Programming Assignment: recursion, recursion, recursion

Due date:
Thursday, July 26 at 11:29 am

For this project, you will implement several of static methods in a class called Recursive. You will be provided with a file containing the “stubs” for the methods.

All of the individual tasks (which are mostly independent of each other) build on some kind of recursive concept/definition. However, the methods you write may apply these concepts through recursive or non-recursive Java methods. You may of course add (private) helper methods as you see fit.

Part I

Generating all bit strings of length $k$ in increasing order of their numerical value.

Below are the length-3 bitstrings.

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Complete the method bitStrings as below. The returned ArrayList contains the length $k$ bitstrings in the specified order.

Hint: in the above example, suppose you were given the bitstrings of length 2; how would you use them to construct all bitstrings of length 3?

```java
public static ArrayList<String> bitStrings(int k) {

}
```

Part II

Generating Grey Codes of a given length $k$.

Now you want to produce the same set of bit strings, but in a different order. A sequence is a grey code if each string differs from the preceding string in only one bit position. (If you are familiar with Karnaugh Maps, you have seen grey codes to label rows and in the map).

Examples (with hint):

for 1-bit:

0
for 2-bits:

0 0
0 1
1 1
1 0

for 3-bits:

0 0 0
0 0 1 put 0 in front
0 1 1
0 1 0
1 1 0
1 1 1 reverse sequence for
1 0 1 2-bits and put 1 in
1 0 0 front

Write the method `greyCodes()` which produces a list of length-k bit-strings in grey code sequence. Hint: see above for how to create a grey code sequence for k bits from a grey code sequence for k − 1 bits.

```java
public static ArrayList<String> greyCodes(int k) {
    // Method implementation
}
```

Part III

Using Pascal’s recurrence to generate all subsets of size k using bit-string representation.

Given a set of size n, a subset can be represented as a bit string of length n.

For example, if \( S = \{a_1, a_2, a_3, a_4\} \), then

- 1111 represents \( S \) itself.
- 0000 represents the empty set
- 0101 represents the set \( \{a_2, a_4\} \)

and so on.

For this part of the assignment, you are to write a Java method which takes \( n \) and \( k \) and produces an ArrayList of all bitstrings represent subsets of size exactly \( k \).
For example, if \( n = 4 \) as above and \( k = 2 \), your method would return an ArrayList containing the following strings:

1100
1010
1001
0110
0101
0011

Your method should use Pascal’s recurrence to produce this list recursively. The bitstrings should be in the same order implied by the example above – reverse lexicographic order.

Here is the method stub:

```java
public static ArrayList<String> sizeKsubsets(int n, int k) {
}
```

Part IV

Generating all palendromes of length \( 2k \) or less from the characters ’a’, ’b’, ’c’

Every palendrome must appear exactly once in the ArrayList returned.

Note: the empty string is a palendrome of length 0 (\( k = 0 \))

Update: the problem should have been specified as: “given \( k \), generate all palendromes of length \( k \) or less” (rather than \( 2k \)).

You can interpret it either way, however the point here is that

```
   a b a
   a b c b a
   a c c c a
   c b a a b c
```

```java
public static ArrayList<String> palendromes(int k) {
}
```

Update: the problem should have been specified as: “given \( k \), generate all palendromes of length \( k \) or less” (rather than \( 2k \)).

You can interpret it either way and we will figure it out when grading, however the point here is that palendromes can be of odd or even length, so even if you use the original spec, you will still have to
include odd-length palendromes. For example, if $k = 3$, then you would generate palendromes of length 6 or less and this includes palndromes of length 5.

    Sorry for the confusion...

    Update 2: Yes, I spelled palindrome incorrectly. But let’s stick with the incorrect spelling.

Part V:

Generating all legal (balanced) strings of length $2k$ or less according to the following rules:

    Every legal string must appear exactly once in the returned ArrayList.

    • The empty string is balanced
    • If $S$ is a balanced string, then $(S)$ and $\{S\}$ are balanced strings
    • if $S$ and $T$ are both balanced strings then $(ST)$ and $\{ST\}$ are also a balanced strings.

    NOTE: the third rule is a change from a previous version of this assignment. It performs concatenation of two legal strings, but requires that the resulting string be “bracketed” by either paren’s of curly braces.

    This disallows strings like “(())()”. Instead you could have “((())())” or “(())(()())” (for example) and each of these would be considered distinct.

    This might seem awkward, but it has an advantage: there is only one derivation of any legal string.

Think of it like a fully parenthesized arithmetic expression like $((a + b) + c)$ instead of $a + b + c$.

