CMPS 101: Spring 2016: HW 3

Due: 27th April 2016

- All assignments must be submitted through git. Please look at the Piazza guide on submitting assignments.
- The assignment is to be attempted in groups of two. If you choose to not work with a partner, one point will be automatically deducted from your score.
- LaTeX is preferred, but neatly handwritten and scanned solutions will also be accepted. Please submit as PDF.
- Solutions to the problems should be clearly labeled with the problem number.
- Although no points are given for neatness, illegible and/or poorly organized solutions can be penalized at the graders option.
- Clearly acknowledge sources, and mention if you discussed the problems with other students or groups. In all cases, the course policy on collaboration applies, and you should refrain from getting direct answers from anybody or any source. If in doubt, please ask the instructors or TAs.
- There are some extra credit questions.

Q1 (1.5 points): Solve the following recurrences and state the time complexity in O. In all cases, you can assume that $T(3) \leq 1$. Also, you can assume that n is a power of 3. Give the best possible answer.

$$T(n) \leq 7T(n/3) + n^2$$
$$T(n) \leq 7T(n/3) + n$$
$$T(n) \leq 7T(n/3) + 1$$

Answer:

Here is the definition of the master theorem that will be used to solve the recurrences in this problem. We found this master theorem on the website:
http://cse.unl.edu/ choueiry/S06-235/files/MasterTheorem-Handout.pdf

Theorem (Master Theorem)

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$T(n) \leq 7T(n/3) + n^2$

a = 7, b = 3, d = 2
$7 < 3^2$
$7 < 9$
Case 1 Applies, therefore $T(n) = \theta(n^2)$

$T(n) \leq 7T(n/3) + n$

a = 7, b = 3, d = 1
$7 > 3^1$
$7 > 3$
Case 3 Applies, therefore $T(n) = \theta(n^{\log_3 4})$
However, this can simplify to be $T(n) = \theta(n)$

$T(n) \leq 7T(n/3) + 1$

a = 7, b = 3, d = 0
$7 > 3^0$
$7 > 1$
We see that Case 3 applies again, therefore $T(n) = \theta(n^{\log_3 4})$

Q2 (1 point): Consider the recurrence $T(n) \leq 2T(n/2) + \sqrt{n}$, $T(1) = 1$. We proved in class that this was $O(n)$ using the recursion tree. Prove that using induction.

Answer:

Our guess is $T(n) \leq cn$. We will use induction to try and prove this.

Base Case:
Assume n = 2, $T(\frac{n}{2}) \leq c\frac{n}{2}$
$T(\frac{2}{2}) \leq c\frac{2}{2}$
$T(1) \leq c$✓
Our Base Case is proven

Induction:

$T(n-1) \leq 2T(\frac{n-1}{2}) + \sqrt{n}$

$\leq 2c(n-1) + \sqrt{n}$

$\leq \frac{n(2c-2c)}{n} + \frac{\sqrt{n}}{n}$

$\leq c + \frac{n^{\frac{1}{2}}}{n}$

$\leq cn$

Our proof by Induction is completed

Q3 (2 points): In array A, an inversion is a pair (i, j) such that $i < j$ but $A[i] > A[j]$. Design a divide and conquer algorithm to count the number of inversions in an array. Give pseudocode and perform a time-complexity analysis. (Hint: you might want to modify the code for Mergesort.)

Answer:

