Handout 4
CSCI 136: Spring 2017
20 February

Lab 3

Due 27 February

Recursion, Recursion, Recursion, ...

1 Prelab

Given the structure of this lab, a design doc is not required this week. However, as always, you should read through the lab carefully and think about how you will structure your solutions. **Also, you should come to lab with a considered design for each warm-up problem described below.**

2 Overview

This week's lab is structured as several small problems that can be solved in isolation. Recursion can be a difficult concept to master and one that is worth concentrating on separately before using it in large programs. The goals of this lab are:

- To practice writing recursive programs; and
- To solve a variety of interesting algorithmic problems.

Each problem will have a fairly short solution, but that doesn't mean you should put this assignment off until the last minute! Recursive solutions can often be formulated in just a few concise, elegant lines, but they can be very subtle and hard to get right.

Recursion is a tricky topic so don't be dismayed if you can't immediately sit down and code these perfectly the first time. Take time to figure out how each problem is recursive in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the same problem. You will need to depend on a recursive "leap of faith" to write the solution in terms of a problem you haven't solved yet. Be sure to take care of your base case(s) lest you end up in infinite recursion.

The great thing about recursion is that once you learn to think recursively, recursive solutions to problems seem very intuitive. Spend some time on these problems and you'll be much better prepared when you are faced with more sophisticated recursive problems.

3 Warm-ups

To help get you started, we present two warm-up problems. You should **come to lab on Wednesday with a considered design for each one**. We will discuss them briefly during lab. Try to work through them by yourself, and if you get stuck, ask for help from me or the TAs. Feel free to discuss the details of the warm-ups with other students as well. The goal of the warm-ups is to practice recursion fundamentals before lab on Wednesday.

3.1 Digit Sum

Write a recursive method `digitSum` that takes a non-negative integer and returns the sum of its digits. For example, `digitSum(1234)` returns $1+2+3+4 = 10$. Your method should take advantage of the fact that it is easy to break a number into two smaller pieces by dividing by 10 (i.e., $1234/10 = 123$ and $1234\%10 = 4$).

For these methods, we do not need to construct any objects. Therefore, you can declare them to be `static` methods and call them directly from `main`:

```
public static int digitSum(int n) { ... }
```

3.2 Subset Sum

Subset Sum is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to the target number. For example, given the set $\{3, 7, 1, 8, -3\}$ and the target sum 4, the subset $\{3, 1\}$ sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this method is:

```
public static boolean canMakeSum(int setOfNums[], int targetSum)
```

Assume that the array contains `setOfNums.length` numbers (i.e., it is completely full). Note that you are not asked to print the subset members, just return true/false. You will likely need a wrapper method to pass additional state through the recursive calls. What additional state would be useful to track?

4 Lab Programs

For each problem below, you must thoroughly test your code to verify it correctly handles all the necessary cases. For example, for the "Digit Sum" warm-up, you could use test code to call your method in a loop on the first 50 integers or use a loop to allow the user to repeatedly enter numbers that are fed to your method until you are satisfied. Testing is necessary to be sure you have handled all the different cases. You can leave your testing code in the file you submit — there is no need to remove it.

For each exercise, we specify the method signature. **Your method must exactly match that prototype** (same name, same arguments, and same return type). You will want to add additional helper methods for a number of these questions. **Your solutions must be recursive**, even if you can come up with an iterative alternative.

IMPORTANT: Before starting, copy the starter files, as described in Section 5.

4.1 Counting Cannonballs

Spherical objects, such as cannonballs, can be stacked to form a pyramid with one cannonball at the top, sitting on top of a square composed of four cannonballs, sitting on top of a square composed of nine cannonballs, and so forth. Write a recursive method `countCannonballs` that takes as its argument the height of a pyramid of cannonballs and returns the number of cannonballs it contains. The prototype for the method should be as follows:

```
public static int countCannonballs(int height)
```

4.2 Palindromes

Write a recursive method `isPalindrome` that takes a string and returns true if it is the same when read forwards or backwards. For example,

```
isPalindrome("mom")    →  true
isPalindrome("cat")    →  false
isPalindrome("level")  →  true
```

The prototype for the method should be as follows:

```
public static boolean isPalindrome(String str)
```

You may assume the input string contains no spaces.

4.3 Balancing Parentheses

In the syntax of most programming languages, there are characters that occur only in nested pairs, called bracketing operators. Java, for example, has these bracketing operators:

```
( . . . )
[ . . . ]
{ . . . }
```

In a properly formed program, these characters will be properly nested and matched. To determine whether this condition holds for a particular program, you can ignore all the other characters and look simply at the pattern formed by the parentheses, brackets, and braces. In a legal configuration, all the operators match up correctly, as shown in the following example:

```
{ ( [ ] ) ( [ ( ) ] ) }
```

The following configurations, however, are illegal for the reasons stated:

`( ( [ ] )`	The line is missing a close parenthesis.
`) (`	The close parenthesis comes before the open parenthesis.
`{ ( } )`	The parentheses and braces are improperly nested.

