Homework #5
Introduction to Algorithms/Algorithms 1
600.463
Spring 2016

Submitted By: Sindhuula Selvaraju
JHED ID:sselvar4
In Collaboratin With: Anchit Pandya, Adhiraj Yadav

March 3, 2016

1 Problem 1 (10 points)

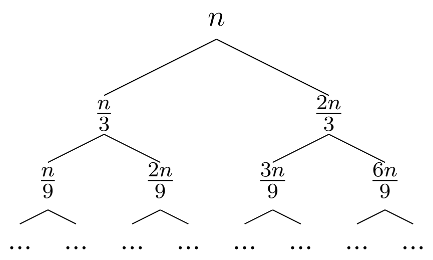
In class we considered the algorithm SELECT, which determines the ith smallest element in the array of size n in $O(n)$ time for the worst case input. The first step of this algorithm is division into $n/5$ groups of 5 elements each. Consider other two versions of this algorithm: the first one uses division into $n/3$ groups of 3 elements each and the second one uses division into $n/7$ groups of 7 elements each. Otherwise both algorithms implement the same routine as SELECT. Which algorithm is asymptotically faster, the one with division into groups of 3, groups of 5 (SELECT) or groups of 7? Prove your statement.

1.1 ANSWER:

(a.) Dividing into $n/7$ groups of 7 elements each Using the proof given in CLRS chapter 9 for groups of 5 elements each.
We have that atleast $\lceil \frac{7}{2} = 4 \rceil$ elements will be greater than the i_{th} smallest element, except for the group containing elements ≤ 7 and the one containing the i_{th} element.
there are $4(\lceil \frac{1}{2} \lceil \frac{n}{7} \rceil \rceil - 2)$ elements greater than the i_{th} element
$\implies 4(\lceil \frac{1}{2} \lceil \frac{n}{7} \rceil \rceil - 2) \geq \frac{2n}{7} - 8$
Similarly atleast $\frac{2n}{7} - 8$ elements are less than the i_{th} element.
So select will be called recursively on $\frac{5n}{7} + 8$ elements in the worst case

So the recurrence becomes:
$T(n) \le T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n)$
Solving by Substitution(Similar to case given in textbook):
$T(n) \le T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n)$
$T(n) \le c\frac{n}{7} + c + c(\frac{5n}{7} + 8) + an$
$T(n) \le c\frac{n}{7} + c + \frac{5cn}{7} + 8c + an$
$T(n) = \frac{6cn}{7} + 9c + an$
$T(n) = cn + (\frac{-cn}{7} + 9c + an)$
This will be atmost cn if $(\frac{-cn}{7} + 9c + an) \le 0$
Similar to the case in CLRS this too will be linear
Running time $T(n)_7 = O(n)$

(b.) Dividing into $n/3$ groups of 3 elements each
As done above the number of elements greater and less than the i_{th} element is atleast:
$2(\lceil \frac{1}{2} \lceil \frac{n}{3} \rceil \rceil - 2) \ge \frac{n}{3} - 4$
In this case the recurrence becomes:
$T(n) \le T(\lceil \frac{n}{3} \rceil) + T(\frac{2n}{3} + 4) + O(n)$
$T(n) \le c\frac{n}{3} + c + c(\frac{2n}{3} + 4) + an$
$T(n) \le c\frac{n}{3} + c + \frac{2cn}{3} + 4c + an$
$T(n) = \frac{cn}{7} + 5c + an$
As this clearly shows that we cannot proceed as in case of $n/7$. let us try an alternate approach.
Discounting the constant terms(as they do not contribute to the asymptotic complexity) this becomes similar to Bob's algorithm in Homework 3 question 1.
$T(n) \le T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n) =$



Each level of the problem takes atmost cn partitioning time. The left child of each node is $\frac{1}{3}$ times the size of the parent and the right child is $\frac{2}{3}$ times the size of the parent. Since the smaller subproblems are on the left, by following a path of left children, we get from the root down to a subproblem size of 1 faster than along any other path. This happens after $\log_3 n$ levels.

Similarly to reach a subproblem of size 1 down the right sub-tree it will take $\log_{\frac{3}{2}} n$ levels.
Since till the left sub-tree reaches a size of 1 there are n nodes the partitioning time for each level is cn. But once the left sub-tree has finished partitioning(i.e. all the elements there are now sorted) the right sub-tree still has some levels to complete but not the time will be atmost cn since the number of elements has reduced and will be O$(n \log_{\frac{3}{2}} n)$.
So the running time of this algorithm is O$(n \lg n)$
$T(n)_3 = O(n \lg n)$
Hence, it is asymptotically faster to run the algorithm that uses division into $n/7$ groups of 7 elements each or of $n/5$ groups of 5 elements as they both have a linear running time as compared to groups of 3 elements.

2 Problem 2 (20 points)

Professor asked Bob to find the median of an integer array A, of size n, which is stored on the lab server. Bob has access to the server, however, his rights are very limited: he can only read data from the server, but cannot write to the server.
The array is so large that Bob can not just copy it to his machine. Bob's computer has only $O(logn)$ memory. Help Bob to develop an efficient algorithm which finds the median of A. Provide a correctness proof and running time analysis. Full score will be given for $O(n \log n)$ expected time algorithm.

