CS 325 -002 Quiz1

Lyell Read

TOTAL POINTS

51 / 50

QUESTION 1

Q1 10 pts

1.1 Q1a 5 / 5

✓ - 0 pts Correct

1.2 Q1b 5 / 5

✓ - 0 pts Correct

QUESTION 2

Q2 10 pts

2.1 Q2a 4 / 4

✓ - 0 pts Correct

2.2 Q2b 6 / 6

✓ - 0 pts Correct

QUESTION 3

Q3 7 pts

3.1 a 1 / 1

✓ - 0 pts Correct

3.2 b 1 / 1

✓ - 0 pts Correct

3.3 c 0 / 1

✓ - 1 pts incorrect

(high + low)/2

3.4 d 2 / 2

✓ - 0 pts Correct

3.5 e 2 / 2

✓ - 0 pts Correct

QUESTION 4

Q4 8 pts

4.1 4a 2 / 2

✓ - 0 pts Correct

4.2 4b 5 / 6

✓ - 0.5 pts incorrect c in conclusion

✓ - 0.5 pts incorrect n in conclusion

QUESTION 5

Q5 5 pts

5.1 a 1 / 1

✓ - 0 pts Correct

5.2 b 1 / 1

✓ - 0 pts Correct

5.3 c 1 / 1

✓ - 0 pts Correct

5.4 d 1 / 1

✓ - 0 pts Correct

5.5 e 1 / 1

✓ - 0 pts Correct

QUESTION 6

Q6 10 pts

6.1 a 4 / 4

✓ - 0 pts Correct

6.2 b 6 / 6

✓ - 0 pts Correct

QUESTION 7

7 EXTRA CREDIT 3 / 0

+ **5 pts** Correct

✓ + **3 pts minor errors**

+ **0 pts** blank or incomplete

+ **1 pts** effort

+ **2 pts** major error

+ **0 pts** not Theta(nlogn)

+ **4 pts** incomplete explanation

+ **4 pts** minor error in running time

CS 325 – 002 Winter 2020

Quiz 1

You may use a 3"x5" notecard and calculator. You have 50 minutes to complete the exam. Do not leave early, remain seated. When time is called pass your exam to the aisle. Failure to submit your exam in a timely manner will result in a zero.

Print your name: Lyell Read

Student ID: 933-609-535

1. (*10 pts*) Determine the theoretical running time of the following iterative algorithms. Use theta notation and give a brief explanation.

a)

```
int AlgoA(int n)
{
    int sum = 0;                                   c1
    for (int i = 0; i < n; i++) {                  n
        for (int j = i; j < n; j++) {              n/2
            for (int k = 1; k <= 3; k++) {         3
                sum = sum + 1;                     c2
                cout << " sum " << sum << endl;    c3
            }
        }
    }
}
```

i..n $\quad \frac{1}{2}n$

$n \cdot \frac{n}{2} \cdot 3 = \frac{3}{2} \cdot n^2 = \boxed{\Theta(n^2) \text{ running Time}}$

n^2 because two loops runtime depends on n, one of them runs n times, one runs $n/2$ times — this makes for total running time of $\frac{n^2}{2} \rightsquigarrow \Theta(n^2)$.

b)

```
int AlgoB(int n)
{
    int count = 1;                                 c1
    if ( n < 2 ) {
        cout << " One";                            c2
        return 1;
    } else {
        for (int i = 2; i*i <= n; i++) {
            count = count + 1;                     c1
            cout << " count " << count << endl;    c2
        }
        return count;
    }
}
```

$i=2 \quad i^2 <= n$

$\boxed{\Theta(\sqrt{n}) \text{ Running Time}}$

Because the only loop in the program runs about $\sqrt{n}$ times, the overall running time is $\Theta(\sqrt{n})$. It runs $\sqrt{n}$ times because it squares i when comparing it to n, so the largest ~~i~~ i will be $\sqrt{n}$, thus the running time.

Name: Lyell Read

OSU ID: 933-609-535

2. (*10 pts*) Consider the following recursive algorithm:

```
Algo1(n) {                                   c1
    total = 0                                c2
    if n <= 1 return 2                       c2
    else {                                   A(n/4)
        total = Algo1(n/4) + Algo1(n/4)      A(n/4)
        for i = 1 to n do                    n
            for j = 1 to n do                n
                total = i + j                c3
        return total                         c4
    }
}
```

(a) Write a recurrence for the running time T(n) of Algo1(n).

$$\boxed{T(n) = 2T\left(\frac{n}{4}\right) + n^2}$$

(b) Solve the recurrence for the asymptotic running time. Assume that addition can be done in constant time. Give the tightest bound possible.

