ICS 311, Fall 2020, Problem Set 02, Topics 3 & 4 Solutions

34 points

#1. Proofs of Asymptotic Bounds

5 points

(a, 4pts) 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 $\forall\ n \geq n_0$

Solution:

1. Show that $0 \leq c_1 n^2 \leq 3n^2 - 2n \leq c_2 n^2,\ \forall\ n \geq n_0$
2. Choose $c_1 = 1$, $c_2 = 4$, $n_0 = 2$. So, show that $0 \leq n^2 \leq 3n^2 - 2n \leq 4n^2,\ \forall\ n \geq 2$
3. $0 \leq n^2$ is always true for $n > 1$. For the rest: divide by n^2 and simplify:

   $1 \leq 3 - 2/n \leq 4$

   $1 \leq 3 - 2/n$ will be true for $\forall\ n \geq 2$ since at n=2, 3 - 1 = 2, and as n grows the quantity subtracted from 3 (that is, 2/n) gets smaller, so 3-2/n will always be larger than 1.

   $3 - 2/n \leq 4$ is always true as we can't get larger than 4 by subtracting a positive value from 3.

(b, 1pt) "little o": Consider the following proposed proof that $3n^2 + n = o(n^2)$

Let c=4, $n_0 = 2$.

Then $3n^2 + n \mathbf{<} 4n^2 = 3n^2 + n^2$, 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?

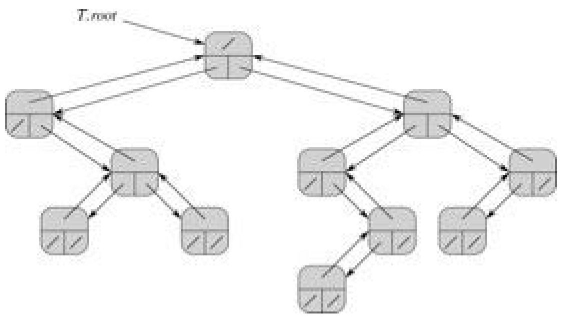
Solution: No, it is not. By the definition of little-o, one must prove the inequality for <u>all</u> constants c > 0, not just one. For example, it does not work for c = 1. *(Thus, the definitions of little-o and ω differ from their "big" counterparts not only in strict inequality but also the quantification of constants.)*

#2. Tree Traversals

5 points (3 for code, 2 for explanation and runtime)

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 another tree traversal procedure.

(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.***



Solution:

```
printBinaryTreeNodes(root)
    if (root != null)
        s = new Stack()                         // create new stack
        s.push(root)
        while (s is not empty)
            currentNode = s.pop()
            print(currentNode.key)
            if (currentNode.rightChild != null)
                s.push(currentNode.rightChild)
            if (currentNode.leftChild != null)
                s.push(currentNode.leftChild)
        }
     }
}
```

(b) Explain why your solution works and is O(n).

Every node is pushed on the stack exactly once because a tree is a connected graph (we reach all nodes via child pointers) with no cycles (we cannot repeat a node). Nodes are printed in preorder because the first thing we do to a node coming off the stack is to print its key; then we process the children. The items in the stack are first-in, last-out. By inserting the right child first, then the left child, we guarantee that the left child is processed first.

Every single line in the code above is O(1). Each node is pushed and popped only once, and there are n nodes, so the runtime 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, 1pt) 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.

Solution: The only way to implement catenation of two stacks implemented as arrays is to copy contents of the array implementing one stack into the array implementing the other stack, or to copy both into a larger array if there is not enough room in the first array. To copy the contents of the array s2, one must look at every element, thus taking $\Omega(n)$ time, where n is the number of elements in s2. Therefore, it's impossible to do it in O(1) time.

(b, 6 pts) 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).

