CSCI-GA.1170-001/002 Fundamental Algorithms September 22, 2015

Solutions to Problem 1 of Homework 2 (14 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 22*

Consider the recurrence $T(n) = 8T(n/4) + n$ with initial condition $T(1) = 1$.

(a) (2 points) Solve it asymptotically using the "master theorem".

Solution:

From Master Theorem, $a = 8, b = 4$ and $f(n) = n$

$$\begin{aligned}
& f(n) = n \\
\implies & f(n) = O(n^{\log_4(8-4)}) \\
\implies & f(n) = O(n^{\log_b(a-\epsilon)}) \;\; \textit{with} \;\; \epsilon > 0 \\
\implies & T(n) = \Theta(n^{\log_b a}) \\
\therefore & T(n) = \Theta(n^{\log_4 8}) \\
& \quad\;\; = \Theta(n^{3/2})
\end{aligned}$$

□

(b) (4 points) Solve it by the "guess-then-verify method". Namely, guess a function $g(n)$ — presumably solving part (a) will give you a good guess — and argue by induction that for all values of n we have $T(n) \leq g(n)$. What is the "smallest" $g(n)$ for which your inductive proof works?

Solution:

Guess - $T(n) \leq cn^{3/2}$

Base Case - $1 = T(1) \overset{?}{\leq} c(1)^{3/2} = c \implies c \geq 1$

Induction - $T(n) = 8T(n/4) + n \overset{\text{ind}}{\leq} 8(c(n/4)^{3/2}) + n = cn^{3/2} + n \overset{?}{\leq} cn^{3/2} \implies$ not valid

$\therefore$ Our guess must be wrong. Hence we need to make another guess

Next Guess - $T(n) \leq cn^{3/2} - dn$

Base Case - $1 = T(1) \overset{?}{\leq} c(1)^{3/2} - d(1) = c - d \implies c \geq d + 1$

Induction - $T(n) = 8T(n/4) + n \leq 8(c(n/4)^{3/2} - d(n/4)) = cn^{3/2} - 2dn + n \overset{?}{\leq} cn^{3/2} - dn$
$\implies d \geq 1$ and $c \geq 2$, $\forall n \geq 2$

In the ideal case, $cn^{3/2} - dn$ has to be smallest $\implies$ c has to be smallest and d has to be greatest. But we give more importance to c being smaller as c is attached to $n^{3/2}$.
$\therefore$ for $d = 1$, $c \geq 1 + 1 = 2$

Hence the smallest $g(n)$ for which the inductive proofs works is $2n^{3/2} - n$

□

(c) (4 points) Solve it by the "recursive tree method". Namely, draw the full recursive tree for this recurrence, and sum up all the value to get the final time estimate. Again, try to be as precise as you can (i.e., asymptotic answer is OK, but would be nice if you preserve a "leading constant" as well).

Solution:

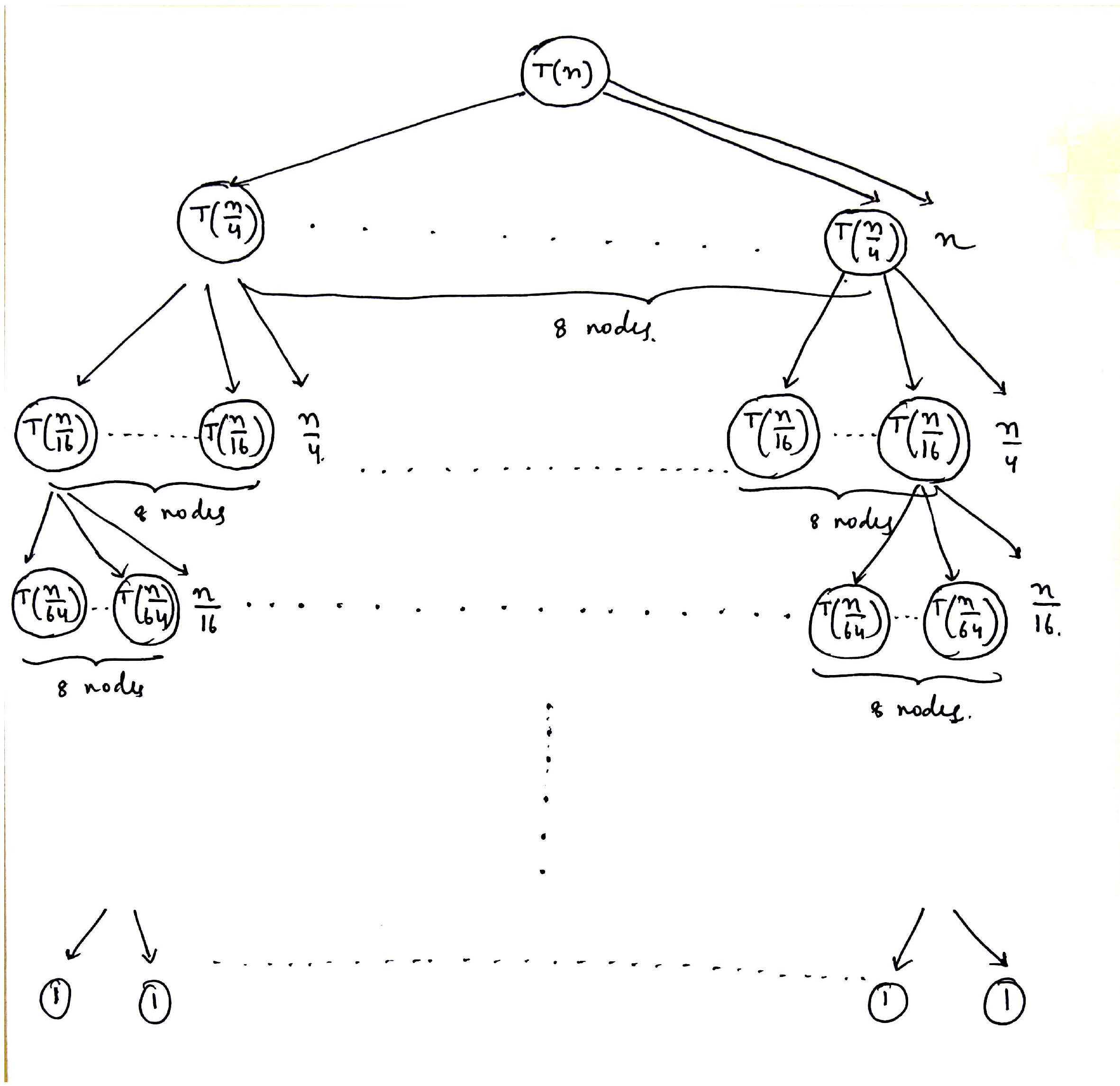

Figure 1: Recursive Tree of $T(n)$

Work done in first level is n
Work done in second level is $8(n/4) = 2n$
Work done in third level is $64(n/16) = 4n$
$\vdots$
Work done in k^{th} level is $2^{k-1}n$
Height of tree $= \log_4 n = \log_2 \sqrt{n}$
$\therefore$ Total number of computations is

$$\begin{aligned} \Sigma_k &= n + 2n + 4n + \ldots + 2^{k-1}n \\ &= n(1 + 2 + 4 + \ldots + 2^{k-1}) \\ &= n(\frac{2^k - 1}{2 - 1}) \\ &= n(2^k - 1) \\ \implies \Sigma_{\log \sqrt{n}} &= n(2^{\log n^{\frac{1}{2}}} - 1) \\ &= n(n^{1/2} - 1) \\ \therefore T(n) &= n^{3/2} - n \\ &= \Theta(n^{3/2} - n) \end{aligned}$$

$\square$

(d) (4 points) Solve it *precisely* using the "domain-range substitution" technique. Namely, make several changes of variables until you get a basic recurrence of the form $S(k) = S(k-1) + f(k)$ for some f, and then compute the answer from there. Make sure you carefully maintain the correct initial condition.

Solution: Let $n = 4^k$

$$\begin{aligned} T(4^k) &= 8T(4^k/4) + 4^k \\ T(4^k) &= 8T(4^{k-1}) + 4^k \\ Let \;\; S(k) &= T(4^k) \implies S(0) = 1 \\ \implies S(k) &= 8S(k-1) + 4^k \\ \frac{S(k)}{8^k} &= \frac{S(k-1)}{8^{k-1}} + (1/2)^k \\ Let \;\; P(k) &= \frac{S(k)}{8^k} \implies P(0) = 1 \\ P(k) &= P(k-1) + (1/2)^k \\ \implies P(k) &= 2 - (\frac{1}{2})^k \\ \implies S(k) &= 2.8^k - 4^k \end{aligned}$$

$$\implies T(4^k) = 2.(4^k)^{3/2} - 4^k$$
$$\implies T(n) = 2n^{3/2} - n$$

$\square$

(e) This part will not be graded. However, briefly describe your personal comparison of the above 4 methods. Which one was the fastest? The easiest? The most precise?

