COMP 6651 – ADT - Assignment – 1

Student Name :Numan Salim Shaikh
Student ID : **40266934**

Q1:
A:
FALSE
By the definition of O, $f(n) \le cg(n)$ for $c>0$ for some $c > 0$ and n_o such that $n \ge n_o$ i.e. the function g(n) grows faster than f(n) but doesn't imply that f(n) grows faster than g(n).
<u>For example:</u>
$f(n)=n$ and $g(n)=n^2$. This satisfies the condition $f(n) = O(g(n))$ as $n = O(n^2)$ but $g(n) = O(f(n))$ as $n^2 \neq O(n)$. So, $f(n) = O(g(n))$ doesn't imply $g(n) = O(f(n))$.

B:
FALSE
$f(n)=O((f(n))^2)$
From the definition of O, $f(n) \le cg(n)$ for some $c > 0$ and n_o such that $n \ge n_o$

Applying the rule,
$f(n) \le c((f(n)^2))$. This holds true for $f(n) \ge 1$ however this fails when $f(n) < 1$.
<u>For example:</u>
$f(n)=5$ then $5 \le c.5^2$.
$f(n)=-3$ then $-3 \le c.3^2$. For this to hold true, the value of c should be negative, which is against the definition of O.

C:
FALSE
$f(n)=\Theta(f(n/2))$
By the definition of Θ, we know that
$c_1g(n) \le f(n) \le c_2g(n)$ for some $c_1,c_2 > 0$ and n_o such that $n \ge n_o$

Applying the rule,
$c_1f(n/2) \le f(n) \le c_2f(n/2)$. Consider $f(n) = 2^{2n}$, then $f(n/2) = 2^n \Rightarrow c_1.2^n \le 2^{2n} \le c_2.2^n$. For this to hold true, there should exist a constant c_2 such that, $2^n \le c_2$. But this is impossible as 2^n increases exponentially and will exceed any constant c_2.

D:
TRUE
$f(n)+o(f(n)) = \Theta(f(n))$.
Assume, $g(n) = o(f(n))$. From the definition of o, $f(n) < cg(n)$ for some $c > 0$ and n_o such that $n \ge n_o$ i.e. $0 \le g(n) < c(f(n))$
Adding f(n) on both sides of equation,

$f(n) \le f(n) + g(n) < f(n)+ c.f(n)$
$f(n) \le f(n) + o(f(n)) < (1+c).f(n)$. So, from the definition of o,
we can say, $f(n)+o(f(n)) = \Theta(f(n))$

Q2:

Recurrence Equation is $T(n)=4T(n/2)+n$

(a) Failed Proof with $T(n) \le cn^2$

Let's try proving by substitution that $T(n) \le cn^2$

We are given:

$T(n) = 4T(n/2) +n$

Assume $T(n) \le cn^2$

Substituting this into the recurrence:

$$T(n) \le 4 (c(n/2)^2) + n$$

$$= 4(cn^2/4) + n$$

$$= cn^2 + n$$

So, we get:

$T(n) \le cn^2 + n$

This fails to satisfy $T(n) \le cn^2 + n$, because the extra ‘n’ term prevents the inequality from holding for large ‘n’.

Thus, the assumption $T(n) \le cn^2 + n$, does not work.

(b) Correcting the Proof:

We can attempt to subtract a lower-order word in order to make the substitution proof work. Let us deduct from T(n) a term proportional to n:

Assume $T(n) = cn^2 - bn$, where $c>0$ and $d>0$.

Substituting this in the recurrence:

$T(n) = 4T(n/2) + n$
$\leq 4(c((n/2)^2) - bn/2)+n$
$= 4(cn^2/4 - bn/2)+n$
$= cn^2 - 2bn +n$
$= cn^2 – bn – (b-1) n$
$\leq cn^2 - bn$ if and only if $(b-1)n$ is positive.

