nTurtles, radius*2);
10
11         Turtle[] hm = new Turtle[nTurtles]; //nTurtles 匹のタートルを作って表示
12         for (int i = 0; i < nTurtles; i++) {
13             hm[i] = new Turtle(2*radius*i,radius,0);
14             f.add(hm[i]);
15         }
16         hm[nTurtles - 1].lt(360/starEdges); //星形の最初の向きを変えておく
17
18         for (int step = 0; step < steps; step++) {
19             hm[0].fd((int)(Math.PI*2*radius/steps)); //円の一部を描く
20             hm[0].rt(360/steps);
21             if (step % (steps/starEdges) == 0)      //星形の一部を描く
22                 hm[nTurtles - 1].rt(360*2/starEdges);
23             hm[nTurtles - 1].fd(10*radius/steps);
24
25             for (int i = 1; i < nTurtles - 1; i++) {
26                 //i:(nTurtles-1-i) の内部点に i 番目のタートルを移動
27                 moveToDivPoint(hm[0], hm[nTurtles-1], i, nTurtles-1-i, hm[i]);
28             }
29         }
30     }
31     // turtle0 と turtle1 を m:n に内分した位置へ turtle2 を移動する
32     static void moveToDivPoint(Turtle turtle0, Turtle turtle1,
33                                int m, int n, Turtle turtle2) {
34         turtle2.moveTo(divPoint(turtle0.getX(),turtle1.getX(),m,n),
35                        divPoint(turtle0.getY(),turtle1.getY(),m,n),
36                        0);
37     }
38
39     // x0 と x1 を m:n に内分した値を求める
40     static int divPoint(int x0, int x1, int m, int n) {
41         return (x0*n+x1*m)/(m+n);
42     }
43
44 }
```

図 3: 内分点に移動するメソッドを用いたプログラム

1.3 さらなる整理

他の処理や計算もクラスメソッドによって整理を続けると、図 4 のプログラムになる。以下のようなクラスメソッドが定義されている:

- `createTurtles`: タートルを作り、表示し、配列にしまって返す
- `circleStep`: 円 (正 n 角形) の一辺を描く
- `starStep`: 星形の一辺を描く

`createTurtles` は、タートルオブジェクトを「表示する」という処理と、タートルオブジェクトをしまった配列を「答えとして返す」という 2 つを同時に行っている。このような処理の場合は、処理をし (26〜30 行目) た上で、答え返す`return` 文 (31 行目) を書くことになる。

2 アルゴリズム

アルゴリズム (algorithm) とは、問題を解くための手順のことである。例えば、掛け算を筆算で行う方法や、逆行列を求めるガウスの消去法などは数学的なアルゴリズムである。(ただし、一般に計算機が扱う問題は数学的なものに限らない。)

一つの問題に対するアルゴリズムは複数あり得る。同じ問題を解く場合でも、アルゴリズムによっては繰返しや計算の回数が大きく違うことがある。プログラムは、アルゴリズムに従って計算機が処理をする手順を書き下したものなので、アルゴリズムの差はプログラムの実行時間や使用するデータ量などに反映することになる。

ある問題に対して、それを解くアルゴリズムがあれば、それに従ってプログラムを書くことはそれほど難しくない。しかし、問題を解くアルゴリズムを見つけること自体は簡単ではない。多くの問題は、より小さな部分問題に分けて考えることができる。また、典型的な問題については定番となるアルゴリズムが知られている。そこで普通は、(1) 問題を小さな問題に分割する (2) 典型的な問題について定番アルゴリズムを当てはめる、といったことをする。もちろん、分割もできずアルゴリズムも知られていない問題もあるので、その場合は知恵を働かせて新しいアルゴリズムを考え出すことになる。

ここではいくつかの例題について、それをどのように分割し、アルゴリズムを見つけるかを見てゆく。

2.1 最大公約数

問題: 2 つの整数 a, b の最大公約数を求める (ただし $a < b$ とする)

2.1.1 アルゴリズム 1

この問題は「ある数が a, b の公約数かを調べる」ことと「公約数のうち最大のものを見つける」ことに分けると簡単になる。

アルゴリズム 1(最大公約数): i を $1, 2, \ldots, a$ の順に変化させ、i が a, b の公約数であるかを調べる。最後に公約数であると分かったときの i の値が最大公約数である。 ∎

例えば、$a = 6, b = 21$ の場合であれば、

- $i = 1$: i は a, b の公約数で**ある**
- $i = 2$: i は a, b の公約数でない
- $i = 3$: i は a, b の公約数で**ある**
- $i = 4$: i は a, b の公約数でない