2.1 ANSWER:

The algorithm for finding the median of an Array in $O(n \log n)$ time and $O(\log n)$ space:

Algorithm 1: Finding the median of array A.

```
Input  : Unsorted array A of size n
Output: Median of A
findRank $(A[], num)$
{
rank = 1;
for i = 1 to size of A
if A[i] $<$ num then
    rank ++;
else
    continue
end
end for
return rank;
}
findMedian$(A[])$
{
leftlimit = - $\infty$
rightlimit = $\infty$
rank = 0
median = 0
while rank $\neq \frac{n}{2}$ do
    median = random number from array if median $\geq$ leftlimit AND median
      $\leq$ rightlimit then
        rank = findRank(A,median)
    else
        if rank $> n\frac{}{2 \text{ then}}$
            rightlimit = median
        else
            leftlimit = median
        end
        end
    end
    return median; }
```

2.2 Correctness Proof:

We can prove the correctness of this algorithm step by step.
The findRank function iterates over the array A on the server and counts the number of elements less than the element passed to it i.e. it returns the index the element would have been had the array been sorted.
Generating the random number is similar to the random generator used in randomized quicksort.
The findMedian function is the one that actually calculates the median.
This function always checks if the produced randomized value is equal to the rank of the median element($\frac{n}{2}$ or $\frac{n}{2}+1$)
If not it checks if the rank of the element is lesser or greater than the rank of the median.
If it is greater the element is compared against the rightlimit(upper bound) and if lesser than the rightlimit, it is initialized as the new rightlimit.
A similar procedure is performed with the leftlimit(lowerbound).
The distance between the rightlimit and leftlimit gradually decreases and as soon as the median value is selected it is returned.
Thus, we can see that our algorithm is sound and complete.

2.3 Running Time Analysis:

The findRank algorithm is linear in n as it iterates over the array and compares each element with the number passed to it.
Generating a random number takes constant time.
In the Median Function:
The if block is only executed if the random number generated is between the leftlimit and rightlimit. Thus even if unbalanced pivots(random numbers) are found the leftlimit anf rightlimit keeps constricting, corresponding to a finite constant split.
Since, we are utilizing the same principle as randomized quicksort, where constant split is created based on the pivot element, we are dealing with the same recurrence relation here.
Just like in randomized quick sort, we can see that generating a random number and successively reducing the search space, the if block will only execute $\log n$ amount of times, (like in the case of randomized quick sort the depth of the recursive tree generates was $\log n$).
Since the if block calls findRank(), log n number of times we know that the time complexity of the median function is $O(n \log n)$

This can also be seen in CLRS(chapter 7) within the balanced partitioning section of quicksort that any split of constant proportionality results in a recursion tree of depth $\theta(\log n)$. Since, the time complexity associated at each step is $O(n)$ we can see that the total time complexity will be $O(n \log n)$

2.4 Space Complexity Analysis:

As we are not modifying the original array. We only need space to store the current values of leftlimit, rightlimit , median and other minor variables which will take some constant space. Hence we will not be using more than O($\log n$) space

3 Problem 3 (20 points)

Bob is a presidential candidate. He is planning to visit every city in the state M to give a talk. Bob's budget is very tight so he can buy only two flight tickets. Bob can land in any city A in the state M and he can depart from any city B in the state M, but inside the state he can only commute by car. Between any two cities in the state M there is exactly one highway (the network of roads can be represented by a complete graph). Because of Bob's tight budget, Bob cannot stop in any city more than once. To make things more complicated for Bob, each highway is a one-way highway. Prove that there exists cities A and B and a path from A to B, such that Bob will visit every city exactly once.

Note: you need to prove existence of such A, B and a path from A to B for any for any given configuration of the highway's one way directions.

A complete graph is a simple graph with an edge between any two vertices.

3.1 ANSWER:

Imagine each city in M is represented as a node on a complete graph G of the highway networks where each highway forms the edge.
We have to find a path from A to B such that we can visit each city in M exactly once when going from A to B.
NOTE: Henceforth we are assuming A to be the start point and B to be the end point
We can prove that such a path exists by induction:

Base Case: Let there be only 2 cities X and Y in M. If the highway goes from X to Y then $X = A$ and $Y = B$ else $X = B$ and $Y = A$

Induction Hypothesis: Assume there are n cities$(C_1, C_2...C_n)$ in M such that there exists a path to cover all the the cities and $A \in (C_1, C_2...C_n)$ and $B \in (C_1, C_2...C_n)$ and $A \neq B$

Induction Step: Let there be $n + 1$ cities$(C_1, C_2...C_{n+1})$ in M.
From the Induction Hypothesis there will be a path to cover all the the cities and $A \in (C_1, C_2...C_n)$ and $B \in (C_1, C_2...C_n)$ and $A \neq B$
There can now be the following possible cases for city $(Cn + 1)$:

1. If there is a path from B to C_{n+1} then $B = C_{n+1}$
2. If there is a path from C_{n+1} to B then recalculate A and B:
    (a) If there is a path from C_{n+1} to A then $A = C_{n+1}$
    (b) If there is a path from A to C_{n+1} and C_{n+1} to next stop from A then C_{n+1} becomes the next stop from A.
    (c) If there is a path from any intermediate stop D to C_{n+1} and a path from C_{n+1} to stop after D i.e. D_2, Bob will travel from D to C_{n+1} to D_2.

Hence, we can always find a path through all cities of M by varying the start and end points.