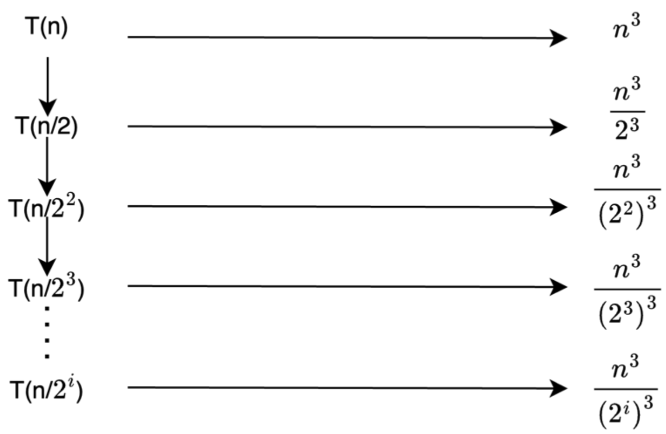
By subtracting the lower term, the substitution proof works and we got $T(n) = \Theta(n^2)$.

Therefore, by subtracting the lower order term dn, the substitution proof works and

we have $T(n) = \Theta(n^2)$

Q3:

(a)



Here, the recurrence continues until the subproblem reaches 1.
i.e. $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow k=c$
The height of the tree is $\log_2(n)$.
Total Cost $T(n) = n^3 + n^3/2^3 + n^3/(2^2)^3 + n^3/(2^3)^3 + \ldots\ldots + n^3/(2^i)^3$
$= \sum_1^{\log n} n^3 /((2^i)^3)$
$< \sum_1^{\infty} n^3 /((2^i)^3)$
So, using the formula of Geometric progression for infinite series with $a= n^3$ and $r=1/2^3$. Then, we get:
$= n^3 / (1 – 1/8)$

$= n^3 \ / \ (7 / 8)$
$= (8/7)\ n^3$

So, $T(n) = 0\ (n^3)$

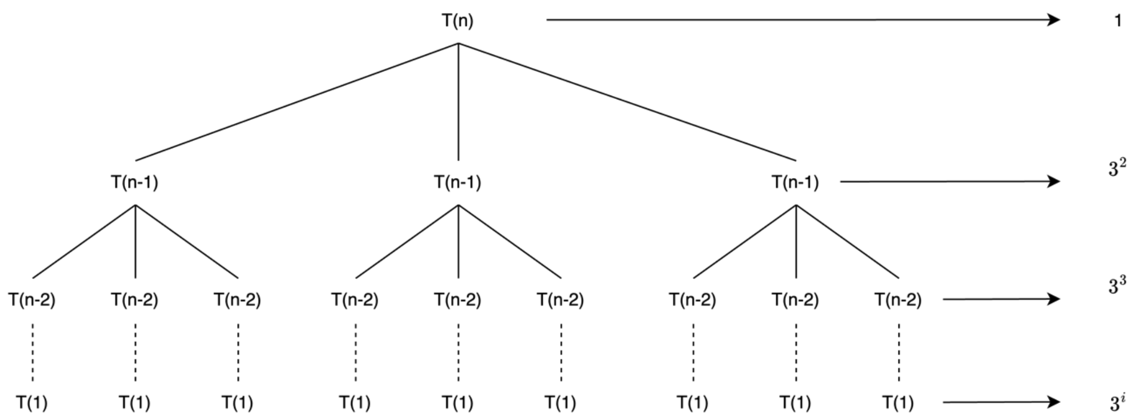
<u>Proof by substitution method:</u>

From the definition of O, $T(n) \leq cn^3$. Substituting the value in given recurrence relation,

$T(n) = T(n/2)+n^3$
$\leq cn^3/8+n^3\ =.\ n^3\ (c/8+1)$

For a positive value of c/8+1 => c > -8, it holds true for any value. So, the solution is correct.

