CSC236 Winter 2020
Assignment #2: recurrences & correctness
PREVIEW (posted 02/18)

WARNING: This document contains only the first 2 questions of assignment 2. The full assignment will have one additional question. It will be posted at a later date (during reading week), along with starter `.tex` source.

1. In lecture, we used the following recurrence to represent the steps taken by an implementation of mergesort on a list of size n:

$$T_0(n) = \begin{cases} 1 & \text{if } n = 1 \\ n + 2T_0(n/2) & \text{if } n > 1 \end{cases}$$

(This recurrence assumes n is a power of 2, hence the absence of floor and ceiling. You may maintain this assumption throughout this question.)

In reality, some implementations of divide-and-conquer algorithms stop the recursion before the input size becomes trivial. For example, a programmer may find that their mergesort implementation ends up running a bit faster if they stop recursing when the list size is less than 10, sorting these small lists using selection sort.

Consider the following recurrence, which models this scenario:

$$T(n) = \begin{cases} c & \text{if } n \leq k \\ n + 2T(n/2) & \text{if } n > k \end{cases}$$

$k, c \in \mathbb{N}^+$ are fixed constants, where k represents the largest problem size which is solved non-recursively, and c represents the cost of solving these small problems.

(a) Use unwinding[1] to find a closed form for $T(n)$ when $n \geq k$. (You do not need to prove that your closed form is correct, but it should be clear how you arrived at it.)

[1] **Logistical note:** If you wish to use tree diagrams for the unwinding portions of this question (parts (a) and (c)), you are welcome to include scanned hand-drawn images, or diagrams generated using other software. See this chapter of the LaTeX Wikibook for information on including images in LaTeX documents. You may also describe the solution tree without explicitly drawing it (a table may be helpful).

(b) What is the big-Θ complexity of $T(n)$? Does it depend on k? Briefly justify your answer (no proof required). You may not assume $n \geq k$ for this part.

(c) Rather than assigning a fixed cost to the $n \leq k$ case, it may be more realistic to use a function of n, since there are a range of input sizes which are handled non-recursively. Since selection sort is a $\Theta(n^2)$ algorithm, we'll define our new recurrence $T'(n)$ to be

$$T'(n) = \begin{cases} n^2 & \text{if } n \leq k \\ n + 2T'(n/2) & \text{if } n > k \end{cases}$$

Find a closed form for $T'(n)$ for $n \geq k$, and show how you got there. Rather than unwinding from scratch, you may find it simpler to build on your work from part (a).

(d) Is $T'(n) \in \Theta(T(n))$? Why or why not? Briefly justify your answer. As in part (b), you may not assume $n \geq k$.

2. Our boss has tasked us with writing a program to find the *unique* maximum of a non-empty list of positive integers. If there is no unique maximum, our program should signal this by returning a negative number. For example, on input `[5, 2, 1, 2]`, our algorithm should return 5. Given `[2, 1, 2]`, we may return -1, -2, -236, or any other negative number.

Below is our first attempt to solve this problem.[2]

```python
def umax(A):
    if len(A) == 1:
        return A[0]
    head = A[0]
    tail = A[1:]
    tmax = umax(tail)
    if head == tmax:
        return -1
    elif head > tmax:
        return head
    else:
        return tmax
```

(a) Based on the informal specification above, write precise pre- and post-conditions for umax. Your postcondition should use symbolic notation rather than restating the English description above ("find the unique maximum..."). The following postcondition was used in

[2] You can download the code for this question from http://www.cs.toronto.edu/~colin/236/W20/assignments/umax.py

lecture for the function `max`, which found the (not necessarily unique) maximum of a list. It may be a useful starting point:

$\texttt{max(A) = x}$ where $(\exists j \in \mathbb{N}, A[j] = x) \wedge (\forall i \in \mathbb{N}, i < \text{len}(A) \implies A[i] \leq x)$

You may find it helpful to formally define 'helper' functions or predicates, as is done in question 3.[3]

(b) The given Python code above has a bug. Demonstrate the bug by finding a value of A which meets the precondition, where umax misbehaves. For the value of A that you find, you should state the expected behaviour (according to your postcondition) and how it differs from the function's actual behaviour on that input.

(c) Consider our second draft of the function umax below:

```
def umax(A):
    if len(A) == 1:
        return A[0]
    head = A[0]
    tail = A[1:]
    tmax = umax(tail)
    if head == tmax:
        return -1 * head
    elif head > abs(tmax):
        return head
    else:
        return tmax
```

Prove that this function is correct with respect to the specifications you devised in part (a).

[3]To be posted soon!