Three Exercises and Three Questions

Adrian German

December 30, 2018

Abstract

My take on what items can benefit from the Rearrange Code format.

1 Introduction

1.1 You will [not] make it.

There once was an exercise in the early editions of "Computer Concepts with Java Essentials" by Cay Horstmann that was interesting and challenging. To my knowledge it has disappeared in later editions. I am here to argue that if the reason was "the exercise was too challenging" it is time for this exercise to make a comeback, via the new Rearrange Code InteraActivity. Below I explain how. In the process it will become clear that this format can be used for a wide class of problems specifically because of the amount of scaffolding it provides.

Let's start with the text of the problem:

> You don't know how to program decisions, but it turns out that there is a way to fake them using `substring`. Write a program that asks a user to input: (a) the number of gallons of gas in the tank, (b) the fuel efficiency in miles per gallon and (c) the distance the user wants to travel. Then print out
>
> ```
> You will make it
> ```
>
> or
>
> ```
> You will not make it.
> ```

The trick here[1] is to subtract the desired distance from the number of miles the user can drive, then find a way to turn the sign of the resulting expression

[1] This may have been provided with the exercise, I forgot. If it was, this is further evidence that the exercise was eliminated due to its "difficulty" level. I will revisit this idea later.

(positive or negative) into a switch that either includes (in the right place) or totally eliminates the `"not"` in the statement that the program prints.

To make the argument easier to follow let's denote by x the number you obtain by subtracting the desired distance from the number of miles the user can drive. Suppose further that you find a way of setting a value n to 1 if $x \geq 0$ and to 0 if $x < 0$. Then you can solve your problem as follows:

```
String answer = " not "; // note the spaces
System.out.println(
  "You will" +  answer.substring(0, 5 - 4 * n) +
  "make it"); // ^ sometimes not!
```

To calculate n we start by observing that

$$x + |x| = \begin{cases} 2x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Then we can divide by $2x$ to obtain n:

$$n = \frac{x + |x|}{2x} \tag{2}$$

However, we need to worry about x being 0 (zero) so we set:

$$n = \frac{x + |x| + \epsilon}{2x + \epsilon} \tag{3}$$

Here ϵ is a suitably chosen[2] constant that is small (e.g., $\epsilon = 10^{-6}$).

Then the relevant part of the solution becomes:

```
Scanner in = new Scanner(System.in);
```

```
double gallons = in.nextDouble();
       efficiency = in.nextDouble();
       distance = in.nextDouble();
```

```
double x = efficiency * gallons - distance;
       epsilon = 0.000001;
```

```
long n = Math.round( ( (x + Math.abs(x) ) * x + epsilon) /
                     (              2 * x * x + epsilon)
                   );
```

```
System.out.println("You will" +
                   " not ".substring(0, 5 - 4 * (int)n) +
                   "make it.");
```

The boxes indicate a possible "slicing" of the code.

[2]What makes a good ϵ value? If we think a little we realize ϵ need not be microscopic, and in fact as taken it's not, since 10^{-6} miles is 1.6 millimeters or about $\frac{1}{15}$ of an inch; ϵ can be as large as the car itself or even bigger, for example the order of magnitude of whatever could be considered "within walking distance".

1.2 Compute a Time Interval

Another very interesting problem has survived to this day and reads as follows:

> Write a program that reads two times in military format (e.g., `0900` and `1730`) and prints the number of hours and minutes between the two times. In the example shown the answer should be: 8 hours and 30 minutes (from 9am to 5:30pm). Extra credit if you can deal with the case that the first time is later in the day than the second time: (`1730`, `0900`). In this case the answer should be: 15 hours and 30 minutes, because that's how much time is between 5:30pm and 9am.

We first state our firm belief that no "extra credit" should be offered for any part of the problem. Instead the problem should ask that any solution should simply be able to process inputs just as described. Like in the previous problem the student is not allowed to use `if` statements. Here's a possible solution:

```
public class Two {
  public static void main(String[] args) {
    String a = args[0], b = args[1];
    int h1 = Integer.parseInt(a.substring(0, 2)),
        m1 = Integer.parseInt(a.substring(2)),
        h2 = Integer.parseInt(b.substring(0, 2)),
        m2 = Integer.parseInt(b.substring(2));
    int t1 = h1 * 60 + m1,
        t2 = h2 * 60 + m2;
    int d = (t2 - t1 + 24 * 60) % (24 * 60);
    System.out.println( d / 60 + " hours and " +
                        d % 60 + " minutes." );
  }
}
```

This solution is more suitable to a Rearrange Code format (and slices have been indicated clearly above) because: (a) it reads the input from the command line (which has not been covered at that stage in the text) and (b) it requires a fair amount of mathematical creativity to process all inputs correctly without using anything by arithmetical operators. So in this format we can present the student with pieces of ideas that fall inside (or on the outskirts of) the zone of "proximal development". In the example above the student can: (a) read ahead to determine the nature of `String args[]` and (b) try to produce an argument (or proof) as to why the calculation can proceed as indicated.