```
// 円と星を描き、その間を補完する
public class Mix4 {
    public static void main(String[] args) {
        int radius = 100;     //円の半径
        int nTurtles = 4;     //図形の数 (含円と星)
        int starEdges = 5;    //星の角数
        int steps = 30;       //分割数

        TurtleFrame f = new TurtleFrame(radius*2*nTurtles, radius*2);
        Turtle[] hm = createTurtles(nTurtles, radius, f); //タートルを作り配置
        hm[nTurtles - 1].lt(360/starEdges); //星形の最初の向きを変えておく

        for (int step = 0; step < steps; step++) {
            circleStep(hm[0],radius,steps);                          //円を描く
            starStep(hm[nTurtles-1], radius, steps, starEdges, step); //星形を描く

            for (int i = 1; i < nTurtles - 1; i++) {
                //i:(nTurtles-1-i) の内部点に i 番目のタートルを移動
                moveToDivPoint(hm[0], hm[nTurtles-1], i, nTurtles-1-i, hm[i]);
            }
        }
    }

    //n 匹のタートルの配列を作って返す。タートルは 2interval 間隔で配置され、f に表示される
    static Turtle[] createTurtles(int n, int interval, TurtleFrame f){
        Turtle[] hm = new Turtle[n];
        for (int i = 0; i < n; i++) {
            hm[i] = new Turtle(2*interval*i,interval,0);
            f.add(hm[i]);
        }
        return hm;
    }

    //タートル m を使って半径 radius の正 steps 角形の 1 辺を描く
    static void circleStep(Turtle m, int radius, int steps) {
        m.fd((int)(Math.PI*2*radius/steps));
        m.rt(360/steps);
    }

    //タートル m を使って n 角星形の 1 部を描く。辺長は 2*radius。steps 回の分割の i 回目。
    static void starStep(Turtle m, int radius, int steps, int n, int i) {
        if (i % (steps/n) == 0) { m.rt(360*2/n); } //(steps/n) 回に一度回転
        m.fd(2*n*radius/steps);
    }

    // turtle0 と turtle1 を m:n に内分した位置へ turtle2 を移動する
    static void moveToDivPoint(Turtle turtle0, Turtle turtle1,
                               int m, int n, Turtle turtle2) {
        turtle2.moveTo(divPoint(turtle0.getX(),turtle1.getX(),m,n),
                       divPoint(turtle0.getY(),turtle1.getY(),m,n),
                       0);
    }

    // x0 と x1 を m:n に内分した値を求める
    static int divPoint(int x0, int x1, int m, int n) {
        return (x0*n+x1*m)/(m+n);
    }

}
```

図 4: メソッドにより整理された morphing プログラム

- $i = 5$: i は a, b の公約数でない
- $i = 6$: i は a, b の公約数でない

となるので、最後に見つけた公約数である 3 が答えとなる。これをプログラムにすると次のようになる:

```
1  public class GCD1 {
2    public static void main(String[] args){
3        int a = 6, b = 21;
4        int lastCommonDivisor = 0; // 一番最近に見つかった公約数
5        for (int i = 1; i <= a; i++) {
6            if ( i は a,b の公約数? ) {
7                lastCommonDivisor = i; // i は公約数なので覚える
8            }
9        }
10       System.out.println("最大公約数は" + lastCommonDivisor);
11   }
12 }
```

変数lastCommonDivisor は、一番最近に公約数となったi の値を覚えておくためのものである。if 文によって、i が公約数であると分かる度に、そのときのi の値が代入される。

for 文は i を 1 からa まで順に変化させる。結果として for 文が終了したときのlastCommonDivisor には、一番最後に公約数となった値、つまり最大公約数が入っていることになる。

分割した残りの問題は次のようにして解くことができる:

- i は a,b の公約数であるとは、i がa の約数、かつ、i がb の約数である。つまり次のような式になる:

  i が a の約数 && i が b の約数

- i が a の約数であるとは、a をi で割った余りが 0 であること。つまり次のように書ける (a % i はa をi で割った余りを計算する式。==は値が等しいかどうかを比較する演算子):