Write a recursive method

```
public static boolean isBalanced(String str)
```

that takes a string `str` from which all characters except the bracketing operators have been removed. The method should return `true` if the bracketing operators in `str` are *balanced*, which means that they are correctly nested and aligned. If the string is not balanced, the method returns `false`. Although there are many other ways to implement this operation, you should code your solution so that it embodies the recursive insight that a string consisting only of bracketing characters is balanced if and only if one of the following conditions holds:

- The string is empty.
- The string contains "`()`", "`[]`", or "`{}`" as a substring and is balanced if you remove that substring.

For example, the string "`[(){}]`" is shown to be balanced by the following chain of calls:

```
isBalanced("[(){}]")   →
 isBalanced("[{}]")    →
  isBalanced("[]")     →
   isBalanced("")      →  true
```

4.4 Substrings

Write a method

```
public static void substrings(String str)
```

that prints out all subsets of the letters in `str`. Example:
    `substring("ABC")` → "", "A", "B", "C", "AB", "AC", "BC", "ABC"
Printing order does not matter. You may find it useful to write a helper method

```
public static void substringHelper(String str, String soFar)
```

that is initially called as `substringHelper(str, "")`. The variable `soFar` keeps track of the characters currently in the substring you are building. To process `str` you must: 1) build all substrings containing the first character (which you do by including that character in `soFar`), and 2) build all substrings not including the first character. When `str` has no more characters in it, it will be one possible substring.

4.5 Print in Binary

Computers represent integers as sequences of bits. A bit is a single digit in the binary number system and can therefore have only the value 0 or 1. The table below shows the first few integers represented in binary:

binary	decimal
0	0
1	1
1 0	2
1 1	3
1 0 0	4
1 0 1	5
1 1 0	6

Each entry in the left side of the table is written in its standard binary representation, in which each bit position counts for twice as much as the position to its right. For instance, you can demonstrate that the binary value 110 represents the decimal number 6 by following this logic:

$$\begin{array}{rccccccc} \text{place value} \rightarrow & 4 & & 2 & & 1 & & \\ & \times & & \times & & \times & & \\ \text{binary digits} \rightarrow & 1 & & 1 & & 0 & & \\ & \downarrow & & \downarrow & & \downarrow & & \\ & 4 & + & 2 & + & 0 & = & 6 \end{array}$$

Basically, this is a base-2 number system instead of the decimal (base-10) system we are familiar with. Write a recursive method

```
public static void printInBinary(int number)
```

that **prints** the binary representation for a given integer. For example, calling `printInBinary(3)` would print `11`. Your method may assume the integer parameter is always non-negative.

Hint: You can identify the least significant binary digit by using the modulus operator with value 2 (ie., `number % 2`*). For example, given the integer 35, the value* $35\%2 = 1$ *tells you that the last binary digit must be 1 (ie., this number is odd), and division by 2 gives you the remaining portion of the integer (17).*

You will probably want to use the method `System.out.print` in the problem. It is just like `println`, but does not follow the output with a new line.

4.6 Extending Subset Sum

You are to write two modified versions of the `canMakeSum` method:

- Change the method to print the members in a successful subset if one is found. Do this without adding any new data structures (i.e. don't build a second array to hold the subset). Just use the unwind of the recursive calls.

  ```
  public static boolean printSubsetSum(int nums[], int targetSum)
  ```

- Change the method to report not just whether any such subset exists, but the count of all such possible subsets. For example, in the set shown earlier, the subset 7, -3 also sums to 4, so there are two possible subsets for target 4. You do not need to print all of the subsets.

  ```
  public static int countSubsetSumSolutions(int nums[], int targetSum)
  ```

5 Getting Started

We provide basic starter code for this assignment. To obtain it, execute the following command to copy the recursion lab folder to your current directory (note: the ending period in this statement is important):

```
cp -r /opt/mac-cs-local/share/cs136/labs/recursion .
```

The `recursion` directory contains the following file:

- `Recursion.java`: All of your code will go into this file.

6 Submission

Place the file `Recursion.java` inside of your `<unix>-lab3` directory (with `<unix>` replaced by your Williams Unix), and submit it in the drop-off folder for your lab section.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Since we are learning about running times and big-O analysis in lecture, for this lab, **include the running time (in big-O notation) for each method in the comments above the method**.