Solution:

The fastest and the easiest technique was to use Master Theorem. The most precise technique was to use Domain-Range substitution $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms	September 22, 2015

Solutions to Problem 2 of Homework 2 (10 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 22*

Consider the following recursive procedure.

BLA(n):
If $n = 1$ **Then Return** 1
Else Return BLA($n/3$) + BLA($n/3$)

(a) (3 points) What function of n does BLA compute (assume it is always called on n which is a power of 3)?

Solution:

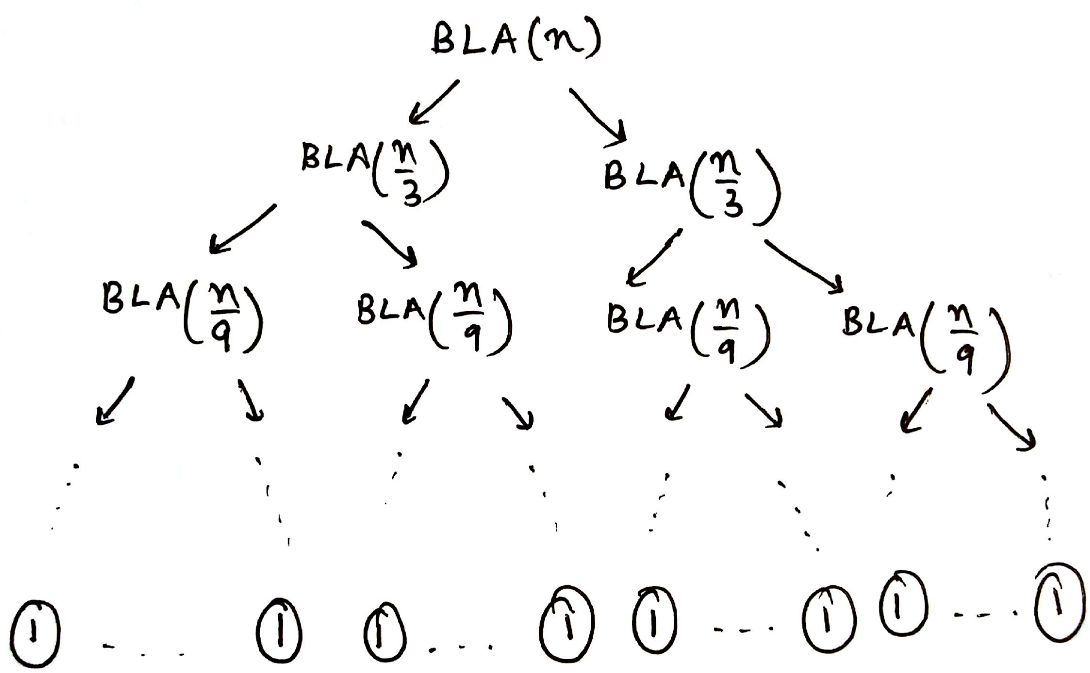

Figure 2: Recursive Tree of $BLA(n)$

In the above tree, at any k^{th} level, number of nodes are 2^k. The height of the tree is $\log_3 n$. Therefore at the last level, total number of $1's$ returned will be $2^{\log_3 n}$.
$\therefore BLA(n) = 2^{\log_3 n}$ $\square$

(b) (3 points) What is the running time $T(n)$ of BLA?

Solution: From the tree, we can see that the recurrence equation will be,

$$T(n) = 2T(n/3) + 1$$

Using Master Theorem, $a = 2, b = 3$ and $d = 0 \implies d < \log_b a$

$$\therefore T(n) = O(n^{\log_a b})$$
$$\implies T(n) = O(n^{\log_3 2})$$

□

(c) (4 points) How do the answers to (a) and (b) change if we replace the last line by "**Else Return** $2 \cdot$ BLA$(n/3)$"?