$2T\left(\frac{n}{4}\right) + n^2$ $\quad a = 2$ $\quad b = 4$ $\quad c = 2$ $\quad f(n) = n^2$ ▶ $\log_4 2 = \frac{1}{2} < 2$

$$\boxed{\therefore \text{Running time} = f(n) = n^2 \sim \Theta(n^2)}$$

By Case 3 of Master Theorem

3. (*7 pts*) Complete the code below for a recursive divide and conquer algorithm that determines the maximum value of an unsorted array of integers. The function maxOfA divides the array A in half and recursively determines the maximum value of each half and then returns the larger value. For example if A[5] = {1, 12, 6, 4, 7} then maxOfA(A, 0, 4) would return 12.

```
int maxOfA( int A[], int low, int high )
{    // Returns the maximum value in array A[low....high]
     int maxLeft, maxRight, mid;       // max of the left half, max of the right half, midpoint
     // check if there is more than one element in A
     if( low ==high ) {
          // return current value if there is only one element
          return A[low] ;
     } else {
          // find the midpoint between low and high

          int mid = (int)((high-low)/2) ; //

          // find the max of the left and right halves
          maxLeft = maxOfA(A, low, mid) ;
          maxRight = maxOfA(A, mid+1, high) ;
          // return the max
          if (maxLeft >= maxRight)
               return maxLeft ;
          else
               return maxRight ;
     }
}
```

Name: Lyell Read

OSU ID: 933-609-535

4. (*8 pts*) Consider the following statement:

h

g

f

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$, for all non-decreasing positive functions $f(n)$, $g(n)$ and $h(n)$.

a) This statement is: (*circle one*)

TRUE | FALSE

b) If true prove using the formal definition of asymptotic notation. If false give a counterexample.

given that $f(n) = O(g(n))$, there must exist n_{0fg} C_{fg} s.t.

after all n_{0fg}, $f(n) < C_{fg}\, g(n)$

$\rightarrow$ the same is true for ordered sets of functions $\{g(n), h(n)\}$

and f

by the theorem proved in class slides that states

that

"$f(n) = O(g(n))$ && $g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$"

Therefore, given that $f(n) = O(g(n))$, $g(n) = O(h(n))$,

we can conclude that $f(n) = O(h(n))$ ∎

$0: O \quad 1: \Theta \quad \infty\ \Omega$

5. (*5 pts*) For each pair of functions, select **the one best** answer. If the answer is Θ select only Θ

a. f(n) = lg n; g(n) = log(n) + 5

- ○ f(n) is O(g(n))
- ○ f(n) is Ω(g(n))
- ● f(n) is Θ(g(n))

$\lim_{n \to \infty} \frac{\log}{\log} = 1$

b. f(n) = log(log n); g(n) = log n

- ● f(n) is O(g(n))
- ○ f(n) is Ω(g(n))
- ○ f(n) is Θ(g(n))

$\lim_{n \to \infty} \frac{\log(\log(n))}{\log(n)} \to 0$

c. $f(n) = \sqrt{n}$; g(n) = log n

- ○ f(n) is O(g(n))
- ● f(n) is Ω(g(n))
- ○ f(n) is Θ(g(n))

$\lim_{n \to \infty} \frac{\sqrt{n}}{\log n} = \infty$

d. $f(n) = e^n$; $g(n) = 2^n$

- ○ f(n) is O(g(n))
- ● f(n) is Ω(g(n))
- ~~●~~ f(n) is Θ(g(n))

$\lim_{n \to \infty} \frac{e^n}{2^n} \to \infty$

e. $f(n) = 2^n$; g(n) = n!

- ● f(n) is O(g(n))
- ○ f(n) is Ω(g(n))
- ○ f(n) is Θ(g(n))

$\lim_{n \to \infty} \frac{2^n}{n!} = 0$

Name: Wyell Read

OSU ID: 933-609-535

6. *(10 pts)* The code below recursively finds the minimum value of an array of integers.

```
int Find_Array_Min(int A[], int n)
{
    if (n== 0)                              c1
        return( A[0] );                     c2

    else {
        int minL = Find_Array_Min(A, n-1);  REC
        if ( A[n] < minL )
            return A[n];
        else                                c3
            return minL;
    }

}
```

a) Write a recurrence for T(n) the running time of the algorithm in terms of the input array size n.

$T(n) = T(n-1) + 1$

b) Solve the recurrence to determine that asymptotic running time of the algorithm. Use theta notation.

Pseudo Tree Method

$T(n)$

$T(n-1)$

$T(n-2)$

$T(n-3)$

$\vdots$

$\therefore$ Running Time $= n * c_{1-3}$ – c's are constant time so are ignored. Therefore, this algorithm is $\boxed{\Theta(n)}$

EXTRA CREDIT: *Attempt this problem only after completing all other problems.*

Describe (with words) a $\Theta(n \lg n)$ time algorithm that, given a **set S** of *n* integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Explain why the algorithm is $\Theta(n \lg n)$.

length n

$Alg\,01(\{S\}, x)$

To get $n\log n$, then you need recurrence s.t.
$\log_b a = 1 = c$

$T(n) = 2T(n/2) + n$

This algorithm will linearly travel the list of length n, noting the index of the first (if exist) and second (if exist) instance of x. Each time it stops, it will aditionally perform log(n) constant time operations in order to slow the running time to nlogn (n elements with log(n) busywork at each).

The other algorithm would be one with recurrence relation $T(n) = 2t(n/2) + n$ $\left(\begin{smallmatrix} a=2 \\ b=2 \end{smallmatrix}\right.$ $c=1$ $\log_2 2 = c = 1$ $\therefore \Theta(n\lg n)$

This algorithm runs recursively ~~over~~ over each half of the given array but is also passed the main array — it checks the ~~sublist main list, ignores the sub list, then returns. In this way, each level~~ sub-list and sets the ~~pointers~~ indicies of matches if found. then it recurses on the halves of the sub array, stoping at size = 1