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 $n > 1$

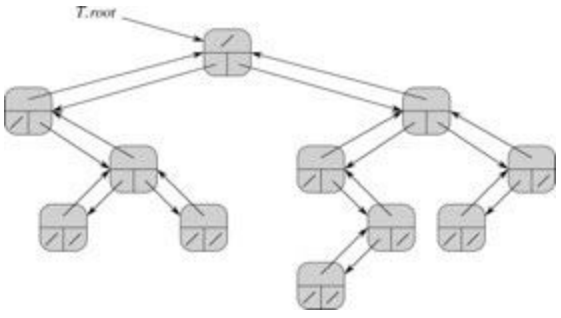
We showed strict inequality. Is this a correct little-o proof? Why or why not?

#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?

ICS 311 Fall 2020, Problem Set 04, Topics 7 & 8

Firstname Lastname <uhumail@hawaii.edu> , Section 1
Due by midnight Tuesday Sept. 29nd. 40 points total.

#1. Master Method Practice (6 pts)

Use the Master Method to give tight Θ bounds for the following recurrence relations. Show *a*, *b*, and *f*(*n*). Then explain why it fits one of the cases, choosing ε where applicable. Write and *simplify* the final Θ result

(a) $T(n) = 3T(n/9) + n$

(b) $T(n) = 7T(n/3) + n$

(c) $T(n) = 2T(n/4) + \sqrt{n}$

#2. Solving a Recurrence (10 pts)

Solve the following recurrence by analyzing the structure of the recursion tree to arrive at a "guess" and using substitution to prove it. Justify all steps to show that you understand what you are doing.

$T(n) = T(2\sqrt{n}) + c$ if $n > 8$
$T(n) = c$ if $n \leq 8$

#3. Binary Search Tree Proof (11 pts)

The procedure for deleting a node in a binary search tree relies on a fact that was given without proof in the notes:

> **Lemma:** If a node X in a binary search tree has two children, then its successor S has no left child and its predecessor P has no right child.

In this exercise you will prove this lemma. Although the proof can be generalized to

duplicate keys, for simplicity assume no duplicate keys. The proofs are symmetric, so we start by proving for the successor. We rule out where the successor cannot be to narrow down to where it must be. Drawing pictures may help.

(a) Prove by contradiction that the successor S cannot be an ancestor or cousin of X, so S must be in a subtree rooted at X.

(b) Identify and prove the subtree of X that successor S must be in.

(c) Show by contradiction that successor S cannot have a left child.

(d) Indicate how this proof would be changed for the predecessor.

#4. Deletion in Binary Search Trees (5 pts)

