CIS 120 Data Structures and Algorithms — **University of Wisconsin-Stevens Point**

Lab 04: Recursion: Sum of Digits, Fibonacci Numbers [i]

Due Date: as specified in the course schedule.

The source files for this lab have been put together in an Eclipse project entitled **Lab04-Source Files** available in a Module called *Lab 04 Recursion* in the content section of D2L. Instructions for importing that project are contained in a separate document.

NOTE: As before, be sure to add a "purpose" statement above each non-trivial method. Add pre- and post-condition statements for each non-trivial method.

Part 1: Counting and Summing Digits in an Integer

The problem of counting the digits in a positive integer or summing those digits can be solved recursively. For example, to count the number of digits:

- If the integer is less than 10 there is only one digit (the base case).
- Otherwise, the number of digits is 1 (for the units digit) plus the number of digits in the rest of the integer (what's left after the units digit is taken off). For example, the number of digits in 3278 is 1 + the number of digits in 327.

The following is the recursive algorithm implemented in Java.

```
public int numDigits (int num)
{
    if (num < 10)
      return (1);    // a number < 10  has only one digit
    else
      return (1 + numDigits (num / 10));
}
```

Note that in the recursive step, the value returned is 1 (counts the units digit) + the result of the call to determine the number of digits in *num / 10*. Recall that *num/10* is the quotient when *num* is divided by 10 so it would be all the digits except the units digit.

The file DigitPlay.java contains the recursive method *numDigits* (note that the method is static -- it must be since it is called by the static method main). Copy this file to your directory, compile it, and run it several times to see how it works. Modify the program as follows:

1. Add a static method named *sumDigits* that finds the *sum* of the digits in a positive integer. Also add code to main to test your method. The algorithm for *sumDigits* is very similar to *numDigits*; you only have to change two lines!

2. Most identification numbers, such as the ISBN number on books or the Universal Product Code (UPC) on grocery products or the identification number on a traveler's check, have at least one digit in the number that is a *check digit*. The check digit is used to detect errors in the number. The simplest check digit scheme is to add one digit to the identification number so that the sum of all the digits, including the check digit, is evenly divisible by some particular integer. For example, American Express Traveler's checks add a check digit so that the sum of the digits in the id number is evenly divisible by 9. United Parcel Service adds a check digit to its pick up numbers so that a weighted sum of the digits (some of the digits in the number are multiplied by numbers other than 1) is divisible by 7.

   Modify the main method that tests your sumDigits method to do the following: input an identification number (a positive integer), then determine if the sum of the digits in the identification number is divisible by 7 (use your sumDigits method but don't change it -- the only changes should be in main). If the sum is not divisible by 7 print a message indicating the id number is in error; otherwise print an ok message. (FYI: If the sum is divisible by 7, the identification number could still be incorrect. For example, two digits could be transposed.) Test your program on the following input:

- 3429072 --- error
- 1800237 --- ok
- 88231256 --- ok
- 3180012 --- error

DELIVERABLES for Part 1: Printouts of DigitPlay.java and screenshots of the output demonstrating the test data.

Part 2: Efficient Computation of Fibonacci Numbers

The *Fibonacci* sequence is a well-known mathematical sequence in which each term is the sum of the two previous terms. More specifically, if fib(n) is the n^{th} term of the sequence, then the sequence can be defined as follows:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)   n>1
```

1. Because the Fibonacci sequence is defined recursively, it is natural to write a recursive method to determine the n^{th} number in the sequence. File Fib.java contains the skeleton for a class containing a method to compute Fibonacci numbers. Save this file to your directory. Following the specification above, fill in the code for method *fib1* so that it recursively computes and returns the n^{th} number in the sequence.

2. File TestFib.java contains a simple driver that asks the user for an integer and uses the *fib1* method to compute that element in the Fibonacci sequence. Use TestFib to test your *fib1* method. First try small integers, then larger ones. You'll notice that the number doesn't have to get very big before the calculation takes a very long time. The problem is that the *fib1* method is making lots and lots of recursive calls. To see this, add a print statement at the beginning of your *fib1* method that indicates what call is being computed, e.g., "In fib1(3)" if the parameter is 3. Now run TestFib again and enter 5 -- you should get a number of messages from your print statement. Examine these messages and figure out the sequence of calls that generated them. (This is easiest if you first draw the call tree on paper. Since it is part of the deliverables for this assignment, take a moment NOW and draw out the calls on a piece of paper. HINT: turn the paper to "landscape" position. See the drawing of "combinations" – Figure 4-3 on page 284 of your textbook as an example.) Since fib(5) is fib(4) + fib(3),you should not be surprised to find calls to fib(4) and fib(3) in the printout. But why are there two calls to fib(3)? Because both fib(4) and fib(5) need fib(3), so they both compute it -- very inefficient. Run the program again with a slightly larger number and again note the repetition in the calls.

3. The fundamental source of the inefficiency is not the fact that recursive calls are being made, but that values are being recomputed. One way around this is to compute the values from the beginning of the sequence instead of from the end, saving them in an array as you go. Although this could be done recursively, it is more natural to do it iteratively. Proceed as follows:

    a. Add a method *fib2* to your Fib class. Like *fib1*, *fib2* should be static and should take an integer and return an integer.
    b. Inside *fib2*, create an array of integers one larger than the size of the value passed in (to be able to return the "nth" Fibonacci number).
    c. Initialize the first two elements of the array to 0 and 1, corresponding to the first two elements of the Fibonacci sequence. Then loop through the integers up to the value passed in, computing each element of the array as the sum of the two previous elements. When the array is full, its last element is the element requested. Return this value.

d. Modify your TestFib class so that it calls *fib2* (first) and prints the result, then calls *fib1* and prints that result. You should get the same answers, but very different computation times.

e. To verify the difference in computation times:
   1. write a loop that will output the first 15 Fibonacci numbers using Fib1, and then
   2. Copy the code but change the method call from *fib1* to *fib2.* You now have two loops generating the first 15 Fibonacci numbers, the first doing so recursively, the second iteratively.
   3. Using the timing technology shown below, determine the elapsed time of both loops. Write output statements to the screen revealing the different values.

DELIVERABLES for Part 2: (1) Printouts of the TestFib and Fib classes. (2) screen shots of the output of TestFib. (3) Paper and pencil drawing of the call tree for the recursive Fib(5). (4) A word doc that explains any difference in the elapsed times of the recursive vs iterative versions of Fib.

How to measure the elapsed time in a Java program:

From: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#nanoTime()

nanoTime

public static long **nanoTime**()

Returns the current value of the most precise available system timer, in nanoseconds.

This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years (2^{63} nanoseconds) will not accurately compute elapsed time due to numerical overflow.

For example, to measure how long some code takes to execute:

```
long startTime = System.nanoTime();
  // ... the code being measured ...
long elapsedTime = System.nanoTime() - startTime;
```

Returns:

The current value of the system timer, in nanoseconds.

Hardcopy to hand in:

1. Cover sheet
2. Deliverables from Part 1, sumDigits.
3. Deliverables from Part 2, Fibonacci.

This project will also be evaluated via electronic submission

1. Upload the project file for this assignment to the appropriate dropbox in D2L. Instructions for exporting that project from Eclipse are contained in a separate document. Please drop the Word doc there as well.

[i] Portions taken from http://cs.roanoke.edu/~cpsc/AW/labs/labs.html.