  ```
  a % i == 0
  ```

練習 7-1: (最大公約数 1) 上の説明を読んで、実際に最大公約数を求めるプログラムを作れ。

2.1.2 アルゴリズム 2

アルゴリズム 1 の変型として、大きい順に公約数を探すこともできる。アルゴリズム 1 では 1 から a までの全ての数について調べるまで最大公約数を見つけられなかったが、大きい順に調べれば公約数を 1 つ見つけた時点で繰り返しを中断できる。

アルゴリズム 2: i を $a, a-1, \ldots, 2, 1$ の順に変化させ、i が a, b の公約数であるかを調べる。**最初**に公約数であると分かったときの i の値が最大公約数である。 ∎

これをプログラムにすると次のようになる:

```
1  public class GCD2 {
2    public static void main(String[] args){
3        int a = 6, b = 21;
4        int i = a;
5        while (![i は a と b の公約数?]) { //![式]は[式]の否定
6            i--;                          // i を 1 減らす
7        }
8        System.out.println("最大公約数は" + i + ".");
9    }
10 }
```

このプログラムでは、i が公約数**でない間**、i を 1 ずつ減らしている。i が公約数となると while 文の条件が false になり、繰り返しが終了する。結果として while 文が終了したときの i は**最初に見つかった**公約数になる。

練習 7-2: (最大公約数 2) 上のプログラムを完成させよ。

2.1.3 アルゴリズム 3(ユークリッドの互除法)

ここまでのアルゴリズムは単純に 1 つずつ数を調べてゆくものであったが、次の定理を用いるとより少ない計算回数で最大公約数を求めることができる。

定理 b を a で割った余りを r とする。a と b の最大公約数と r と a の最大公約数は等しい。

この定理 (証明は省略するが、難しくない) と「r が 0 のとき a は b の約数である」ことに注目すると、以下のようなアルゴリズムが得られる。

アルゴリズム 3(ユークリッドの互除法): b を a で割った余りを r とする。r が 0 でなければ、a, b をそれぞれ r, a に置き換えてその最大公約数を求める。r で 0 であれば、a が (最初の a, b の) 最大公約数。 ∎

例えば、143 と 1469 の最大公約数を求める場合であれば、以下のようにな計算になる:

- $b = 1469$ を $a = 143$ で割った余りは $r = 39$ なので 39, 143 を次の a, b とする
- $b = 143$ を $a = 39$ で割った余りは $r = 26$ なので 26, 39 を次の a, b とする
- $b = 39$ を $a = 26$ で割った余りは $r = 13$ なので 13, 26 を次の a, b とする
- $b = 26$ を $a = 13$ で割った余りは $r = 0$ なので 13 が最大公約数

これをプログラムにすると以下のようになる。