Solution: Answer to part(a) remains the same as computationally $BLA(n/3)+BLA(n/3) = 2BLA(n/3)$

However, the answer to part(b) changes as we are reducing the total number of executions by not making a call to $BLA(n/3)$ again and instead, multiplying $BLA(n/3)$ by 2. Therefore, the new recurrence equation will now be

$$\begin{aligned} T(n) &= T(n/3) + 1 \\ &= T(n/9) + 1 + 1 \\ &= \underbrace{1 + 1 + \ldots + 1 + 1}_{log_3 n \;\; terms} \\ \implies T(n) &= \log_3 n \\ &= \Theta(\log_3 n) \end{aligned}$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms	September 22, 2015
Solutions to Problem 3 of Homework 2 (16 points)	
Name: GOWTHAM GOLI (N17656180)	*Due: Tuesday, September 22*

Let $A[1 \dots n]$ be an array of pairwise different numbers. We call pair of indices $1 \le i < j \le n$ an *inversion* of A if $A[i] > A[j]$. The goal of this problem is to develop a divide-and-conquer based algorithm running in time $\Theta(n \log n)$ for computing the number of inversions in A.

(a) (8 points) Suppose you are given a pair of *sorted* integer arrays A and B of length $n/2$ each. Let C an n-element array consisting of the concatenation of A followed by B. Give an algorithm (in pseudocode) for counting the number of inversions in C and analyze its runtime. Make sure you also argue (in English) why your algorithm is correct.

Solution:

Given, C is concatenation of two sorted arrays A and B. Therefore, the first half and remaining half of C is sorted (similar to the *Merge* step in *Merge Sort*).

We know that the *Merge* step can be accomplished in $O(n)$ time. Hence by adding some $O(1)$ executions to the *Merge* step, it could be possible to count number of inversions in $O(n)$ time.

Approach

Let *mid* be the middle element of $C \implies C[0 \dots mid]$ and $C[mid + 1 \dots n]$ are sorted.

Therefore, there cannot be any inversions in $C[0 \dots mid]$ and $C[mid + 1 \dots n]$. The only inversions that can exist in C are the inversions that exist in between the two arrays $C[0 \dots mid]$ and $C[mid + 1 \dots n]$

Let $0 \le i \le mid$ and $(mid + 1) \le j \le n$. For an inversion to occur,
$C[i] > C[j] \implies C[i + 1 \dots mid] > C[j] \quad (\because C[i + 1 \dots mid] > C[i])$

Therefore, the number of inversions will be $mid - i + 1$. Then we keep increasing j to find next inversions until it reaches n

If $C[i] < C[j] \implies$ this is not an inversion. So we keep increasing i to find an inversion until it reaches *mid*

Hence we can see that the above explained algorithm can be easily included in the *Merge* step just adding a few lines of code.

In the following Psuedocode, *Merge&CountInversions(C, p , mid, k, D)*

- p is the starting index of C (in this case $p = 0$)
- *mid* is the mid-point of C
- k is the length of array C (in this case $c = n$)
- D is the final sorted array obtained by merging the two halves of C

$\therefore$ In the main function, *Merge&CountInversions(C, 0 , mid, k, D)* will be called

Pseudo Code

```
Algorithm: Merge&CountInversions(C, p , mid, k, D)
i ← p ; j ← mid+1
r ← 0
numInversions ← 0
while i ≤ mid and j ≤ n do
    if C[i] < C[j] then
        D[r] ← C[i]
        i++ ; r++
    end
    else
        D[r] ← C[j]
        numInversions = numInversions + (mid − i + 1)
        j++ ; r++
    end
end
while  i ≤ mid do
    D[r] ← C[i]
    r++ ; i++
end
while j ≤ n do
    D[r] ← C[j]
    r++ ; j++
end
return numInversions
```

Algorithm 1: Algorithm to count number of inversions in C in $O(\log n)$ time

Time Complexity

The above Psuedocode is identical to *Merge* except for the one line
numInversions = numInversions + (mid - i + 1)

As this line takes only $O(1)$ time, the Time Complexity of
Merge&CountInversions(C, 0 , mid, n, D) is $T(n) = O(n)$ □

(b) (8 points) Give an algorithm (in pseudocode) for counting the number of inversions in an n element array A that runs in time $\Theta(n \log n)$. Make sure you formally prove that your algorithm runs in time $\Theta(n \log n)$ (e.g., write the recurrence and solve it.)
(**Hint**: Combine Merge Sort with part (a).)