(b)



Here, the recurrence continues until the subproblem reaches 1.
i.e. n - i=1 => i=n – 1
The height of the tree is n-1.
Total Cost $T(n) = 1+3^2+3^3+\ldots\ldots+3^i$
$= \sum_1^i 3^i$

So, using the formula of Geometric progression for finite series with a= 1 and r=3. Then, we get:

$= a(r^n-1)/(r-1)$
$= 1(3^n-1)/(3-1)$
$= (3^n-1)/2$

To find the upper bound O, we can say that it depends on the exponential growing value of 3^n

$T(n)= O(3^n)$

<u>Proof by substitution method:</u>

From the definition of O, $T(n) \leq c3^n$. Substituting the value in given recurrence relation,

$T(n) = 3T(n-1)+1$

$\leq 3c3^{n-1}+1$

$=c3^n+1$

So, for a sufficiently large n and appropriate value of c, we can say:

$T(n)=O(3^n)$

Q4:

(a) T(n) = T(n/2) + T(n/3) + T(n/6) + n lg n.

f(x) = n lg n

So, to find the value of p,

$(1/2)^p + (1/3)^p + (1/6)^p = 1$

For p=1,

(1/2) + (1/3) + (1/6) = 1

Therefore, the value of p is 1

Using Akra-Bazzi method:

$$T(n) = \theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\, dx\right)\right)$$

$$T(n) = \theta\left(n^1 \left(1 + \int_1^n \frac{xlgx}{x^2}\, dx\right)\right)$$

$$T(n) = \theta\left(n \left(1 + \int_1^n \frac{lgx}{x}\, dx\right)\right)$$

$$T(n) = \theta\left(n \left(1 + \left[\frac{lg^2x}{2}\right]_1^n\right)\right)$$

$$T(n) = \theta\left(n \left(1 + \left[\frac{lg^2n}{2} - \frac{lg^21}{2}\right]\right)\right)$$

$$T(n) = \theta\left(n \left(1 + \frac{lg^2n}{2}\right)\right)$$

$$T(n) = \theta\left(n + \frac{nlg^2n}{2}\right)$$

$$T(n) = \theta(nlg^2n)$$

(b)

T(n) = (1/3)T(n/3) + 1/n.

f(n) = 1/n

So, to find the value of p,

$(1/3)(1/3)^p = 1$

$(1/3)^{p+1} = 1$

$3^{-p-1} = 1$

$3^{-p-1} = 3^0$

-p-1 = 0

P = -1

Therefore, the value of p is -1

Using Akra-Bazzi method

$$T(n) = \theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}}\, dx\right)\right)$$

$$T(n) = \theta\left(n^{-1} \left(1 + \int_1^n \frac{1/x}{x^0}\, dx\right)\right)$$

$$T(n) = \theta\left(\frac{1}{n} \left(1 + \int_1^n \frac{1}{x}\, dx\right)\right)$$

$$T(n) = \theta\left(\frac{1}{n} (1 + [logx]_1^n)\right)$$

$$T(n) = \theta\left(\frac{1}{n} (1 + [logn - log1])\right)$$

$$(n) = \theta\left(\frac{(1 + logn)}{n}\right)$$

Q5 :

a)To solve this problem with only O(n logn) invocations of the equivalence tester, we can use a divide-and-conquer approach similar to the majority element algorithm. The idea is to recursively divide the set of cards, test for equivalence in each recursive step, and combine the results to find a majority equivalent set.

Algorithm to find the majority set

FindMajority(A, l , r) :

```
If (l == r) then:
        return A[l]
```

```
    mid = ( l+r) /2

    //recursively find majorities in both halves
    left_majority = FindMajority(A, l, mid)
    right_majority = FindMajority(A, mid+1, r)

    if left_majority == right_majority then
        return left_majority

    left_count = countFrequency(X, l, r, left_majority)
    right_count = countFrequency(X, l, r, right_majority)

    //majority element exists in the left half
    if left_count > (r-l+1)/2 then
        return left_majority

    //majority element exists in the right half
    else if right_count > (r-l+1)/2 then
        return right_majority
    else
        return -1


countFrequency(A, l, r, value) :
    count = 0
    for i= l to r do
        //calling the equivalence_tester
        if equivalence_tester(A[i], value) then
            count = count +1
        end if
    end for

    return count
```

b)
Correctness of the Algorithm

We will prove that the algorithm works correctly, using a proof by induction. For the base case, consider an array/collection of 1 element (which is the base case of the algorithm). Such a collection has the majority element, which is the first and the only element, so the base case is correct.

