CSC236 – Problem Set 4 (updated again)

There are two components of this problem set: a preliminary question designed to check your understanding of the basic topics covered this week, and a set of more challenging questions designed to make you think critically about the material and apply it in new contexts. *Get in the habit of starting work early* – the less time you give yourself, the most stressed you’ll find yourself each week!

**Caution:** you must submit two separate files in two separate locations on MarkUs, one for the Preliminary and one for the Challenge.

To avoid suspicions of plagiarism, clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, you consulted at the beginning of your assignment submission.

---

**Preliminary: due February 6, 2014 8:00 pm**

This question is an opportunity for you to check your understanding of the topics and practice writing formal solutions. This is a valuable learning opportunity – if you see that you’re at a loss, get help quickly!

Your goal should not be to get the right answer, but to convince the marker that you know what you’re doing. This question is marked on the following 3-point scale:

<table>
<thead>
<tr>
<th>3: You’ve mastered this topic</th>
<th>1: You don’t really know what you’re doing</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: You’re almost there, but missing something</td>
<td>0: You didn’t submit/had absolutely no clue</td>
</tr>
</tbody>
</table>

This question must be completed INDIVIDUALLY.

Consider the following code, which is a recursive version of linear search. **Assume that list slicing is performed in constant time.** Analyse the asymptotic worst-case runtime of `rec_search`, by:

1. First finding a recursive definition for $T(n)$ the worst-case runtime of $\text{rec} \_\text{search}(A, x)$ when $\text{len}(A) = n$

2. Then using repeated substitution to find a closed form for $T(n)$. You do not need to prove that your closed form is correct.

3. Then converting that closed form into an asymptotic bound. This can be done in one line.

```python
1 def rec_search(A, x):
2     if len(A) == 0:
3         return False
4     else:
5         return A[0] == x or rec_search(A[1..len(A)-1], x) # INCLUDES A[len(A)-1]
```

Note that you’re finding the *worst-case* runtime, so you can assume the or in the last line never short-circuits, say because $x$ isn’t in $A$.  
Challenges: due February 8, 2014 noon

Answer each question completely, always justifying your claims and reasoning. Your solution will not just be graded on correctness, but on its clarity as well. Technically correct answers that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

1. The above implementation of `rec_search` used list slicing on slices of size \( n - 1 \), which in practice can take linear time, invalidating our assumption in the Preliminary.

   (a) [5] Rewrite `rec_search` so that it uses an index parameter, and does not use list slicing. However, your method must still be recursive – that is, simply handing in the standard iterative linear search will receive zero marks.

   In addition to your code, briefly explain what you changed, and why it works.

   Here is a generic template of how your code may be used:

   ```python
   1 def search(...):
   2     # Call your helper function
   3     ... my_rec_search(...) ...
   4     # Process the result of your helper function to return
   5     return ...
   6
   7 # YOU DEFINE THIS FUNCTION
   8 def my_rec_search(...):
   9     ...
   ```

   (b) [3] Show that the runtime of your code satisfies the same recurrence as the original `rec_search`.

2. Consider the following code, which computes the Fibonacci numbers.

   ```python
   1 def fib(n):
   2     if n <= 2:
   3         return 1
   4     else:
   5         return fib(n-1) + fib(n-2)
   ```

   (a) [2] Develop a recursive definition for \( T(n) \), the worst-case runtime of `fib` on input \( n \). You may assume that \( T(1) = T(2) \), and that all constants equal 1 (i.e., replace \( c \)'s and \( d \)'s with 1's in your definition).

   (b) [3] Show that `fib` runs in exponential time by showing that for some constant \( B > 1 \), \( T(n) = \mathcal{O}(B^n) \). You may use any results from previous Problem Sets and the course notes in your solution, without proof.

   Note that for your solution to be convincing, you should determine the value of \( B \). You do not need a “formal proof” of the fact that \( T(n) = \mathcal{O}(B^n) \), just some justification using what you know about Big-O.

   **UPDATE**: Someone asked in office hours about Big-Theta; since I didn’t really review this in class, just find the smallest \( B \) such that \( T(n) = \mathcal{O}(B^n) \). You should do this be finding a closed-form solution for your recurrence from part (a) (again, using previous results from the course). Hint: the answer isn’t \( T(n) = \mathcal{O}(2^n) \).

   (c) [2] Explain why the above code is so inefficient, without referring to your recurrence, or induction. That is, your explanation should be a concrete programmer one, with limited mathematics.