Bryan Arnold

CSE 3500

3/20/17

Homework 6

Divide and Conquer

First, split array A into 2 subarrays A1 and A2 of half the size. Next, choose the majority element of A1 and A2. After that, do a linear time equality operation to decide whether it is possible to find a majority element:

Algorithm:

Input: Array A of objects, with size being 1…n

if n = 1: return A[1]

Initialized an integer k to split the array into two subarrays: k = n/2

Now split the array into two subarrays by recalling the method for each side of the array for each element:

leftArray = majorityElement(A[1...k])

rightArray = majorityElement(A[k+1...n]

if leftArray = rightArray: return leftArray //if the elements are the same, just return one of them

Next, use another method that counts the number of times an element occurs in a given array (elementCount). Use this method for each subarray:

leftTotal = elementCount(A[1...n], leftArray)

rightTotal = elementCount(A[1...n], rightArray)

Finally, return the majority element of the array if the total occurrences of an element exceeds half the size of the initial array, or return there is no majority element if neither subarray has occurrences higher than half the size of the initial array:

if leftTotal > k+1: return leftArray

else if rightArray > k+1: return rightArray

else return: a message saying there is no majority element in the given array.

elementCount computes the number of times an element appears in the given array A. Two calls to elementCount is O(n), since you only need to go through each subarray once to count the total occurrences of an element. The initial separation through recurrence of the array would make it run in logn time, since the problem is being split in half. So, the total runtime of this algorithm would be the splitting of the array time times the elementCount time, so O(nlogn). This

algorithm is correct because a majority element is considered a majority if it occurs over half the time within an array. By splitting the array up, you can go through both parts to see if an element occurs more than half the initial array size, and simply display the element if it does.

Matrix Multiplication

a) The needed multiplications for computing A x A when n = 2 are as follows:

$$A \times A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} (a \times a) + (b \times c) & b \times (a + d) \\ c \times (a + d) & (b \times c) + (d \times d) \end{bmatrix}$$

So, to do this matrix multiplication, the following 5 multiplications are necessary: (a x a), (d x d), c x (a + d), (b x c), and b x (a + d).

b) Tom is incorrect in his approach because he does not realize that matrix multiplication is not communitive. For example, take the matrix above for squaring A:

$$A \times A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2 + (b \times c) & b \times (a + d) \\ c \times (a + d) & (b \times c) + d^2 \end{bmatrix}$$

Now, take a new definition for A to be (for a n x n matrix):

$$A \times A = \begin{bmatrix} P & Q \\ R & S \end{bmatrix} \times \begin{bmatrix} P & Q \\ R & S \end{bmatrix} = \begin{bmatrix} P^2 + QR & PQ + QS \\ RP + SR & RQ + S^2 \end{bmatrix}$$

As you can see, using this definition for A^2 we cannot achieve 5 multiplications for the scalars as multiplication is not commutative. This requires 7 multiplications. Most of these arithmetic subproblems do not all involve squaring, but a mix of squaring and other operations. Therefore, the original squaring of A is different from the nature of the subproblems.

c) i) The O(n^2) portion of the runtime for AB + BA can be seen in how many multiplication operations are needed for the multiplications of AB and BA. For this problem, it would be 16 multiplications for when n = 4, or the number of elements in each matrix. So, n x n + n x n or both AB and BA would total out to be O(n^2). The 3S(n) portion of the runtime comes from the addition found in the subproblems based on the number of matrices. So, there are two matrices, so n = 2, O(n^c) when c >= 2 would mean 4, then multiplied by three to get 12, which was the total number of additions needed for this problem. So, by simply adding the runtimes up for all the subproblems for this matrix problem, we can see 3S(n) + O(n^2) makes sense for AB + BA.

