COMP 250 - Homework #3 Mathieu Blanchette

Due on October 25th 2015, 23:59, via MyCourses.

Question 1 (55 points)

In this question, you will learn how the Java Virtual Machine (JVM) uses a stack to keep track of the value of variables during the execution of recursive methods. As you know already, when a method is executed, its arguments and local variables are allocated on a part memory we called the stack. Why is it called "stack"? Well, because it is a Stack, in the sense seen in class recently, i.e. a LIFO data structure. What exactly does the JVM do when it reaches a recursive call (or in fact any method call)? If creates a new frame with the values of the arguments and local variables of the method being called, pushes them on the Stack and then starts operating on those values. When the computation of the method is over, it pops the frame from the stack and returns the value it calculated (unless its return type is void). How does it know where to return? The information is all there, on top of the stack! Those are the values of the local variables of the method before it made the recursive call. How does it know at what point of the method it was at? That's stored in an additional variable stored in the program frame named program counter (PC).

I found the following video quite useful:
http://www.youtube.com/watch?v=k0bb7UYy0pY

Consider the following recursive method to compute the n-th Fibonaaci number:

```
                int Fibonacci(n) {
PC=1                 if (n<=1) then return n;
PC=2                 int firstFib = Fibonacci(n-1);
PC=3                 int secondFib = Fibonacci(n-2);
PC=4                 return firstFib + secondFib;
                }
```

The Java code available at
http://www.cs.mcgill.ca/~blanchem/250/hw3/FibonacciNonRecursive.java
implements a Fibonacci method that parallels the recursive method shown above but that uses a stack to mimic recursion. I recommend that you take a very good look at this before going further. Of course, there are much simpler ways to compute the Fibonacci numbers without using recursion, but this will serve as a useful example for what follows.

Now, it is your turn to use a stack to mimic recursion. Your task is to implement a non-recursive version of the **mergeSort** algorithm seen in class:

```
                Algorithm mergeSort ( A[], start, stop )
PC=1            if ( start < stop ) then
PC=2                 mid = (start + stop)/2
PC=3                 mergeSort(A, start, mid)
PC=4                 mergeSort(A, mid+1, stop)
PC=5                 merge(A, start, mid, stop)
```

For this, you will need to use a Stack, as I've done in the Fibonnaci code given above.

Start from the skeleton available at:
http://www.cs.mcgill.ca/~blanchem/250/hw3/MergeSortNonRecursive.java
and implement the mergeSort method. Your method must not be recursive and it must not change the header (arguments) of the method. The class ProgramFrame class can be modified if you want.

Notes:

- The java instructions "continue" and "break" will be quite useful. If you are not familiar with them, see http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html
- The Java method that implements the top() operation on a stack (i.e. looking at the element on top of the stack without removing it) is called peek().

Question 2. (15 points)

Consider the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 * T(n - 1) + n & \text{if } n > 1 \end{cases}$$

Obtain an explicit formula for the following recurrence using one of the techniques seen in class. Simplify your explicit formula as much as possible. In the process of doing so, you will possibly come across the summation $\Sigma_{i=1..n} (i * 2^i)$, which can be simplified as $2 * (1 + 2^n * (n-1))$.

Question 3. (15 points)

Consider the following "magic" trick. You have a deck of n cards, labeled 1, 2,..., n (but not necessarily in that order).

You then repeat the following process until no cards are left: (i) Show to the public the card on the top of the deck, and remove it from the deck, and (ii) Take the next card from the top of the deck and place it at the bottom of the deck, without showing it. Your goal is to have previously ordered the cards in the deck so that the cards shown to the public are in increasing order: 1, 2, ..., n. For example, if n=5, then starting from the arrangement 1,5,2,4,3 would work: 1,5,2,4,3 -> 2,4,3,5 -> 3,5,4 -> 4,5 -> 5

Question: Write an algorithm that prints the appropriate initial ordering for any given number n of cards.

Algorithm orderCards(n)
Input: An integer n
Output: Prints the correct card ordering.

Question 4. (15 points)

We have seen it class the precise meaning of the notation $f(n) \in O(g(n))$, essentially meaning that $f(n)$ grows at most as fast as a constant times $g(n)$. A similar notation can be used to say that $f(n)$ grows at least as fast a constant times $g(n)$, using the Omega notation:

$f(n) \in \Omega(g(n))$ if and only if $g(n) \in O(f(n))$.

Finally, we can express the fact that $f(n)$ grows exactly as fast as a constant times $g(n)$ using the Theta notation:

$f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Question: Prove that $\log(n!) \in \Theta(n \log(n))$