In our experience there always are students who don't pay attention to the problem as stated and make use of an `if` statement in the solution they submit. Their argument is that everybody knows `if` statements (from other languages or

simply by looking into the next chapter). In this case it's better to be proactive and offer examples of solutions that are not allowed. The best way to not waste them is to encode them in the Rearrange Code InterActivity format, and I give two examples collected over the years to show what one can come up with if one is determined to not follow the rules (albeit in a creative way):

```
String a = args[0], b = args[1];
int h1 = Integer.parseInt(a.substring(0, 2)),
    m1 = Integer.parseInt(a.substring(2)),
    h2 = Integer.parseInt(b.substring(0, 2)),
    m2 = Integer.parseInt(b.substring(2));
int count = 0;
for (int h = h1, m = m1; h != h2 || m != m2; ) {
  count += 1;
  m = (m == 59) ? 0 : m + 1;
  h = (m == 0) ? (h == 23) ? 0 : h + 1 : h;
}
System.out.print( count / 60 + " hours and " );
System.out.print( count % 60 + " minutes.\n" );
```

The solution above is very intuitive but still not acceptable[3] as a valid answer. However it is definitely accessible (and a good discussion starter once solved) in this sliced format.

1.3 Compute the Largest Value

A third problem asks the user to write a program that reads two integers and then calculate and report: (a) their sum, (b) their difference (in absolute value) and (c) the largest of the two. To make it clear they can't use `Math.max` we can ask them to rearrange the following program:

```
public class Math {
  public static void main(String[] args) {
    int n = Integer.parseInt( args[0] ),
        m = Integer.parseInt( args[1] );
    System.out.println( "max(" + n + ", " + m + ") = " +
                        java.lang.Math.max(n, m) ); // why?
  }
}
```

[3] It uses additional constructs in the language that have not been defined prior to the place in the book where this problem appears.

This activity will make them aware of two things: (a) if they choose the same name for the class as the name we chose (i.e., `Math`, which is a very intuitive name for this problem and therefore some students will legitimately think of it) and if they still want to calculate the largest of the two values by using the library function provided (i.e., `Math.max`) they will necessarily need to refer to it via the package and class name; (b) furthermore we can now explicitly state this restriction (that unlike this sliced InterActivity we expect a solution that only relies on arithmetic operators and no function calls).

Then we can hope they would submit something along these lines:

```
public class Four {
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]),
        m = Integer.parseInt(args[1]);
    int a = n + m;
    int b = Math.abs( n - m );
    int c = (a + b) / 2;
    System.out.println( n + " + " + m + " = " + a );
    System.out.println( "|" + n + " - " + m + "| = " + b );
    System.out.println( "max(" + n + ", " + m + ") = " + c );
  }
}
```

The same format can be used to present this solution:

```
class Khandelr { // communicated by Ramakant Khandel
  public static void main(String[] args) {
    int a = Integer.parseInt(args[0]),
        b = Integer.parseInt(args[1]);
    int c = a - b;
    int k = c >> 31 & 0x1;
    int max = a - k * c;
    System.out.println("max(" + a + ", " + b + ") = " + max);
  }
}
```

This solution[4] is definitely way outside of their "zone of proximal development" because little if any time is spent in the book with bitwise operators[5]. However it is very suitable to the "sliced code" format, in which students don't

[4]Communicated to me on Tue Jan 21, 2014, by one of my graduate assistants, Mr. Ramakant Khandel (IU username: `khandelr`, hence the name of the class).

[5]Not that the book should discuss these operators more but since they exist a reasonably extensive example could (or should) be given and this format is likely the most amenable to presenting those in an active learning setting.

write all the code but they are still forced (or, rather, given the opportunity) to own it at a certain level.