```
1  public class GCD3 {
2    public static void main(String[] args){
3        int a = 143, b = 1469;
4        while ([b を a で割り切れない?]) {
5            [次回の a を今回の b/a の余りにする。
             次回の b を今回の a にする。]
6
7        }
8        System.out.println("最大公約数は" + a + ".");
9    }
10 }
```

練習 7-3: (最大公約数 3) 上のプログラムを完成させよ。

注意: 5 行目を単純に (1) a=[b/a の余り]; (2) b=a; のように書くと、(2) は今回の a の値ではなく次回の a(つまり今回の b/a の余り) を代入してしまう。正しくは、もう 1 つの変数 r を用意して、(1) r=[b/a の余り]; (2) b=a; (3) a=r; のようにしなければいけない。(当然、変数 r を宣言する必要もある。)

2.2 素因数分解

問題: 整数 x の素因数分解、つまり $x = p_1 \cdot p_2 \cdots p_k$ となるような素数の組 $p_1, p_2, \ldots p_k$ を求める

この問題は、「素因数を 1 つ見つける」問題と、「素因数を全て見つける」問題に分割することができ、後者は次のようなアルゴリズムで解ける:

アルゴリズム (素因数分解): x が 2 以上のとき、x の素因数を 1 つ見けて p とする。次回の x を今回の x/p として、残りの素因数を見つける。x が 1 になれば全ての素因数が求められたことになる。 ∎

例えば、$x = 84$ の場合は次のようになる。

- $x = 84$ の素因数として 2 が見つかる。次回の x を $84/2 = 42$ とする
- $x = 42$ の素因数として 2 が見つかる。次回の x を $42/2 = 21$ とする
- $x = 21$ の素因数として 3 が見つかる。次回の x を $21/3 = 7$ とする
- $x = 7$ の素因数として 7 が見つかる。次回の x を $7/7 = 1$ とする
- $x = 1$ なので素因数はもうない

これをプログラムにすると次のようになる:

```
1  public class Factorize {
2    public static void main(String[] args){
3        int x = 84;
4        while (x > 1) {
5            int p;
6            x の素因数を 1 つ求め p に代入する
7            System.out.println("素因数:"+p);
8            x = x/p;                          //次回の x を x/p とする
9        }
10   }
11 }
```

`x の素因数を 1 つ求め p に代入する` 部分は、p を $2, 3, 4, \ldots$ と変化させ、x を割り切る**最初の**p を見つけることで解ける。つまり最大公約数の「アルゴリズム 2」とほとんど同じ形をしている。

練習 7-4: (素因数分解) 上のプログラムを完成させよ。また、自分の学生証番号を素因数分解してみよ。

2.3 曜日の計算

問題: 2004 年 11 月 29 日が何曜日であるかを計算する。
(1900 年 1 月 1 日が月曜日であった事実を利用する。)

この問題は次のように分割して解くことができる。

- 1900 年 1 月 1 日から目的の日 (2004 年 11 月 31 日) の間の日数を days 日とする。(両端の日も 1 日と数える。) days を 7 で割った余りが 0 であれば日曜日、1 であれば月曜日、2 であれば火曜日……(以下略) である。

```
1  public class Weekday {
2      public static void main(String[] args) {
3          int days = 0;
4          1900 年 1 月 1 日から目的の日までの日数を計算し、days に代入
5          int dayOfWeek = days%7; // 0:日曜日 ..  6:土曜日
6          System.out.println("曜日は"+dayOfWeek);
7      }
8  }
```

- 1900 年 1 月 1 日から目的の日のまでの日数を計算し days に代入する簡単な方法は、(年, 月, 日) を $(1900,1,1),(1900,1,2),(1900,1,3),\ldots$ のように 1 日ずつ変化させてゆき、目的の日 (2004,11,31) になるまで繰返すことである。プログラムにすると次のようになる:

```
loop:                                                    // 4.5 節参照
    for (int year = 1900; year <= 2002; year++) {
        for (int month = 1; month <= 12; month++) {
            int daysOfMonth;
            year 年 month 月の日数を計算し daysOfMonth に代入
            for (int date = 1; date <= daysOfMonth; date++) {
                days++;                                  // 日数を 1 増やす
                if ( year 年 month 月 date 日は目的の日か? )
                    break loop;                          // 4.5 節参照
            }
        }
    }
```

ここでは、1900 年 1 月 1 日から 2004 年 12 月 31 日までの繰返しを行い、途中で目的の日に達したら三重の繰返しを中断する break 文を使っている。(教科書 4.5 節参照)

- year 年 month 月の日数を計算し daysOfMonth に代入するには、月の日数が
  - 1,3,5,7,8,10,12 月は 31 日
  - 4,6,9,11 月は 30 日,
  - 2 月は、閏年でなければ 28 日, 閏年であれば 29 日

  であることから if 文を使って下のように書ける:

```
if ( month 月は大の月か? )
    daysOfMonth = 31;
else if ( month 月は (2 月を除く) 小の月か? )
    daysOfMonth = 30;
else if ( year 年は閏年か? )
    daysOfMonth = 29;
else
    daysOfMonth = 28;
```

練習 7-5: (曜日の計算) 教科書の練習問題 3.6 などを参考にして残りの空欄を埋め、曜日の計算プログラムを完成させよ。また、目的日を自分の生まれた日や、自分の来年の誕生日として曜日を計算せよ。

2.4 素数

問題: 正の整数 x が素数かを調べる

2.4.1 単純なアルゴリズム

単純には、2 から $(x-1)$ の間に x の約数があるかどうかを調べればよい。

アルゴリズム 1(素数): i を $2,3,4,\ldots,x-2,x-1$ と変化させ、x を i で割り切れるかどうか調べる。割り切れる数があれば x は素数でない。全ての数で割り切れなければ x は素数である。 ∎

工夫 1 a が x の約数ならば $a \leq \sqrt{x}$ である。よってアルゴリズム 1 は 2 から $\sqrt{x}$ 以下の整数について調べるだけでよい。

2.4.2　エラトステネスのふるい

素数 p が 1 つ見つかったとき、素数の倍数 $2p, 3p, \ldots$ は素数でない。そこで、最初は全ての数を素数の候補としておき、小さい順に候補を見てゆき、素数を見つける度に素数の倍数を候補から消してゆけば、生き残った候補が素数になる。

このアルゴリズムは紀元前の学者エラトステネスによって考られたもので「エラトステネスのふるい (the sieve of Eratosthenes)」と呼ばれる。

アルゴリズム 2(エラトステネスのふるい):

- 2 から x までのそれぞれの数が「素数**でない**」ことを示す表を作る。最初は全ての数が「素数**かも知れない**」としておく
- i を 2 から $x-1$ まで順に変化させ、
  - 表の i 番目が「素数でない」であれば、素数でないので次の i の繰返しを行う
  - 表の i 番目が「素数かも知れない」であれば、i は素数である。表の $2i, 3i, \ldots$ 番目を「素数でない」に変える
- 最後に表の x 番目が「素数かも知れない」のままであれば x は素数である

最初の数回について「素数でないこと示す表」が変化していくかを見ると次のようになる:

- 最初は全てが「素数かも知れない」である
  $2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, \ldots$
- 2 が素数と分かるので、2 の倍数 $(4, 6, 8, \ldots)$ を「素数でない」とする。(素数でないものに下線を引いて示す)
- $2, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, 9, \underline{10}, 11, \underline{12}, 13, \underline{14}, 15, \underline{16}, 17, \underline{18}, 19, \underline{20}, 21, \underline{22}, 23, \underline{24}, 25, \underline{26}, 27, \underline{28}, 29, \underline{30}, 31, \underline{32}, 33, \underline{34}, \ldots$
- 3 が素数と分かるので、3 の倍数 $(6, 9, 12, \ldots)$ を「素数でない」とする。
- $2, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, \underline{9}, \underline{10}, 11, \underline{12}, 13, \underline{14}, \underline{15}, \underline{16}, 17, \underline{18}, 19, \underline{20}, \underline{21}, \underline{22}, 23, \underline{24}, 25, \underline{26}, \underline{27}, \underline{28}, 29, \underline{30}, 31, \underline{32}, \underline{33}, \underline{34}, \ldots$
- 5 が素数と分かるので、5 の倍数 $(10, 15, 20, \ldots)$ を「素数でない」とする。
- $2, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, \underline{9}, \underline{10}, 11, \underline{12}, 13, \underline{14}, \underline{15}, \underline{16}, 17, \underline{18}, 19, \underline{20}, \underline{21}, \underline{22}, 23, \underline{24}, \underline{25}, \underline{26}, \underline{27}, \underline{28}, 29, \underline{30}, 31, \underline{32}, \underline{33}, \underline{34}, \ldots$
- 7 が素数と分かるので、7 の倍数 $(14, 21, 28, \ldots)$ を「素数でない」とする。
