CS 240 Homework 2

Alex Vondrak

Due: January 20, 2012

Consider the following functions:

$$
\begin{aligned}
f_0(n) &= n \\
f_1(n) &= \sqrt{n} \\
f_2(n) &= \log(n) \\
f_3(n) &= \log(\log(n)) \\
f_4(n) &= \log^2(n) \\
f_5(n) &= n/\log(n) \\
f_6(n) &= (1/3)^n \\
f_7(n) &= (3/2)^n \\
f_8(n) &= 17
\end{aligned}
$$

In this homework, you'll write a program to compare their relative growths. To do this, we can represent each function as an instance of an object. While this is an annoying way to do it, Java lacks features that make the task appreciably simpler. Below is most of the code we'll be using for the `Function` objects. The only missing part is the implementation of the `compareTo` method (see `http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`).

```
class Function implements Comparable {
   public static final int MIN_ID = 0;
   public static final int MAX_ID = 8;

   private int id;
   private double n;

   public Function(int id) {
      if (this.MIN_ID > id || id > this.MAX_ID) {
         throw new IllegalArgumentException("Function ID out of range: " + id);
      }
      this.id = id;
   }

   public double value() {
      switch(this.id) {
         case 0: return this.n;
         case 1: return Math.sqrt(this.n);
         case 2: return Math.log(this.n);
```

```
            case 3: return Math.log(Math.log(this.n));
            case 4: return Math.pow(Math.log(this.n), 2);
            case 5: return this.n / Math.log(this.n);
            case 6: return Math.pow(1.0 / 3.0, this.n);
            case 7: return Math.pow(3.0 / 2.0, this.n);
            case 8: return 17;
            default: throw new RuntimeException("Bad␣Function␣ID:␣" + this.id);
        }
    }

    public String toString() {
        return "f" + this.id;
    }

    public void setN(double n) {
        this.n = n;
    }

    public int compareTo(Object that) {
        /* IMPLEMENT ME */
    }
}
```

By giving a specific n, we can sort the functions f_0–f_8 by their values at that particular n. As we experiment with larger values of n, we'll see how the functions behave in the limit. However, this doesn't necessarily tell us which function is O of another. We don't know what constant multiples may be at play, nor how big we might have to make n before one function's value is finally larger than another's.

Input Format

Each line of input gives a Java **double** value for n.

Output Format

Sort the functions f_0–f_8 by their output values at the given n from smallest to largest. For each n, print out an array of the sorted functions. *Hint:* you'll enjoy using `java.util.Arrays`.

After all of these lines have been printed, print out a final line with the functions sorted such that each is O of the next. That is, you'll have to figure out the functions' asymptotic ordering for yourself and hard-code the output in the same format as the other lines.

Input Sample

```
1
10
100
1000
```

Output Sample

```
[f3, f2, f4, f6, f0, f1, f7, f8, f5]
[f6, f3, f2, f1, f5, f4, f0, f8, f7]
[f6, f3, f2, f1, f8, f4, f5, f0, f7]
[f6, f3, f2, f8, f1, f4, f5, f0, f7]
[LAST LINE ELIDED; FIGURE IT OUT]
```