Solution:

Similar to Merge Sort, we can calculate the number of inversions in any array A by dividing the array into two equal halves $A[0 \ldots mid]$ and $A[mid + 1 \ldots n]$.

Let

- $count_1$ be the number of inversions in $A[0 \ldots mid]$
- $count_2$ be the number of inversions in $A[mid + 1 \ldots n]$
- $count_3$ be the number of inversions between $A[0 \ldots mid]$ and $A[mid + 1 \ldots n]$

$\therefore$ Total number of inversions in A is $count_1 + count_2 + count_3$,
where $count_3$ can be calculated using *Merge&CountInversions(A, p, mid, k, C)* assuming that $A[p \ldots mid]$ and $A[mid + 1 \ldots k]$ are sorted

Psuedocode

```
Algorithm: Sort&CountInversions(A, i, k)
if i is k then
    return 0
end
else
    mid ← (i+k)/2
    count_1 ← Sort&CountInversions(A, i, mid)
    count_2 ← Sort&CountInversions(A, mid+1, k)
    Create a new array C of length k − i
    count_3 ← Merge&CountInversions(A, i, mid, k, C)
    Copy C[0 ... k − i] to A[i ... k]
    return count_1 + count_2 + count_3
end
```

Algorithm 2: Algorithm to count number of inversions in A in $O(n \log n)$ time

Time Complexity

We know from part(a), *Merge&CountInversions* runs in $O(n)$ time. Copying C to A also takes only $O(n)$ time. Therefore the recurrence equation is

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

Using Master Theorem, $a = 2, b = 2, d = 1 \implies d = \log_b a$

$$\therefore T(n) = O(n^d \log n)$$
$$\implies T(n) = O(n \log n)$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms	September 22, 2015

Solutions to Problem 4 of Homework 2 (9 points)

Name: GOWTHAM GOLI (N17656180) *Due: Tuesday, September 22*

Solve the following recurrences using any method you like. If you use "master theorem", use the version from the book and justify why it applies. Assume $T(1) = 2$, and be sure you explain every important step.

(a) $T(n) = T(9n/10) + n$.

Solution: From Master Theorem, $a = 1, b = 10/9$ and $f(n) = n$

$$\begin{aligned} & f(n) = n \\ \implies & f(n) = \Omega(n^{\log_{10/9}(1+9/10)}) \\ \implies & f(n) = \Omega(n^{\log_b(a+\epsilon)}) \;\; with \;\; \epsilon > 0 \\ \implies & T(n) = \Theta(f(n)) \\ \therefore & T(n) = \Theta(n) \end{aligned}$$

□

(b) $T(n) = 2T(n/2) + n \log n$.

Solution: From Master Theorem, $a = 2, b = 2$ and $f(n) = n \log n$

$$\begin{aligned} & f(n) = n \log n \\ \implies & f(n) = \Theta(n^{\log_2 2} \log^1 n) \\ \implies & f(n) = \Theta(n^{\log_b a} \log^k n) \;\; with \;\; k = 1 \\ \implies & T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \\ \therefore & T(n) = \Theta(n \log^2 n) \end{aligned}$$

□

(c) $T(n) = T(\sqrt{n}) + 1$. (**Hint**: Substitute ... until you are done!)

Solution:

Let $n = 2^k \implies T(2^k) = T(2^{k/2}) + 1$

Let $S(k) = T(2^k) \implies S(0) = T(1) = 2$ and $S(k) = S(k/2) + 1$

Let $k = 2^x \implies S(2^x) = S(2^{x-1}) + 1$

Let $P(x) = S(2^x) \implies P(0) = S(1) = c$ and

$$\begin{aligned}
P(x) &= P(x-1) + 1 \\
\implies P(x) &= x + S(0) \\
\implies S(2^x) &= x + c \\
\implies S(k) &= \log_2 k + c \\
\implies T(2^k) &= \log_2 k + c \\
\implies T(n) &= \log_2(\log_2 n) + c \\
\therefore T(n) &= O(\log(\log n))
\end{aligned}$$

$\square$