For the inductive step, suppose that *FindMajority* will correctly find the majority element on any array of length less than n. Suppose we call *FindMajority* function on an array of size n. It will recursively call *FindMajority* on two arrays of size n/2. By the induction hypothesis, these calls will correctly find the majority element in both these arrays. Hence, after the recursive calls, we will be able to find the majority element in both or any one half of the subarray between indices l,….mid and mid+1,…..r respectively, if the majority element exists. Note:- It can also happen that upon combining the subarray, we get the majority element.
Now, we have shown in our algorithm that if the majority element exists, the count of those elements will be greater than half the length of the array, hence after executing it, we will get the majority element if there exists any between l and r.
This concludes our proof.

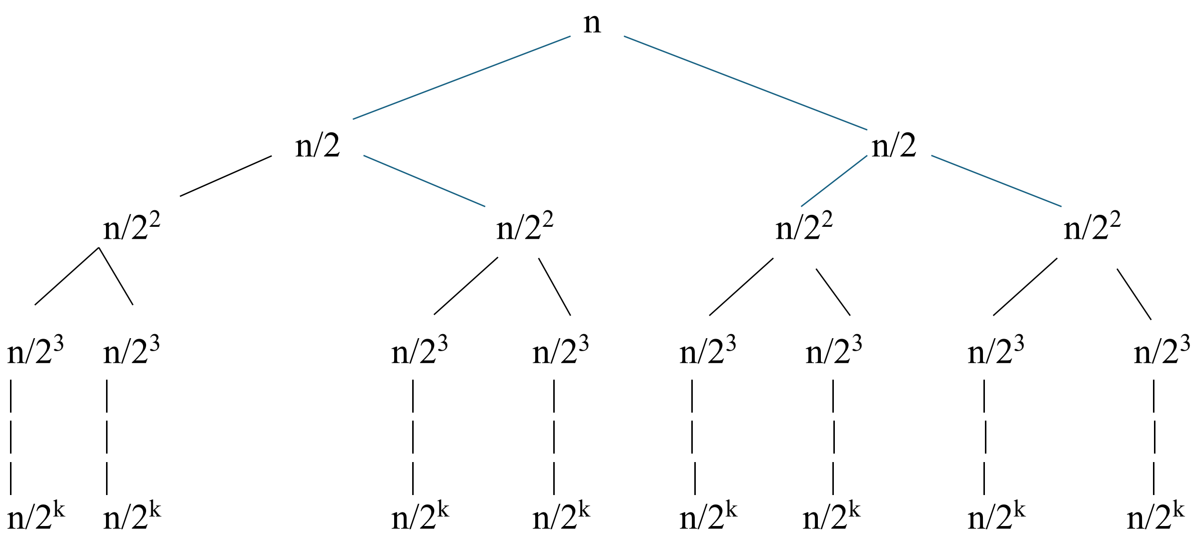
c) In the given algorithm, the *equivalence_tester* is called for each element in the array for every range considered in the recursive calls. Also, the algorithm makes two recursive calls with arrays of size n/2 each.

Let T(n) be the number of invocations of equivalence_tester for an array of size n.
The recurrence relation can be expressed as follows:

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

Using a recursion tree method to generate a guess for the time complexity.

Diagram :



So, there are a total of k steps and at each step it is taking n amount of time.
Therefore, total time taken is n*k.

We assume that $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

Therefore, the total time taken is n*k = n*log n
So, T(n) = O(n log n)

Now, let's use substitution method to prove the same.
$T(n) = 2T(n/2) + n$
$\Rightarrow T(n) = 2[2T(n/2^2)+n/2] + n$ *//Substituting T(n/2)*
$\Rightarrow T(n) = 2^2\, T(n/2^2) + n + n$
$\Rightarrow T(n) = 2^2\, [2T(n/2^3) + n/2^2] + 2n$ *//Substituting $T(n/2^2)$*
$\Rightarrow T(n) = 2^3\, T(n/2^3) + 3n$