Solution:
Push: Head points to new element; new element points to prior first element.
Pop: copy data out of first element; make head point to head.next.next.
Size: need to maintain a counter for this.
isEmpty: check if size is 0.
Commented out lines are for the next question.

```
Stack()
   ListNode top = null
   size = 0
   ListNode tail = null    // needed to make it catenable
```

```
push(Object o)
   ListNode newNode = new ListNode(o)
   newNode.next = top
   top = newNode
   size++
   if (size == 1) tail = top    // needed to make it catenable

Object pop()
   if isEmpty()
      error "Stack underflow"
   else
      if (size == 1) tail = null    // needed to make it catenable
      returnObj = top.key
      top = top.next
      size--
   return returnObj

Object top()
   if isEmpty()
      error "Stack underflow"
   else
     return top.key

int size()
   return size

boolean isEmpty()
   return (size == 0)
```

(c, 3pts) 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.

Solution:
To do this we need a tail pointer (implemented as above by the commented lines). Tail points to the last node of the list. Then:

```
catenate(Stack s)
   if isEmpty()
      head = s.head
   else
      tail.next = s.head
```

```
if !s.isEmpty()  tail = s.tail
size+=s.size
```

Each line of the above is a simple assignment statement or boolean test. There is no iteration or recursion so it is O(1).

#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, 3pts) Show that InsertionSort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time. To do this:

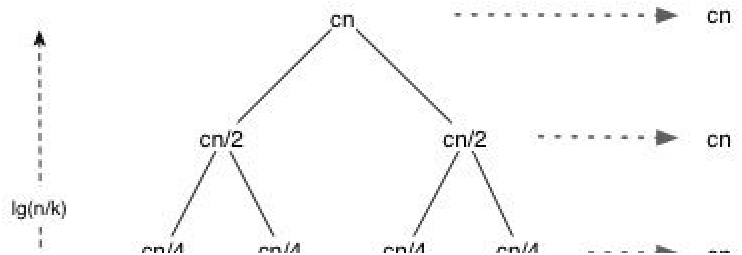
1. write the cost for sorting k items with InsertionSort,
2. multiply by how many times you have to do it, and
3. show that the expression you get simplifies to $\Theta(nk)$.

Solution:
$\Theta(k^2)$ is cost to sort k items with Insertion Sort
There are n/k sublists. Multiplying:
$\Theta(k^2*(n/k)) = \Theta(nk)$.

(b, 4pts) 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.

Solution:
1. The tree is shown to the right. The students' version of



the tree should have the correct branching, have correct costs at each node (n, $n/2$, ... etc.), and indicate that the recursion ends when lists are of size k. (The c's are not required as the question asks "how many elements are merged".) The other elements are answers to the next items:

2. As shown, n elements are merged at each level.

3. There are n/k lists (of length k) when the recursion ends, and we merge them pairwise working our way back up the tree until they become one list, so it will take $\Theta(\lg(n/k))$ merges to reduce the n/k lists to one list.

4. Multiply n ("width") times $\lg(n/k)$ ("height") to get the total work in the recursion tree $= \Theta(n\lg(n/k))$.

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, 4pts) 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 $\Theta(n \lg n)$.

Solution:

1. Referencing $\Theta(nk + n \lg (n/k))$:
If $k = f(n) > \lg(n)$ in terms of growth rate, then the overall expression will have a term $nk = \omega(n \lg n)$, so the hybrid algorithm would have a runtime growth rate greater than (run slower than) Merge Sort, which is $\Theta(n \lg \text{n})$. Thus an upper bound on k is $\lg(n)$.

2. First let's rewrite: $\Theta(nk + n\lg(n/k)) = \Theta(nk + n\lg(n) - n\lg(k))$.
Substituting $k = \lg(n)$ into the above, we get
$\Theta(n\lg(n) + n\lg(n) - n\lg(\lg(n))) = \Theta(2n\lg(n) - n\lg(\lg(n)))$
but we can ignore the constant 2 and the second term grows slower, so this is $\Theta(n\lg(n))$, showing that $k = \lg(n)$ is the point at which growth rate of the hybrid

algorithm equals that of unmodified MergeSort.

(d, 1pt) 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?

Solution: Use the largest value of *k* for which InsertionSort is faster than MergeSort.

(Since MergeSort is recursive, you are substituting a call to MergeSort on data of size k with a call to InsertionSort on that data. The actual value of k depends on the implementation, i.e., the constants that are ignored in the analysis. It would make sense to start your empirical tests of small list sort time at lg n for expected lists sizes n and work down from there, since the theoretical analysis says we break even at lg n.