ii)

$$AB = \begin{bmatrix} X*0+0*0 & XY+0*0 \\ 0*0+0*0 & 0*Y+0*0 \end{bmatrix} = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix}$$

$$BA = \begin{bmatrix} 0*X+Y*0 & 0*0+Y*0 \\ 0*X+0*0 & 0*0+0*0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$AB + BA = \begin{bmatrix} 0+0 & XY+0 \\ 0+0 & 0+0 \end{bmatrix} = \begin{bmatrix} 0 & XY \\ 0 & 0 \end{bmatrix} (= AB)$$

iii) XY is two 2n by 2n matrices, meaning their dimensions replace that of the given 3S(n) to be 3S(2n), both being 2n x 2n. This is simply simplified to 3S(n) using common sense rules, as well as O(2n^2) turning into O(n^2). So, the runtime is still 3S(2n), but can be simplified down to 3S(n) to reinforce that the runtime is really O(n^c) when c >= 2. As it can be seen, part a is reinforced by part b, showing all the subproblems needed with all the runtimes of each subproblem being O(1), but they must be added up for true runtime. All of the runtimes and number of computations lined up correctly as well. Since S(n) = O(n^c) when c >=2, and the latter half of the runtime is O(n^2), they are virtually the same thing. Finally, they can be simplified using common sense rules to go out to O(n^c) as stated earlier while c >= 2.

Exercise 5.1

For this algorithm, we need two query pointers for the databases. Denote these pointers as p and p1. Now, query the medians of the databases to obtain the medians of both databases. Denote the medians as m and m1. The median of the joint database of the two databases must be in between the two medians . There are at least n records in DB and DB1 which are smaller than or equal to max(m, m1). So, it makes sense that the median of the joint database is not greater than max(m, m1). It can also be shown the median of the joint database is not smaller than min(m, m1). Therefore, the pointers p and p1 can be moved accordingly. Here is the following algorithm:

Initialize p and p1 to their pointer values: p = p1 = n/2

for i = 2…logn

m = QueryDB(DB1, p1) //get the median of DB

m1 = QueryDB(DB2, p2) //get the median of DB1

Next, if the median of DB is bigger than DB1, query the median to the upper half of DB and the lower half of DB1:

if m > m1 then

p1 = p1 − n/2^i

p2 = p2 + n/2^i

Otherwise, do the opposite:

else

p1 = p1 + n/2^i

p2 = p2 − n/2^i

end if else chain

end for i

Finally, return the minimum of the two medians: return min (m, m1)

End algorithm

By the end of the for loop, m and m1 are the nth and the (n + 1)th smallest numbers of the joint database, hence we return the smaller one among m and m1, making he algorithm valid. T(n) represents the total number of queries. Each instance of divide, the problem size is reduced by half using two queries. Therefore, T(n) = T(n/2) + 2. Solving this recurrence, we obtain T(n) = O(logn) using the master theorem.

Subset of Lists

For this algorithm, a few things must be noted beforehand. First, in the standard linear-selection algorithm, you select a random pivot, or you can use the algorithm to find the optimal pivot as well, and see how many numbers fall on each side of it, the left and right side of the collection of numbers in L. Now, run the linear-time selection algorithm. Then, either accept or reject one half while still working on the other half. Each number has now been looked at in each half. Thus, the cost of each pivot stage is linear, since you must go through each half's contents completely, but the amount of numbers being managed at each stage reduces fast enough that the total cost is still only linear. The cost of a pivot stage will still be only linear if you take the sum of all the numbers above the pivot.

Using this, you can work out if accepting all these numbers, together with any numbers previously selected, would give you a collection of numbers that add up to at least C. So, if the numbers after the pivot add up to be at least C, you can disregard the other numbers below the pivot and use the numbers above the pivot for the next pass. Likewise, if the numbers after the pivot do not add up to at least C, you can accept all the numbers above the pivot, and use the numbers below the pivot for the next pass. As with the selection algorithm, the pivot itself and any ties give you a few special cases and the possibility of finding an exact answer early.

Essentially what is being done is, you look above the pivot to see if the numbers added up are at least C, and if they are you use only those numbers as another pass that again looks if they add up to at least C. This is the divide portion of the algorithm. You would continue to divide all the way down until 1 number in the accepted half you are looking at, and back track by conquering by seeing if the numbers added together are at least above C. So, eventually you will end up with the least number of integers needed to get at least C, and you would report this and exit the program when it happens. If the numbers above the pivot are not at least C when added together, you disregard them and then do the same divide and conquer method as described

above by recurring the numbers below the pivot in the same manner as described above. Display no amount of integers equal at least C if no solution is found.