ICS 311, Fall 2020, Problem Set 2, Topics 3 & 4 (Section 1)

Firstname Lastname <uhumail@hawaii.edu>

Due by midnight Tuesday September 15th. Please include your name in this document as well as in the file name.

#1. Proofs of Asymptotic Bounds

5 points

(a) Show that the function **$f(n) = 3n^2 - 2n$ is $\Theta(n^2)$**. Suggested steps:

1. Write the inequalities required by the definition of Θ, replacing f(n) and g(n) with the actual functions above.
2. Choose the needed constants, and rewrite the inequalities with these constants.
3. Prove that the inequalities hold for $\forall\ n \geq n_0$

(b) "little o": Consider the following proposed proof that $3n^2 + n = o(n^2)$

Let c=4, $n_0 = 2$.

Then $3n^2 + n < 4n^2 = 3n^2 + n^2$, for all $n >= n_0$, since $n < n^2$ for all $n > 1$

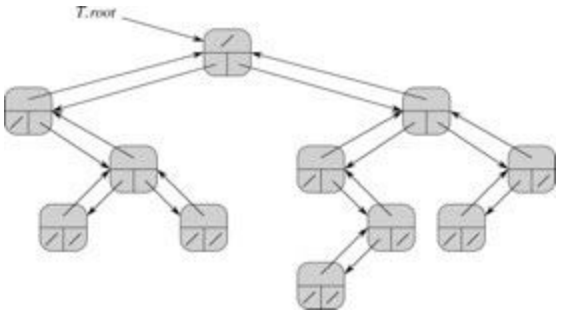
We showed strict inequality. Is this a correct little-o proof? Why or why not?

#2. Tree Traversals

5 points

In class you wrote a recursive procedure for traversal of a binary tree in O(n) time, printing out the keys of the nodes. Here you write two other tree traversal procedures. The first is a variation of what you wrote in class; the second is on a different kind of tree from CLRS pages 248-249 and in the lecture notes and screencast.

(a) Write an O(*n*)-time ***non-recursive*** procedure that, given an *n*-node binary tree, prints out the key of each node of the tree in ***preorder***. Assume that trees consist of vertices of class `TreeNode` with instance variables `parent`, `left`, `right`, and `key`. Your procedure takes a `TreeNode` as its argument (the root of the tree). ***Use a stack as an auxiliary data structure.***



printBinaryTreeNodes(TreeNode root) {

(b) Prove that your solution works and is O(n).

#3. Catenable Stack

10 points

In this problem you will design a data structure that implements Stack ADT using singly-linked list instead of an array. In addition your stack will have the following additional operation:

public **catenate**(Stack s); // appends the contents of Stack s to the current stack

The new operation will have the following properties:
Let n = s1.size(), m = s2.size(). Then executing s1.catenate(s2) results in the following:

1. The new size of s1 is the sum of the size of s2 and the original size of s1, i.e., the following evaluates to true: s1.size() == n+m
2. Top n elements of s1 after the call s1.catenate(s2) are the same as the elements of s1 before the call. The bottom m elements of s1 after the call s1.catenate(s2)are the same as the elements of s2 before the call.

Notice that s1 is modified (we don't make a new Stack object).

(a) The implementation described in the book, lecture notes and screencasts uses an array to implement Stack ADT. Can you implement catenate(Stack s) operation that runs in O(1) time for such implementation? If yes, write down the algorithm that achieves that and prove that it runs in O(1) time. If not, describe what goes wrong.

(b) Write down algorithms that implement the original Stack ADT using a singly-linked list instead of the array. Using class `ListNode` with instance variables `key` and `next`, write pseudocode for implementing each operation of Stack ADT: Stack(), push(Object o), pop(), size(), isEmpty(), top(). Be sure your code supports the catenate operation (next question).

(c) Design an algorithm that implements catenate(Stack s) operation in O(1) time. Write down the algorithm and prove that it runs in O(1) time.

#4. A Hybrid Merge/Insertion Sort Algorithm

14 points

Although MergeSort runs in $\Theta(n \lg n)$ worst-case time and InsertionSort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort (including that fact that it can sort in-place) can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the MergeSort recursion tree by using InsertionSort within MergeSort when subproblems become sufficiently small.

Consider a modification to MergeSort in which n/k sublists of length k are sorted using InsertionSort and are then merged using the standard merging mechanism, where k is a value to be determined in this problem. In the first two parts of the problem, we get expressions for the contributions of InsertionSort and MergeSort to the total runtime as a function of the input size n and the cutoff point between the algorithms k.

(a) Show that InsertionSort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time. To do this:

1. write the cost for sorting k items with InsertionSort,
2. multiply by how many times you have to do it, and
3. show that the expression you get simplifies to $\Theta(nk)$.

(b) Show that MergeSort can merge the n/k sublists of size k in $\Theta(n \lg (n/k))$ worst-case time. To do this:

1. draw the recursion tree for the merge (a modification of figure 2.5),
2. determine how many elements are merged at each level,
3. determine the height of the recursion tree from the n/k lists that InsertionSort had already taken care of up to the single list that results at the end, and
4. show how you get the final expression $\Theta(n \lg (n/k))$ from these two values.

Putting it together: The asymptotic runtime of the hybrid algorithm is the sum of the two expressions above: the cost to sort the n/k sublists of size k, and the cost to divide and merge them. You have just shown this to be:

$$\Theta(nk + n \lg (n/k))$$

In the second two parts of the question we explore what k can be.

(c) The bigger we make k the bigger lists InsertionSort has to sort. At some point, its $\Theta(n^2)$ growth will overcome the advantage it has over MergeSort in lower constant overhead. How big can k get before InsertionSort starts slowing things down? Derive a theoretical answer by proving the largest value of k for which the hybrid sort has the same Θ runtime as a standard $\Theta(n \lg n)$ MergeSort. This will be an upper bound on k. To do this:

1. Looking at the expression for the hybrid algorithm runtime $\Theta(nk + n \lg (n/k))$, identify the upper bound on k expressed as a function of n, above which $\Theta(nk + n \lg (n/k))$ would grow faster than $\Theta(n \lg n)$. Give the f for $k = \Theta(f(n))$ and argue for why it is correct.
2. Show that this value for k works by substituting it into $\Theta(nk + n \lg (n/k))$ and showing that the resulting expression simplifies to $\Theta(n \lg n)$.

(d) Now suppose we have two specific implementations of InsertionSort and MergeSort. How should we choose the optimal value of k to use for these given implementations in practice?