Consider Tree-Delete (CLRS page 298), where `x.left`, `x.right` and `x.p` access the left and right children and the parent of a node x, respectively.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4     TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)  // successor
 6      if y.p != z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
 10       TRANSPLANT(T, z, y)
 11       y.left = z.left
 12       y.left.p = y
```

(a) How does this code rely on the lemma you just proved in problem 3? Be specific, referring to line numbers. Be careful that you are using the actual lemma above, and not a similar fact proven elsewhere.

(b) When node *z* has two children, we arbitrarily decide to replace it with its successor. We could just as well replace it with its predecessor. (Some have argued that if we choose randomly between the two options we will get more balanced trees.) Rewrite Tree-Delete to use the predecessor rather than the successor. Modify this code just as you need to and underline or boldface the changed portions.

#5. Constructing Balanced Binary Search Trees (8 pts)

Suppose you have some data keys sorted in an array `A` and you want to construct a *balanced binary search tree* from them. Assume a tree node representation `TreeNode` that includes instance variables `key`, `left`, and `right`. (No `p` needed in this problem.)

(a) Write pseudocode (or Java if you wish) for an algorithm that constructs the tree and returns the root node. (We won't worry about making the enclosing BinaryTree class instance.) You will need to use methods for making a new TreeNode, and for setting its left and right children.

Hints: Think about how BinarySearch works on the array. Which item does it access first in any given subarray it is called with? How can we set up the tree so that a search sequence for a given key examines the same keys as it does in the array?

(b) What is the Θ time cost to construct the tree, assuming that the array has already been sorted? Justify your answer.

(c) Compare the expected runtime of BinarySearch on the array to the expected runtime of BST TreeSearch in the tree you just constructed. Have we saved time?

ICS 311, Fall 2020, Problem Set 03, Topics 5 & 6

Firstname Lastname <uhumail@hawaii.edu> (section 1)
Due by midnight Tuesday Sept. 22nd. 30 points total.

#1. Expected Length of Coding Scheme

6 points

Suppose that several information sources generate symbols at random from a five-letter alphabet {A, B, C, D, E} with different probabilities. We will try different encoding schemes for encoding these symbols in binary. Given the probabilities and an encoding scheme, you will compute the expected length of the encoding of n letters generated by the information source. You may use any rigorous method of analysis you like, but must show your work and justify your answer.

(a) Suppose that the symbols occur with probabilities Pr[A] =0.4 , Pr[B] = 0.2, Pr[C] = 0.2, Pr[D] = 0.1, and Pr[E] = 0.1, and the coding scheme encodes these symbols into binary codes as follows:

A	001
B	010
C	011
D	100
E	101

What is the expected number of bits required to encode a message of n symbols?

(b) Now suppose that the symbols occur with the same probabilities Pr[A] =0.4 , Pr[B] = 0.2, Pr[C] = 0.2, Pr[D] = 0.1, and Pr[E] = 0.1, but we have a different encoding scheme:

A	0
B	10
C	110
D	1110
E	1111

What is the expected number of bits required to encode a message of n symbols?

(c) Now consider a different information system that generates symbols with probabilities Pr[A] =0.5 , Pr[B] = 0.3, Pr[C] = 0.1, Pr[D] = 0.05, and Pr[E] = 0.05. We will use the same encoding scheme:

A	0
B	10
C	110
D	1110
E	1111

What is the expected number of bits required to encode a message of n symbols?

#2. Random Gene Sequences

6 points

Suppose a strand of DNA is generated by appending random nucleotides with equal probability from the set {A,T,G,C} to the end of the sequence. What is the expected length of the sequence needed to get two As in a row? You may use any rigorous method of analysis you like, but must show your work and justify your answer.

#3. Hashing with Chaining

6 points

(a) (2 pts) Consider a hash table with m slots that uses chaining for collision resolution. The table is initially empty. What is the probability that, after k keys are inserted, there is a chain of size k? Include an argument for or proof of your solution.

(b) (4 pts) Show the table that results when 20, 51, 10, 19, 32, 1, 66, 40 are cumulatively inserted in that order into an initially empty hash table of size 11 with chaining and $h(k) = k \bmod 11$. Use the Google Doc table below, with positions indexed 0 to 10, and linked lists going off to the right. Do not enter anything in cells that are not used.

h(k)	Linked list cells			
0				
1				
2				
3				
4				
5				
6				

7				
8				
9				
10				

#4. Open Addressing Strategies

12 points

(a) (4 pts): Show the table that results when 20, 51, 10, 19, 32, 1, 66, 40 are cumulatively inserted in that order into an initially empty hash table of size 11 with linear probing and

h'(k) = k mod 11

h(k,i) = (h'(k) + i) mod 11, *where the first probe is probe i=0.*

Draw this and the next result as horizontal arrays indexed from 0 to 10 as shown below. (You can fill in the Google Doc table.) Show your work in part (b) to justify your answer!

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

(b) (1 pt): How many re-hashes after collision are required for this set of keys? *Show your work here so we can give partial credit or feedback if warranted.*

(c) (4 pts): Show the table that results when 20, 51, 10, 19, 32, 1, 66, 40 are cumulatively inserted in that order into an initially empty hash table of size m = 11 with double hashing and

$h(k,i) = (h_1(k) + ih_2(k)) \bmod 11$

$h_1(k) = k \bmod 11$

$h_2(k) = 1 + (k \bmod 7)$

Refer to the code in the book for how i is incremented. Show your work in part (d) to justify your answer!

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

(d) (1 pt): How many re-hashes after collision are required for this set of keys? *Show your work here so we can give partial credit or feedback if warranted.*

(e) (2 pts): Open addressing insertion is like an unsuccessful search, as you need to find an empty cell, i.e., to *not* find the key you are looking for! If the open addressing hash functions above were uniform hashing, what is the expected number of probes at the time that the last key (40) was inserted? *Use the theorem for unsuccessful search in open addressing and show your work.* Answer with a specific number, not O or Theta.

Pau

FYI, we removed the skip list questions, as they were merely tracing algorithms without analysis. When you study for exams, see the class problem solutions.

ICS 311, Fall 2020, Problem Set 01, Topic 2 (Section 1)

Firstname Lastname <uhumail@hawaii.edu>

Due by midnight Tuesday September 8. Please include your name in this document as well as in the file name.

#1. Correctness of Linear Search

7 points

(a) Show the pseudocode for Linear Search that you will be analyzing. (It should be code that you understand and believe is correct. You may revise your solution from class, or use the instructor's solution.) Give each line a number for reference in your analysis.

(b) Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties (page 19 CLRS).

(Hint: The loop can exit for two reasons. Rather than trying to write and prove an invariant that covers the two cases, use a simpler invariant that deals with only the correctness of the loop completion returned value, as it would be too complex to cover both, and the within-loop returned value is easy to show correct. This is similar to what we did in class for binary search.)

#2. Runtime of BinarySearch

8 points

This problem steps you through a recursion tree analysis of BinarySearch to show that it is Θ(lg *n*) in the worst case. Here is a recursive version of BinarySearch:

```
BinarySearch(x, A, low, high)
1     if (low > high)
2           return "NOT FOUND" // or a sentinel such as -1
3     else
4           mid =⌊(low + high) / 2⌋
5           if x < A[mid]
6                 return BinarySearch(x, A, low, mid-1)
7           else if x > A[mid]
8                 return BinarySearch(x, A, mid+1, high)
9           else
10                return mid
```

(Comments: The problem involves mathematical equations and drawing diagrams. It is your choice whether to figure out how to do this in Google Docs & Drawing, to do it in your favorite program, or to do your work on paper and scan or photograph it. We prefer that you compile the results in one PDF, but if that is difficult compile them into one zip file.)

(a) Write the recurrence relation for BinarySearch, using the formula T(n) = aT(n/b) + D(n) + C(n). (We'll assume T(1) = some constant c, and you can use c to represent other constants as well, since we can choose c to be large enough to work as an upper bound everywhere it is used.)

(b) Draw the recursion tree for BinarySearch, in the style shown in podcast 2E and in Figure 2.5 of CLRS. Don't just copy the example for MergeSort: it will be incorrect. Make use of the recurrence relation you just wrote!

(c) Using a format similar to the counting argument in Figure 2.5 of the text or of podcast 2E, **use the tree to show that BinarySearch is Θ(lg n) in the worst case**. Specifically,

1. show what the row totals are,
2. write an expression for the tree height (justifying it), and
3. use this information to determine the total computation represented by the tree.

#3. Correctness of BubbleSort

15 points

BubbleSort is a well known but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order, and doing so enough times that there can be no more elements out of order. From CLRS:

```
BUBBLESORT(A)
1  for i = 1 to A.length - 1
2     for j = A.length downto i + 1
3           if A[j] < A[j - 1]
4                 exchange A[j] with A[j - 1]
```

Let A' (an array) denote the output of BubbleSort. In order to prove that BubbleSort is correct, we need to show that

1. A' is a permutation of A; that is, A' has the same elements as A.
2. A'[1] ≤ A'[2] ≤ A'[3] ... ≤ A'[n-1] ≤ A'[n]; that is, A' is sorted

The first part can be proven by showing that BubbleSort never removes or adds items to the array: it merely swaps them in line 4. In this homework, you will use loop invariants to prove the

second part, that the resulting array is in order. Since there are nested loops this requires two loop invariants, handled in (a) and (b).

(a) State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

(b) Using the termination condition of the loop invariant proved in part (a), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove the inequality $A'[1] \leq A'[2] \leq A'[3] \ldots \leq A'[n-1] \leq A'[n]$. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

(c) Three parts:

- Give the worst-case running time of BubbleSort, in big-O notation, with justification.
- Give the best-case running time of BubbleSort, in big-O notation, with justification.
- How do these compare to the running time of InsertionSort?