# COM S 311 HOMEWORK 1

NATHAN TUCKER (NJTUCKER@IASTATE.EDU)

This assignment represents my own work in accordance with University regulations. Collaboration done with Haadi Majeed (hmajeed@iastate.edu) and Matthew Hoskins (mgh@iastate.edu)

- Nathan Tucker

# Homework 1 Submission

(1) Prove or disprove the following statements.

(a) $n^2 - 10n + 2 = O(n^2)$.

$f(n) = n^2 - 10n + 2$, $g(n) = n^2$

We set c = 2 and n = 1

$(1)^2 - 10(1) + 2 \leq 2 * (1)^2$

$-7 \leq 2$

and this will hold true for all values of n greater than $n_0$

(b) $2^{n^2} = O(2^{2n})$.

$f(n) = 2^{n^2}$, $g(n) = 2^{2n}$

We can set c = 1 and n = 4 and we get $f(n) = 65536$ and $g(n) = 256$, and as you get larger and larger n values, this difference grows, as the power of $f(n)$ is polynomial versus the linear power of $g(n)$. This, the inequality $f(n) \leq c * g(n)$ does not hold for $n > n_0$ for any c.

Disproven.

(c) $n \log_2(n) = O(n \log_{10}(n))$.

We can set c = 2 in this case

For c = 2, there is no value of n such that

$log_2(n) \leq log_{10}(n)$

for all values of n greater than $n_0$

Disproven

(d) $n \log_2(n) = O(n)$. We can set c = 1 in this case. This yields the inequality

$n \log_2(n) \leq n$

Dividing by n gives us

$log_2(n) \leq 1$

There is no value of n that will satisfy this for all values of n greater than $n_0$ as the statement will grow and the constant will stay constant

Disproven

(e) $n2^n = O(2^{2n})$.

As per the limit definition, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ means $f(n) = O(g(n))$

$\lim_{n\to\infty} \frac{n2^n}{2^{2n}} = 0$

Therefore, we can determine that $n2^n = O(2^{2n})$

(2) Consider the following algorithms (written in pseudocode):

```
1 Alg1(A)
  | Input: Array of integers of length n
2 | constant number of operations
3 | for i = 1 to n do
4 | | constant number of operations
5 | for j = n to 1 do
6 | | for k = j to 1 do
7 | | | constant number of operations
```

```
1 Runtime of each line
2 | c
3 | n + 1
4 | c * n
5 | n + 1
6 | n * (n + 1)
7 | c * n * n
```

We can sum up the lines of the equation to yield

$$c * n^2 + n^2 + 1 + n + 1 + c * n + n + 1 + c$$
$$= (c + 1) * n^2 + (c + 2) * n + c + 3$$

As the largest leading term in the above runtime is a constant value times $n^2$ we can also say that the big O is $O(n^2)$

```
1 Alg2(A)
  | Input: Array of integers of length n
2 | for i = n to 1 do
3 | | constant number of operations
4 | | i = i/2
```

Looking at the algorithm, we can see that we are dividing n by 2 in each operation, or making n half of its original size with each iteration. If you were to express this as a summation it could be expressed as:

$$\sum_{i=1}^{n} n/2$$

This summation yields us an overall Big O of $O(logn)$ with an exact runtime of $log(c * n)$, the c value being the result of the number of constant time operations performed.

(3) Given an array $A$ of integers, we say that an integer $k$ is the *majority element of A* if $k$ occurs in $A$ strictly more than $A$.length /2 times. Give a Divide and Conquer algorithm which, given an array $A$, determines if $A$ contains a majority element. If $A$ does contain a majority element $k$, it outputs $k$, otherwise, it outputs "null". Formally analyze the runtime of your algorithm, giving a recurrence relation and a big oh bound on the runtime of your algorithm. You **must** use a divide and conquer strategy. You do not have to prove correctness.

```
1  MajorityElement(A, p, q)
       Input: Array of integers of length n, the left most index p, the right most index q
       /* Arrays of size 1 and 2 handling (base case) */
2      if p == q then
3          return "null"
4      if p + 1 == q then
5          return A[p]
6      LeftMajorityElement = GetMajorityElement(A, p, ⌊A.size/2⌋)
7      RightMajorityElement = GetMajorityElement(A, ⌊A.size/2⌋, q)
       /* We have ended the recursion, now we need to find how many times we can
          find the majority element of array size 1 */
8      LeftMajorityElementCount = GetNumberOfOccurrences(A, LeftMajorityElement)
9      if LeftMajorityElementCount > (q−p)/2 then
10         return LeftMajorityElement
       /* If we did not find a majority in the left, how about the right?  */
11     RightMajorityElementCount = GetNumberOfOccurrences(A, RightMajorityElement)
12     if RightMajorityElementCount > (q−p)/2 then
13         return RightMajorityElement
       /* What if there is no majority element in the left or the right?  */
14     return "null"
```

