**CSC 3110**

**Algorithm Design and Analysis**

(30 points)

Due: 02/28/2024 11:59 PM﻿﻿﻿

**Note:** **Submit answers in PDF document format. Please read the submission format for appropriate file naming conventions.**

1. Exercises 5.1:
   1. Problem 1 (parts a - d) (2 points)   
      A text on a white background

      Description automatically generated
2. Too bad we can’t assume n is sorted. Let’s use merge sort  
   Pseudocode:  
   Recursively divide array into two halves  
   Find largest element in each half  
   Merge two halves together starting from bottom of recursion   
     
   b.) It will still work, the one that will be returned will be the one from the leftmost index.

c.)Starting to realize my pseudo code sucked. T(n) = 2T(n/2) +n seems to be the standard number of key comparisons for merge sort. Merge sort has an efficiency of nlog(n)  
  
d.) This algorithm is less efficient than the brute force mechanism, brute force only requires that O(n) take place where every element is compared, this requires dividing everything which takes MORE time because inherently for each division an additional basic operation is required. Merge sort’s efficiency is O(n)=nlog(n)

* 1. Problem 5 (parts a - c) (2 points)  
     A math equation with black text

     Description automatically generated  
     Oh hey look it’s master theorem in action.  
     A white background with black text

     Description automatically generated  
     Here is what master theorem looks (though this one I googled uses k instead of d)like so I can stop looking it up, now we just need to identify a,d,and b  
     a.)   
     A math equations and arrows

     Description automatically generated with medium confidence

4 > 2^1 ------> T(n) = N^(log2(4))-> O(n)=n^2

b.) A red arrows and numbers

Description automatically generated with medium confidence  
2^2 = 4 ------> T(n) = n^2log(n)

c.)

A math equations and formulas

Description automatically generated with medium confidence  
4 < 2^3 -----> T(n) = n^3

1. Exercises 5.2:
   1. Problem 8 (2 points)   
      A black text on a white background

      Description automatically generated  
      Too bad we haven’t learned bucket sort yet, it doesn’t say that the array has to be sorted, just that negative has to be on one side, and positive on the other.  
      1.) Define a variable for the left index, L with the initial value equaling to zero, define a variable for the right index, R, the with the initial value equally to one.  
      2.) Check if the value at L is negative, if it is, leave it, increase L by one.  
      3.) Check if the value at L is positive, if it is, begin increasing the value of R by one until a negative number is found. When a negative number is found, swap the positions of the positive number at L and the negative number at R. Increase L by one.

4.)Repeat steps 2-3 until R = (n-1)

This algorithm is pretty simple, the basic idea is put all of the negative numbers on the left side, so we simply start at the left side and reach for the first negative number we find and put it there. Theoretically this algorithm has an efficiency of O(n) because the right index never scans more than the entirety of the array, and the number of swaps never exceeds the number of negative numbers in the array.

* 1. Problem 10 (2 points)  
     A black text on a white background

     Description automatically generated  
     Python has quicksort built in as its default sorting algorithm. I will thereby demonstrate how to use the python implementation of quicksort in python:

List.sort()  
  
Wow. What a sort.

1. Exercises 5.3:
   1. Problem 2 (2 points)   
      A screenshot of a computer code

      Description automatically generated  
      This is a recursive algorithm that starts at the root of a binary tree, and call itself for the left and right branches of that binary tree. However, when the recursion reaches an empty leaf, it returns 0, which makes sense, as empty leaves are not leaves and are the end of the line, however, there is no recursive call for if the leaf is the last leaf and has a value, so as a result the recursion returns 0 for everything.

Here is the fix:

If T=NULL return 0 //not a leaf, not anything  
elseif TL = NULL and TR = NULL return 1 //case where the leaf has a value, but

//its children don’t  
else return LeafCounter(TL)+(LeafCounter(TR) //recursion call

* 1. Problem 5 (parts a - c) (2 points)

A screenshot of a computer

Description automatically generated  
PreOrder: Root, Left, Right  
A diagram of a connected line

Description automatically generated

a,b,d,e,c,f

InOrder: Left,Root,Right

A diagram of a diagram

Description automatically generated  
d,b,e,a,c,f

PostOrder:Left,Right,Root  
  
A diagram of a network

Description automatically generated

D,e,b,f,c,a

1. Exercises 5.4:
   1. Problem 3 (part a & b) (2 points)   
      A math equations and symbols

      Description automatically generated with medium confidence
   2. Problem 5 (2 points)
2. Exercises 6.1
   1. Problem 1 (part a & b) (2 points)
   2. Problem 2 (part a & b) (2 points)  
      A close up of a text

      Description automatically generated
3. Exercises 6.3
   1. Problem 4 (parts a- c) (2 points)  
      A white background with black text

      Description automatically generated  
      A white background with red writing

      Description automatically generated  
      A white paper with red writing on it

      Description automatically generated  
      A white background with red numbers

      Description automatically generated
   2. Problem 7 (part a & b) (2 points)

A white background with black text

Description automatically generated

A screenshot of a test

Description automatically generatedC,O,M,P,U,T,I,N,G = 3,15,13,16,21,20,9,14,7

1. The largest number of comparison will require 4 comparison, this is through the path MPNO or MPTU. The average case efficiency of a 2-3 tree is log(n) for a search, so it is likely that the case efficiency would be log3(9) or around ~2. More specifically it would be the average of (1,2,2,3,3,3,3,4,4) ~2.77778
2. Exercises 6.4
   1. Problem 1 (parts a- c) (2 points)  
      A white background with black text

      Description automatically generated  
        
      a.)  
      A screenshot of a test

      Description automatically generated  
      b.)  
      A screenshot of a math test

      Description automatically generated  
      c.) No, bottom up and top down do not yield the same results always. There are multiple valid ways to build a heap, so the insertion order matters in how the heap is constructed.
   2. Problem 5 (part a & b) (2 points)  
      Black text on a white background

      Description automatically generated  
      a.) The smallest elements of a heap are at its leaves. All leaves in a heap are found in the second half of the heap (n/2). Therefore we just have to iterate through the second half of the heap until the end of the heap and take the lowest value. This value could then be deleted. The efficiency would be O(n/2), as deleting an element at the end of a heap does not require adjusting the elements above it.   
      b.) Unfortunately, an element of a given value can be anywhere in the heap because the element could be the largest or the smallest. The only thing that is certain is that the parent is smaller than its children. So one can simply scan from left to right from 0 to n-1 to find the element that one is searching for. However, once the element is found, and is deleted, the heap must be rebuilt, this takes log(n) for binary heaps. Therefore the time taken would be o(n)+log(n).
   3. Problem 7 (parts a- c) (2 points)  
      A close up of text

      Description automatically generated

S,O,R,T,I,N,G = 19,15,18,20,9,14,7  
A close up of numbers

Description automatically generated  
By increasing order you mean you want the largest number on top right? To say that the heap increases from bottom to top?  
A white paper with red writing

Description automatically generated

A white board with red writing

Description automatically generated