1. Proof 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.

The inequality can be defined as followed:
$c_1n^2 \leqslant 3n^2 - 2n \leqslant c_2n^2$

2. Choose the needed constants, and write the inequalities with these constants.

Let $c_1 = 3$ and $c_2 = 4$
Thus this can be rewritten with the constants:
$3n^2 \leqslant 3n^2 - 2n \leqslant 4n^2$

3. Prove that the inequalities hold for $\forall n \geqslant n_0$

By dividing both sides of the inequalities by n^2
$3 \leqslant 3 - 2n \leqslant 4$ This inequality will hold true given that $n > n_0 = 1$
$\therefore$ It is true that $f(n) = 3n^2 - 2n$ holds to be $\theta(n^2)$

(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 $n \geqslant n_0$, since $n < n^2$ for all $n > 1$. We showed strict inequality. Is this a correct little-o proof? Why or why not?

The definition of little-o is as follows:
$g(g(n)) = (0 \leqslant f(n) < c(g(n)))$
Let $f(n) = 3n^2 + n$ and let $g(n) = n^2$.
Therefore, it can be proven that $f(n) = 3n^2 + n \leqslant 4n^2$ such that $n \geqslant 0$.
Thus, $f(n) \leqslant c * g(n) \forall n \geqslant 0$ where $c = 4$
$\therefore$ The definition of little-o proves that $4n^2 + n = o(n^2)$

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. Hee 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** procedre 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 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 auxillary data structure**.

Algorithm 1 printBinaryTreeNodes(TreeNode root)

```
if (node == NULL)
    return ROOT NULL
s = new()
s.push(root)
while (s.isEmpty() == FALSE)
    TreeNode baseNode = s.top()
    Print out the base node key
    s.pop()
    if (baseNode.right != NULL)
        s.push(baseNode.right)
    if (baseNode.left != NULL)
        s.push(baseNode.left)
```

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

Note: Each line of the algorithm corresponds to a single line of code in pseudo code.
Given the first lines corresponding 1-4 will run in $\theta(1)$ and that lines 5-12 will take $\theta(n)$, this can be mathematically represented as:

$$\theta(1) * theta(n) = theta(n) \text{ by domination of leading terms.}$$

The algorithm will therefore terminate until the stack is empty; this will not occur until every node that is within the tree is pushed onto the stack, and popped off after the key in it is printed.

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); // apends 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 afte 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.

It is not possible for there for be an algorithm for Stack ADT to run in O(1) time. The reason why this is not possible, is because copying contents of an array from one to the other requires the algorithm to search through contents of each element, which takes O(n) time.

(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).

From a review of ICS211, it is given that the operations of Stack work on the first element of a given list of k elements as follows:

Algorithm 2 Stack( )

```
    Node = null
    size = 0
```

The push method will point the head to a new element, and its new element will point to the element before the first.

Algorithm 3 push(Object o)

```
    Node newNode = new Node(o)
    newNode.next = top
    top = newNode
    ++size
```

The pop method will check to see the data that is in the first element, and makes the head of the stack point to the head.next.next.

Algorithm 4 Object pop()

```
if isEmpty()
    print the error of nothing in Stack
else
    newObj = top.key
    top = top.next
    –size
return newObj
```

The size will return the size of the stack, which maintains a counter for how many items are within the stack that changes.

Algorithm 5 int size()

```
return size of the stack
```

This method will return 0, if the size of the stack is 0.

Algorithm 6 isEmpty()

```
return size == 0
```

(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.

Algorithm 7 catenate(Stack s)

```
if stack = empty
    point the head to front of the stack and sets to s.head
else
    point the next tail to the head of the list
if stack != EMPTY
        set the tail equal to stack s.tail
size += stack size
```

Given that each line respective will run O(1) times, an equation to represent it would be as follows:

$$0(c1 + c2 + c3 + c4 + c5 + c6) = O(1)$$

4. A Hybrid Merge/Insertion Sort Algorithm

14 points

Although MergeSort runs $\theta(nlgn)$ worst-case time and InsertionSort runs in $\theta(n^2)$ worst-case time, the constant factors in insertion sort (including the 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 $\frac{n}{k}$ 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.

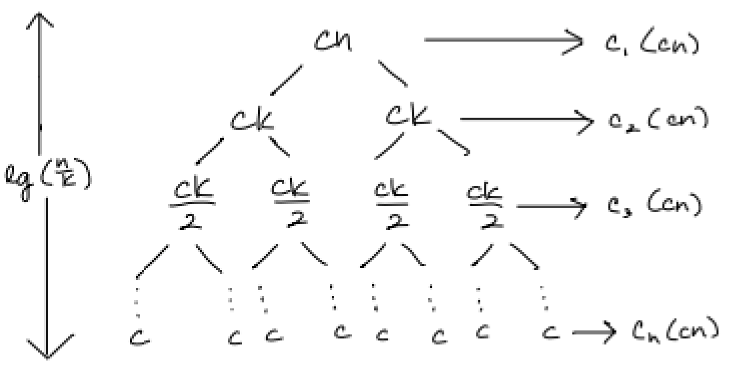
(a) Show that InsertionSrt can sort the $\frac{n}{k}$ sublists, each of length k, in $\theta(nk)$ worst-case time. To do this:

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 $\theta(nk0)$.

The cost for sortking k items with InserstionSort is $O(k^2)$ time, depending on the amount of k-elements within the given list. By multiplying this by the sublist, given as $\frac{n}{k}$, you now have $O(k^2 * \frac{n}{k})$, which simplifies to $O(nk)$. This is given to be the worst case time scenario.

(b) Show that MergeSort can merge the $\frac{n}{k}$ sublists of size k in $\theta(nlg(\frac{n}{k}))$ 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 $\frac{n}{k}$ 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 $\theta(nlg(\frac{n}{k}))$ from these two values.



By multiplying n elements by the height of the tree given as $lg(\frac{n}{k})$, you have the following equation

$$n * lg(\tfrac{n}{k}) \Leftrightarrow n\,lg(\tfrac{n}{k})$$

$\therefore\ \Theta(n\,lg(\frac{n}{k}))$ is given 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 $\frac{n}{k}$ sublists of size k, and the cost to divide and merge them. You have just shown this to be:

$$\theta(nk + nlg(\tfrac{n}{k}))$$

In the second 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 for k for which the hybird sort has the same θ runtime as a standard $\theta(nlgn)$ MergeSort. This will be an upper bound on k. To do this:

1. Looking at the expression for the hybrid algorithm runtime $\theta(nk + nlg(\frac{n}{k}))$, identify the upper bound on k expressed as a function of n, above which $\theta(nk + nlg(\frac{n}{k}))$ would grow faster than $\theta(nlgn)$. 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 $\theta(nk + nlg(\frac{n}{k}))$ and showing that the resulting expression simplifies to $\theta(nlgn)/$

The upper bound on k, expressed as a function of n above can be denoted to not grow faster than *logn*. The modified algorithm therefore, must have the same running time as the standard running time of merge sort. To understand this, the resulting expression can be substituted as follows:

$$\theta(nk + nlog(\tfrac{n}{k})) = \theta(nk + nlogn - nlogk)$$

This equation therefore, must be the same as $\theta(nlogn)$.

(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?

As stated above in part (c), the upper bound of k should be a value that will minimize the average or worst running time of the combined InsertionSort and MergeSort algorithm. Therefore, k should be the list in which InsertionSort would run faster than MergeSort.