To formally analyze the runtime, we can see there are 2 things that depend on the size of the input array A. Firstly, we are recursively splitting A into 2 equal halves until it is an

array of size 1. This can be expressed as

$$\sum_{i=1}^{n} n/2$$

leading us to have a runtime so far of $log(n)$. Inside these recursive calls we make 2 calls to a function called GetNumberOfOccurrences which takes in an Array and a number of times we want to find an item. In this case, the MajorityElement we found in the lines prior. This is searching an array linearly, so each call is an $O(n)$ operation multiplied by each recursive call of $log(n)$ yielding a total Big O of $O(n \log n)$. As stated, we are splitting the array into 2 subarrays that are half the size of the input array. Then we do a linear search to find the majority element of an array. Because of this, the recurrence relation can be expressed as

$$T(n) = 2T(\tfrac{n}{2}) + n$$

(4) Using the Master Theorem, bound the runtime $T(n)$ of the following recurrence.

$$T(n) = 3T(\tfrac{n}{2}) + n^2, \text{ where } T(1) = O(1).$$

You must state which case of the Master Theorem holds, and prove that it does apply.

$$\text{a} = 3,\ \tfrac{n}{b} = \tfrac{n}{2},\ f(n) = n^2$$

$$log_b a = log_2 3 \approx 1.585$$

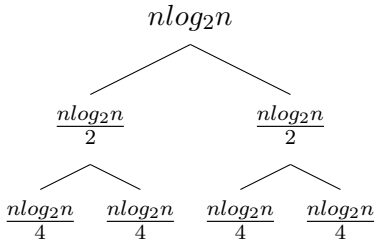
As this is smaller than our value of 2, we know that we would need to add some $\epsilon$ to our value of $n^{log_b a}$, leading us to believe this is case 3 of the Master Theorem

$$3T(\tfrac{n}{2}) + n^2 \Rightarrow \Theta(n^2) \text{ (Case 3)}$$

(5) Using the recurrence tree method, solve the following recurrence:

$$T(n) = 2T(n/2) + n\log_2(n), \text{ where } T(1) = O(1).$$

You do not have to draw the tree (you of course can, if you want). You can, instead, state clearly the sum of each level of the tree, and then rigorously bound the sum of each level using big oh notation.




The depth of the tree can be seen as $\frac{n}{2^i}$ and doing some rearranging, this yields a depth of $log_2n$. The sum of each row is shown to be fractions added up together to equal...just $n * logn$. To solve, we add up each row $log_2n$ times, which can be expressed as

$$\sum_{i=1}^{log_2n} n * log_2n$$
$$\Rightarrow n * log_2n * \sum_{i=1}^{log_2n} 1$$
$$\Rightarrow n * log_2n * (log_2n + 1)$$

(6) Professor Caesar wishes to develop an integer-multiplication algorithm that is asymptotically faster than Karatsuba's $O(n^{\log_2(3)})$ algorithm. His algorithm will use the divide-and-conquer method, dividing each integer into pieces of size n/4, and the divide and combine steps together will take O(n) time. He needs to determine how many subproblems his algorithm has to create in order to beat Karatsuba's algorithm. If his algorithm creates $a$ (an integer) subproblems, then the recurrence for the running time $T(n)$ becomes $T(n) = aT(n/4) + n$. What is the largest integer value of $a$ for which Professor Caesar's algorithm would be asymptotically faster than $O(n^{\log_2(3)})$? Justify your answer.

$$a =?, b = 4, T(n) = aT(\tfrac{n}{4}) + n$$

Master Theorem case 1 states that

$$\text{If } a > b, \text{ then } T(n) = \theta(n^{log_b a})$$

Therefore, we can say that we require $log_4 a$ to be less than $log_2 3$ which is $\approx 1.58496$

$$log_4 a = log_2 3$$
$$\Rightarrow a = 4^{log_2 3}$$
$$\Rightarrow 4^{log_2 3} = 9$$

So we can conclude that the largest integer value for a for which Professor Caesar's algorithm would be asymptotically faster than $O(n^{log_2 3})$ is 8