Repeating this for k times, we get
$T(n) = 2^k\, T(n/2^k) + kn$ …….Eq 1
We assume that $T(n/2^k) = T(1)$

Therefore, $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$
Substituting $n = 2^k$ and $k = \log n$ in Eq 1, we get
$T(n) = 2k\, T(1) + kn$
$T(n) = n*1 + n*\log n$ *Since, T(1) = 1*
$T(n) = n + n \log n$

=> T(n) = O(n log n) Hence proved

Q6:

(a)

To verify the correctness of the STOOGE-SORT algorithm, we can examine different cases:

Case 1: If p = r, the subarray contains only one element, which is already sorted.

Case 2: If p+1= r, the subarray has two elements, and the comparison and possible swap will sort them.

Case 3: For subarrays with more than two elements, the algorithm first checks if the first and last elements are in order, swapping them if necessary. It then performs three recursive calls: sorting the first two-thirds, then the last two-thirds, and finally re-sorting the first two-thirds to ensure everything is correct.

For an array of size nnn, the initial check ensures the outer elements are sorted, while the recursive steps divide the array into smaller parts. The final call to re-sort the first two-thirds fixes any issues that may arise from sorting the last two-thirds. By induction, we can conclude that STOOGE-SORT correctly sorts any array A[1...n]

(b)
We can define the recurrence relation for the given problem in the following way:

T(n) =3T(2n/3)+O(1) (Approximate size of subarray in first and last recursive call is (2/3)n as the size of array is reduced by 1/3 each time and O(1) as a constant time for comparisons).

To find the time complexity, we can use the master's theorem:

T(n)=a. T(bn)+f(n)
where a=3, b=3/2 and f(n)=O(1).

Computing $n^{\log_b a}$:
First computing $\log_b a$: $\log_{3/2} 3 = \log 3/\log(3/2) = \log_{3/2} 3$

$n^{\log_b a} = \Theta(n^{\log_{3/2} 3})$

Since $f(n) < n^{\log_{3/2} 3})$, using the case 1 of Master Theorem:

$$
\begin{aligned}
T(n) &= 3T(2n/3) + \Theta(1) \\
&= (n^{\log_{3/2} 3}) \\
&= O(n^{(\log 3/\log 1.5)}) \\
&\approx O(n^{2.709})
\end{aligned}
$$

(c)
Stooge-Sort of is remarkably the most inefficient sorting algorithm I have seen. Hence, the professors do not deserve tenure.
Insertion-Sort : $O(n^2)$
Merge-Sort: $O(n \lg n)$
Heapsort: $O(n \lg n)$
Quicksort: $O(n^2)$
Stooge-Sort: $O(n^{2.709})$

Q7 :

Algorithm to find median

Let DB1, DB2 be the two databases.
pointer1 = pointer2 = n/2

for i=2 to log n do

median1 = *QuerytoDb*(DB1, pointer1)
median2 = *QuerytoDb*(DB2, pointer2)

if median1 > median2 **then**
pointer1 = pointer1 – $n/2^i$ *//query the bottom half of DB1*
pointer2 = pointer2 + $n/2^i$ *//query the top half of DB2*
else

pointer1 = pointer1 + $n/2^i$
pointer2 = pointer2 - $n/2^i$

end if

end for

return *min(median1, median2)*

Explanation :

pointer1 and *pointer2* are two query pointers for each database in the procedure above.
To get median1 and median2, we first query the medians of both databases. We demonstrate that the joint database's median needs to fall between m1 and m2.

In order to notice this, note that at least n records in each of DB1 and DB2 are less than or equal to max(m1, m2). Therefore, the joint database's median does not exceed max(m1, m2). Similarly, we can demonstrate that the joint database's median is equal to min(median1, median2). The points p1 and p2 can then be moved appropriately. Since median1 and median2 are the nth and (n + 1)th lowest numbers in the joint database at the end of the loop, we return the smaller of median1, median2.

Let T (n) be the total number of queries.

As each round we reduce the problem size by half using two queries, we have

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

Solving this recurrence we obtain

$T(n) = O(\log n)$