    If you’ve already done the method with the old specification, that is ok ... we’ll sort it out.

    public static ArrayList<String> nesters(int k){
    }

Part VI:

Three ways of computing Fibonacci Numbers.

    The Fibonacci function is defined on integers as (See Rosen):

    \[
    f(0) = 0 \quad f(1) = 1 \quad f(n) = f(n-1) + f(n-2) \quad \forall n \geq 2
    \]

    The Fibonacci function grows fast and pretty quickly, $f(n)$ exceeds the largest value we can represent with a Java int. We want to be able to compute $f(n)$ for even larger values of $n$, so we will not use the Java int (or Integer) type to represent Fibonacci numbers. Instead, we will use something called BigInteger, which can represent arbitrarily large integers.

    You will implement three different approaches to computing $f(n)$ each of which takes an int $n$ and returns a BigInteger equal to $f(n)$.

The BigInteger class has many methods in it, but for our purposes, you will just need to understand a few things:
// constants
BigInteger.ZERO

BigInteger.ONE

// constructor taking string rep of an integer
BigInteger(String val)

BigInteger multiply(BigInteger val)

BigInteger add(BigInteger val)

You will implement three Java methods computing f(n):

**Approach A:** First you will implement a “naive” recursive method for computing f(n). This method is a direct translation of the recursive definition of f(n) into Java. It will be called `fibA`.

**Approach B:** The second approach is smarter.

- Start with f(0) and f(1) stored in variables.
- From these, you can compute f(2).
- From f(1) and f(2), you can compute f(3).
- In general, within a loop, you compute f(i) from f(i − 2) and f(i − 1) computed from previous iterations.

**Approach C:** An even smarter approach!

This approach involves matrices (don’t worry, you don’t need to have taken a linear algebra course). We will only need 2-by-2 matrices and 2-by-1 column vectors.

First, multiplication of two 2-by-2 matrices is defined as:

\[
\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}
\]

Similarly, multiplication of a 2-by-2 matrix with a 2-by-1 column vector is defined as:

\[
\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}
\]

Now, consider the following (where f() is the Fibonacci function):

\[
\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f(0) \\ f(1) \end{pmatrix} = ?
\]

What does this equal?

\[
\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f(0) \\ f(1) \end{pmatrix} = \begin{pmatrix} f(1) \\ f(0) + f(1) \end{pmatrix} = \begin{pmatrix} f(2) \end{pmatrix}
\]

Or how about this:
\[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix} \cdot \begin{pmatrix}
f(1) \\
f(2)
\end{pmatrix} = \begin{pmatrix}
f(2) \\
f(1) + f(2)
\end{pmatrix} = \begin{pmatrix}
f(3)
\end{pmatrix}
\]

But, \[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix} \cdot \begin{pmatrix}
f(1) \\
f(2)
\end{pmatrix} = \begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix} \cdot \begin{pmatrix}
f(0) \\
f(1)
\end{pmatrix} = \begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^2 \cdot \begin{pmatrix}
f(0)
\end{pmatrix}
\]
In general, we have \[
\begin{pmatrix}
f(n) \\
f(n+1)
\end{pmatrix} = \begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^n \cdot \begin{pmatrix}
f(0)
\end{pmatrix}
\]

So, what good does this do us? Don’t we still have to do \(n - 1\) matrix multiplications? 

(Reading: See section 2.4 of Weiss (page 47) for a related discussion on efficient scalar exponentiation.)

Not necessarily. Consider computing \[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^8
\]
We can rewrite it as:
\[
\begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^8 = \begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^4 \cdot \begin{pmatrix}
0 & 1 \\
1 & 1
\end{pmatrix}^4
\]

Use this idea so that \texttt{fibC} performs only \(O(\log n)\) \texttt{BigInteger} multiplications.

Here are the method stubs:

```java
public static BigInteger fibA(int n) {
}

public static BigInteger fibB(int n) {
}

public static BigInteger fibC(int n) {
}
```

**Report**

Submit a short report with your code which answers the following:

1. What is the largest \(n\) for which a normal Java \texttt{int} can represent \(f(n)\) – i.e., the largest \(n\) such that \(f(n) < \texttt{Integer.MAX\_VALUE}\)?

2. For each of the Fibonacci algorithms, what is the largest \(n\) for which you can compute \(f(n)\) in:
• 15 seconds
• 30 seconds
• 60 seconds

You will use `System.nanoTime` to help you estimate elapsed time (a short tutorial coming soon).

Submit your report with your code as a file called `report.pdf` or `report.txt`