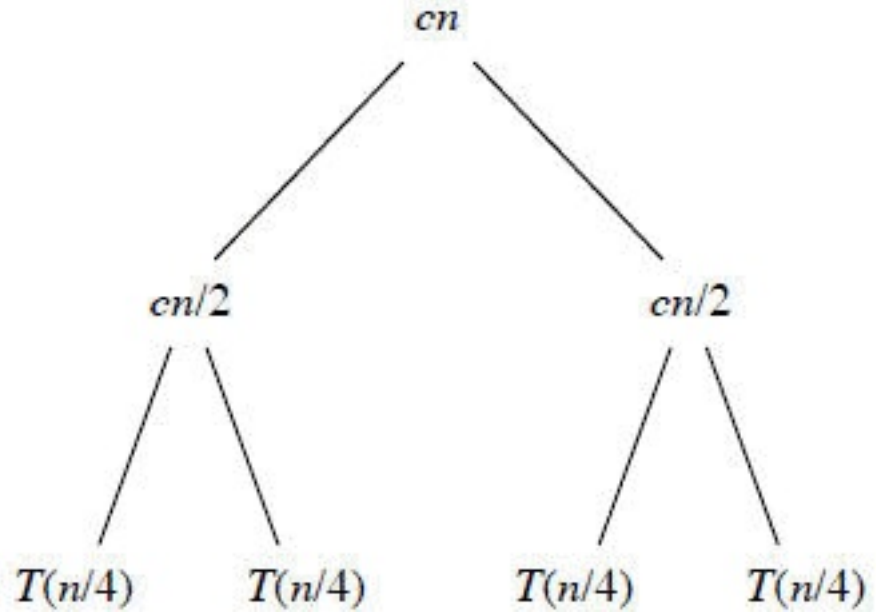
The answer to this question is found when we analyze the "merge" part of mergesort. If we look at this part of the algorithm carefully, we see that when our arrays are merging, the algorithm actually counts the number of inversions during the case of when we have to put an element from the left sorted array, into the final sorted array over the left one. We simply need to tweak the algorithm so we are counting the number of inversions

Psuedo Code:
1.)**USE MERGESORT(A) PSEUDO CODE USED IN LECTURE SLIDE**
2.)ModifiedMerge(B, C)
3.) B[M+1] = C[m+1] = infinity
4.) i = j = k = 1, r = 0 // r is the inversion counter
5.) while B[i] < infinity or C[i] < infinity
6.) if B[i] > C[i] < infinity
7.) if i <j
8.) r = r + 1
9.) D[k] = C[j]
10.) j = j + 1

11.) else
12.) D[k] = B[i]
13.)
14.) j = j + 1
15.) k = k + 1
16.) Return r // return the number of inversions we counted

Time Complexity Analysis:

Since we are making a *very* small change (0(1)) to the original Merge-sort algorithm, our time complexity is still *O(nlogn)*. We will prove this algorithm by using the Master theorem from Q1.This is our recursion tree:



The recurrence relation for this algorithm will be:

$T(n) \leq 2T(n/2) + cn$, where c is some constant and represents the additional run time

a = 2 b = 2 d = 1
$2 = 2^1$
2 = 2

As we can see, Case 2 applies since $a = b^d$, so this means that our time complexity is $T(n) = \theta(n^d \log n)$
.

The time complexity of modified Merge Sort is $T(n) = \theta(n \log n)$

.

Q4 (3 points) Given an array A with n elements, we want to find the k-th smallest element of A. (Formally, this is the value that occurs in position k after sorting.) Suppose we choose the pivot value to be $A[1]$ and partitioned as in class. So we get $A[1 \ldots i-1]$ with elements at most $A[1]$, $A[i]$ has the value of $A[1]$, and $A[i+1 \ldots n]$ has elements at least $A[1]$. There are three cases:

- If $i < k$, then the kth smallest element in A is the kth smallest element in $A[1 \ldots i-1]$.
- If $i = k$, then $A[i]$ is the kth smallest element, and we're done.
- If $i > k$, then the kth smallest element in A is the $(i-k)$th smallest element in $A[i+1 \ldots n]$.

1. Use this observation to design a recursive algorithm for finding the k-th smallest element of A, based on the above idea. You can simply cite any algorithm discussed in class.
2. What is the worst-case time complexity of this algorithm?
3. Now, let us choose the pivot randomly at every step. *Assume all elements are distinct.* Analyze the time-complexity of the new algorithm. (You'll probably get a recurrence that can solved using induction.)

