CS3343 Analysis of Algorithms Fall 2017

Homework 2

Due 9/15/17 before 11:59pm (Central Time)

1. Master theorem (6 points)

Use the master theorem to find tight asymptotic bounds for the following recurrences. If the master theorem cannot be applied, state the reason and give an upper bound (big-Oh) which is as tight as you can justify. Justify your answers (by showing which case of the master theorem applies, and why.)

(1) $T(n) = 9T(n/3) + \sqrt{n}$
$Case\ 1:$

$$\begin{aligned}\sqrt{n} = n^{\frac{1}{2}} &\in O(n^{\log_3 9-\epsilon}) \\ &= n^{\frac{1}{2}} \in O(n^{2-\epsilon}) \text{ for } \epsilon = 1 \\ &= n^{\frac{1}{2}} \in O(n)\end{aligned}$$

$n^{\frac{1}{2}} \leq C \cdot n$ for $C = 1$ and $n_0 = 1$ so $T(n) = \Theta(n^2)$

(2) $T(n) = 2T(n/2) + n \log n$
$Case\ 2:$

$$\begin{aligned}n \log n &= \Theta(n^{\log_2 2} \log^k n) \text{ for } k = 1 \geq 0 \\ &= \Theta(n \log n)\end{aligned}$$

$T(n) = \Theta(n \log^{k+1} n)$ for $k = 1 \Rightarrow T(n) = \Theta(n \log^2 n)$

(3) $T(n) = 36T(n/6) + n^2$
$Case\ 2:$

$$\begin{aligned}n^2 &= \Theta(n^{\log_6 36} \log^k n) \text{ for } k = 0 \geq 0 \\ &= \Theta(n^2)\end{aligned}$$

$T(n) = \Theta(n^2 \log^{k+1} n)$ for $k = 0 \Rightarrow T(n) = \Theta(n^2 \log n)$

(4) $T(n) = T(2n/5) + n$
$Case\ 3:$

$$\begin{aligned}n &= \Omega(n^{\log_{2.5} 1+\epsilon}) \text{ for } \epsilon = 1 \\ &= \Omega(n^1) \\ &= \Omega(n)\end{aligned}$$

$n \geq C \cdot n$ for $C = 1$ and $n_0 = 1$
$Regularity: 1 \cdot \frac{2n}{5} \leq C \cdot n$ for $C = \frac{2}{5} < 1$
Both conditions are satisfied so $T(n) = \Theta(n)$

(5) $T(n) = T(2n/3) + T(n/3) + n$
The two subproblems created in the recurrence are of difference sizes so the master theorem does not apply here.
With a recursion tree, the largest height is $\log_{1.5} n$ and a cost of n at each level. So a reasonable bound is $O(n \log n)$

(6) $T(n) = T(n-2) + n$
b is not > 1 so master theorem does not apply here.
Using substitution and $T(1) = d_0$:
Stops when $n - 2x = 1$ where T(n) recurses $x = \frac{n-1}{2}$ times.
So roughly has a runtime of $T(n) = d_0 + (\frac{n-1}{2})n = O(n^2)$

2. Recursive Program (5 points)

Consider the following recursive function for $n \geq 0$:

Algorithm 1 int recFunc(int n)

```
//Base Case:
if n <= 2 then
    return n;
end if
//Recursive Case:
i = 0;
while i < 100 do
    print("Hello!");
    i = i + 1;
end while
int a = 2*recFunc(n/2); //Integer division
j = 0
while j < a do
    print("Bye!");
    j = j + 1;
end while
return a;
```

(1) Use strong induction to prove the value returned by recFunc(n) is $\leq n$.
BaseCases : $(0 \leq n \leq 4)$
For $n = 0, 1$, or 2 the function immediately returns n so trivially recFunc(n) returns a value $\leq n$.
Inductivestep :
Assume: recFunc(k) returns a value $\leq k$ for all $0 \leq k < n$.
Prove: recFunc(n) returns a value $\leq n$.
If n is odd then $n = 2m + 1$ for some positive integer $m < n$.
So recFunc(n) returns 2· recFunc($\frac{2m+1}{2}$) = 2·recFunc(m) because of integer division. Since $m < n$, under the inductive hypothesis, recFunc(n) $\leq n$.

Similarly, if n is even $n = 2m$ for some positive integer $m < n$.
So recFunc(n) returns 2· recFunc($\frac{2m}{2}$) = 2·recFunc(m). Since $m < n$, under the inductive hypothesis, recFunc(n) $\leq n$.
This covers all cases of $n \geq 0$, thus by strong induction recFunc(n) returns a value $\geq n$.