- $2, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, \underline{9}, \underline{10}, 11, \underline{12}, 13, \underline{14}, \underline{15}, \underline{16}, 17, \underline{18}, 19, \underline{20}, \underline{21}, \underline{22}, 23, \underline{24}, \underline{25}, \underline{26}, \underline{27}, \underline{28}, 29, \underline{30}, 31, \underline{32}, \underline{33}, \underline{34}, \ldots$

このように過去に見つかった素数の倍数を消してゆくことで、最初のアルゴリズムにあった無駄が省かれていることが分かる。

工夫 2　上のアルゴリズムでは i が素数であると分かったときに、表の $2i, 3i, 4i, \ldots$ を「素数でない」に変えていた。しかし、$2i, 3i, 4i, \ldots, (i-1)i$ は、i 未満の素数の倍数なので、すでに「素数でない」印がついているはずである。従って i が素数であると分かったときには、表の $i \cdot i, (i+1)i, (i+2)i, \ldots$ を「素数でない」とするだけでよい。

2.4.3 フェルマーのテスト

フェルマーの小定理: n が素数が、a が n 未満の自然数とする。このとき a^n を n で割った余りは a を n で割った余りと等しい。 ∎

この定理をもとに、次のようにして**確率的に**ある数が素数かどうかを調べることができる:

1. 整数 b の e 乗を m で割った余りを求めるメソッド `static int expmod(int b, int e, int m)` を定義する。

   この際、(b^e) を求めてからそれを m で割った余りを求めてはいけない。(b^e) は整数 (int) 型で扱える値 (最大 $(2^{31}-1)$) よりも大きくなることがあり、桁あふれを起こしてしまうためである。

   桁あふれを起こさずに計算するためには、下のように「b を掛ける度にm で割った余りを求める」計算をe 回繰り返せばよい。

   - `x = b % m;` $(b^1 \bmod m)$
   - `x = (x*b) % m;` $((b^1 \bmod m)b \bmod m = b^2 \bmod m)$
   - `x = (x*b) % m;` $((b^2 \bmod m)b \bmod m = b^3 \bmod m)$

   ⋮

2. 整数n が与えられたときに、2 以上n-1 以下の乱数整数を 1 つ作ってa とする。a のn 乗をn で割った余りを求め、それがa と等しければtrue、等しくなければfalse を答えるメソッド`static boolean fermatTest(int n)` を定義する。

   (「a のn 乗をn で割った余り」は`expmod(a,n,n)` で計算できる。)

3. 整数n が素数**らしいか**どうかを、times 回のフェルマーのテストで判定するメソッド`static boolean primeTest(int n, int times)` を定義する。このメソッドは次のような繰り返しをする:

   - `fermatTest(n)` が true であれば、素数である可能性が残るので、繰り返しを続ける
   - false であれば、素数でないので繰り返しを中断し false (素数でない) と答える
   - times 回繰り返しが続いた (つまり、その間に行った全ての`fermatTest(n)` が true だった) 場合は、素数らしいのでtrue と答える

工夫 3 上の`expmod(b,e,m)` はe 回の繰り返しをしている。e が偶数であれば $b^e = (b \cdot b)^{e/2}$ であり、さらに $b^e \bmod m = (b \cdot b)^{e/2} \bmod m$ であることを利用してより高速化が可能である。

参考のために、この性質を用いて b^e を高速に求めるメソッドを以下に示す。

```
static int exp(int b, int e) { // b の e 乗 (==r とする) を求める
   int x = 1;
   while (e > 0) {           //ここでは常に r == x*(b の e 乗) となっている
     if (e%2 == 0) {         //e が偶数なら: r == x*((b*b) の (e/2) 乗) と
       b = b * b;            //してよいので b を自乗し
       e = e / 2;            //e を半分にする
     } else {                //i が奇数の場合: r == (x*b)*(b の (e-1) 乗) と
       x = x * b;            //いえるので x を b 倍して
       e = e - 1;            //e を 1 減らす
     }
   }                         //繰り返しが終了したときは e==0 であるので
   return x;                 //r==x*(b の e 乗)==x*(b の 0 乗)==x なので x が解
}
```