Here's another "solution" submitted on Thu Sep 08, 2016, by Sam Pilgrim (a student in C212 at the time). This solution is interesting enough to be presented, yet unlikely to be generated "from scratch" by a(ny) significant percentage of the students:

```
public class Sam {
  public static void main(String[] args) {
    int[] slots = new int[3];
    slots[0] = (int)(Math.random() * 100 - 50);
    slots[2] = (int)(Math.random() * 100 - 50);
    System.out.print( slots[0] + " vs. " + slots[2] + ": ");
    try {
      System.out.print(
        slots[
          1 + Math.abs(slots[2] - slots[0]) / (slots[2] - slots[0])
        ]
      );
    } catch (Exception e) { // if we divide by zero we come here
      // the values in slots[0] and slots[2] are equal
      System.out.print( slots[0] ); // or print slots[2], doesn't matter
    }
    System.out.println( " is the max value. " );
  }
}
```

You will notice that this strategy also solves the division by zero situation in the solution to the problem we started with.

2 How to Design Programs

There's an excellent book written by Matthias Felleisen[6] and his colleagues at Northeastern University in Boston that we use in C211 (our first course for majors). The book is entitled "How to Design Programs" (henceforth HtDP2e). Matthias is one of Dan Friedman[7]'s students and he graduated from IU in 1987. The book organizes the steps a student needs to take when solving a problem in a "design recipe". Central to the recipe is a notion of "template". Specifically

[6] https://htdp.org/2018-01-06/Book/

[7] https://www.sice.indiana.edu/all-people/profile.html?profile_id=205

the book explains that given a non-trivial problem, any problem, there is always a way to provide a solution in one of the following three ways: (a) structural recursion, (b) generative recursion or (c) accumulator passing style. I believe Matthias' design recipe and collection of templates could be more effectively communicated via Rearrange Code InterActivities. I give one example below.

2.1 Structural Recursion

The first step of the design recipe is to choose a data representation. The simplest type of arbitrarily large data is the structurally recursive type of data. A little thinking will convince anyone of the ubiquitous veracity of this statement[8], and when your data/input is recursive, the easiest, most natural approach is for your code to emulate the structure of your data. For example, if asked to add the digits in a number one could write:

```
public class Five {
  public static void main(String[] args) {
    System.out.println(sum(args[0]));
  }
  // A Digit is one of:
  //   -- "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
  // A Number is one of:
  //   -- Digit
  //   -- Digit + Number (concatenation)
  // sum : Number -> Integer (signature)
  public static int sum(String n) {
    if (n.length() == 1)
      return digit(n); // non-empty string
    else  // ^ base case
      return digit(n) + sum(n.substring(1));
  } // induction step: ^ promise
  // digit : Digit -> Integer (signature)
  public static int digit(String n) {
    return n.charAt(0) - '0'; // simple atomic template
  }
}
```

[8] If you add a peanut to a serving of peanuts you get (another, slightly larger) serving of peanuts; if you add a brick to a pile of bricks you get (another, slightly larger) pile of bricks; if you go trick or treat-ing on Halloween and you have a purse of candies and you add to it one more piece of candy that your neighbor gave you, you still have a purse of candies; and finally, to change a little, a walk is a step plus (another, slightly shorter) walk.

2.2 Generative Recursion

Structural Recursion is a very powerful[9] special case of Generative Recursion. Generative Recursion is when your approach is recursive because your ideas are recursive in that you can describe the solution to a problem as a combination of solutions of problems of the same kind only smaller. Examples include: generating permutations, mergesort, quicksort, the tower of Hanoi, calculating Fibonacci numbers via their definition, etc.

2.3 Accumulator Passing Style

Accumulator passing style (APS) is a type of generative recursion that builds a memory of what has been done and passes it around in the arguments of the recursive call. The transition from APS to `for`/`while` loops is immediate. The type of problems solved with APS include tree and graph traversal (breadth-first, depth-first) searching for a(n optimal) solution in a space of solutions etc.

3 Conclusion

So I essentially have three questions within the context described above:

(a) Can interesting problems from prior editions (like the first one presented here) come back as Rearrange Code InterActivities? Why did they vanish?

(b) Can we (as users/instructors) create our own items inside the textbook (of this kind and the other four)? Can we annotate the text for our students?

(c) What steps would be necessary to turn a different book (say, like HtDP2e) into a VitalSource text (complete with InterActive items)?

With many thanks and all best wishes,

Sincerely,

[9] Its template includes the case where structures are mutually recursive as is the case, for example, with grammars but not just with them.