(2) Set up a runtime recurrence for the runtime $T(n)$ of this algorithm.

$$T(n) = \begin{cases} 1 & : 0 \leq n \leq 2 \\ 3T(n/2) + 100 & : n > 2 \end{cases}$$

(3) Solve this runtime recurrence using the master theorem.
$Case1$:

$$\begin{aligned} 100 &= O(n^{\log_2 3 - \epsilon}) \text{ for } \epsilon = 1 \\ &= O(n^{1.585-1}) \\ &= O(n^{0.585}) \end{aligned}$$

$100 \leq C \cdot n^{0.585}$ for $C = 100$ and $n_0 = 1$, so $T(n) = \Theta(n^{\log_2 3})$

3. Big-Oh Induction (4 points)

Use strong induction to prove that $T(n) \in O(\log n)$ where $T(1) = 90$ and $T(n) = T(n/2) + 10$ (for $n \geq 2$).

Show $T(n) \leq C \cdot \log n$ for $n \geq 2$.

Base Case ($n = 2$):
LHS: $T(2) = T(2/2) + 10 = T(1) + 10 = 90 + 10 = 100$
RHS: $C \cdot \log 2 = C \cdot 1$
Choose $C = 100 \Rightarrow T(2) = 100 \leq 100 \cdot 1$

Inductive Step:
Assume: $T(k) \leq C \cdot \log k$ for $2 \leq k < n$
Prove: $T(n) \leq C \cdot \log n$

$$\begin{aligned} T(n) = T(n/2) + 10 &\leq C \cdot \log n/2 + 10 \\ &\leq C[\log n - \log 2] + 10 \\ &\leq C \cdot \log n - C + 10 \text{ Choose } C \geq 10 \text{ make extra terms} \leq 0 \\ &\leq C \cdot \log n \end{aligned}$$

Choosing $C = 10$ and $n_0 = 2$ completes the inductive step, so $T(n) \in O(\log n)$

4. Recursive Algorithm and Analysis (6 points)

Suppose company X has hired you as a software engineer.

Company X is desperate to improve the run time of their software. Rather than allow you to use the standard binary search algorithm, company X demands you create a new version. Since recursively dividing the array into two parts is so efficient for searching, the bigwigs at company X conclude dividing the array into three parts will be even more efficient.

This new "3-ary" search divides the given sorted array into 3 equal sized sorted subarrays. It finds which subarray will contain the given value and then recursively searches that subarray. If the given value is in the array you should return true. Otherwise you should return false.

(1) (2 points) Write up this new recursive algorithm using pseudocode. (Hint: For simplicity you can assume the number of elements in the given sorted array, n, is a power of 3)

Algorithm 2 bool threeSearch(int $A[1 \ldots n]$, int val)

```
//Assuming integer division
if n == 1 then
    if A[1] == val then
        return true;
    else
        return false;
    end if
else if n == 2 then
    if A[1] == val or A[2] == val then
        return true;
    else
        return false;
    end if
else if val <= A[n/3] then
    return threeSearch(A[1...n/3], val);
else if val <= A[2*n/3] then
    return threeSearch(A[n/3+1...2*n/3], val);
else
    return threeSearch(A[2*n/3+1...n], val);
end if
```

(2) (1 point) Create a recurrence to represent the worst case run time of our new "3-ary" search from the pseudocode.
(Hint: The worst case run time occurs when the given value is not in the array)
$T(n) = T(n/3) + 1$

(3) (2 points) Use master theorem to solve your above recurrence relation.
$Case2:$

$$\begin{aligned} f(n) = 1 &= \Theta(n^{\log_3 1} \log^k n) \text{ for } k = 0 \\ &= \Theta(n^0) \\ &= \Theta(1) \end{aligned}$$

Clearly, $1 = \Theta(1)$, so $T(n) = \Theta(n^0 \log^{k+1} n)$ for $k = 0$, so $T(n) = \Theta(\log n)$

(4) (1 point) Compare this asymptotic run time to that of binary search. Does our new algorithm perform significantly better than binary search?
No, both algorithms actually perform asymptotically the same, only differing by constants. Binary search is $\Theta(\log_2 n)$ while the "3-ary" search is $\Theta(\log_3 n)$. Their difference is only by a constant factor of a logarithm which is insignificant in asymptotic analysis.

(5) (0 points - Just for fun) Try comparing the constants of binary search and "3-ary" search. You can use the change of base formula for log to convert both to $\log_2 n$.