Answer:

1)

In class, we talked about Quicksort (see lecture slide for reference to the pseudo code). As we can see, this algorithm does not completely sort the two sub arrays when it gives us the position of the pivot. All that we know is that the elements to the left of that pivot are less then it, and the ones that are on the right are greater then it. The ordering within those two sub arrays do is not technically sorted. If we simply swap the pivot that we have with the jth element of the input set, we can use that to find the kth element of the array. Within our Partition(A) code, we can tweak it to find what we are looking for. If j is less then a variable k, then we know that we need to look in the right sub array to get k. If j is more then k, then we know that we have to look in the right subset. As we are partitioning our array and placing our pivots, we do not need to keep iterating through the array afterwards, and we can break the loops as soon as we find out that the position of the pivot is = k, which would give us the kth smallest element.

Psuedo Code:
1.)**USE QUICKSORT(A) PSEUDO CODE USED IN LECTURE SLIDE**
2.)ModifiedQS(A, k)k is the desired element
3.) n = A.length
4.) pivot = A[i], i = 1, j = n+1, k 5.) while(true)
6.) do i++ while A[i] < piv
7.) do j– while A[j] > piv
8.) if i ≥ j, break
9.) if piv ==k
10.) return piv
11.) else if piv < k]]
12.) Run partition on the right half // keep running until piv pos matches k
13.) else if piv > k
14.) Run partition on the left half //run till piv pos matches k

2)

Time Complexity Analysis:
Our recurrence relation is the same of that of Quicksort. Since we are dividing our arrays in half, our recurrence relation becomes:

$T(n) \leq 2T(n/2) + T(n/2)$

Using the Master thm above, we can see that:
a = 1, b = 2, d = 0
$1 = 2^0$
1 = 1

As we can see, Case 2 applies since $a = b^d$, so this means that our time complexity is $T(n) = \theta(n^d \log n)$
.
The time complexity of modified quick Sort is $T(n) = \theta(n \log n)$
.
1)

If the pivot is chosen randomly, and all of the elements are distinct, then we will prove that the running time of our Randomized QuickSort, is still $T(n) =$O$(n \log n)$ using a proof by induction.

Base Case:
Assume n = 2, $c = T(2) \leq a * 2\log_2(2) - 2b = 2a - 2bc \leq 2(a - b)$ ✓

Our Base Case is proven

Induction:
Assume $T(s) \leq as * \log_2(s) - bs$ where for all s < n

$T(n) \leq \frac{n}{2}[\sum_{s=1}^{(n-1)} s * \log_2(s) - bs] + cn$

$= \frac{2a}{n}\sum_{s=1}^{(n-1)} s * \log_2 s - \frac{2b}{n}\sum_{s=1}^{(n-1)} s + cn$

$\leq a_n \log_2 n - bn$

Our proof by Induction is completed

Q5 (2 point) extra credit In Q4 above, for the randomized algorithm, we assume that all elements are distinct. Why is this necessary for your analysis? We can resolve this issue choosing the pivot randomly (as before) and by partitioning into three pieces: elements strictly less than the pivot, elements equal to the pivot, and elements at least the pivot. Can you work out the algorithm and give an analysis?

Answer:

It is necessary to assume all of the elements are distinct to avoid the question of what we mean when we say we want to find the k^{th} smallest element in the array. We simply partition the array in three different ways then compared to just two above.

Psuedo Code:
1.)RQuickSort(x, l, r)
3.) i = 1, k = 1, p = r
4.) while(i < p)
5.) if current value is less than r
6.) swap the next two values
7.) else if current value is equal to r
8.) swap the current value with the one that is before r
9.) if piv ==k
10.) else
11.) iterate i
12.) have another function to find the min between r-k an n-p+1 that swaps the two values

Time Complexity Analysis:

Our analysis from Q4 does not same much because we are only increasing the number of comparisons, which increases our time complexity by a constant factor. Therefore, our time complexity is still $T(n) = \theta(n \log n) * c$, where c is a factor. However, this does not matter in the long run so we can just say it is $T(n) = \theta(n \log n)$