Eddie Christopher Fox

April 17, 2017

CS 325 Analysis of Algorithms

Assignment 2

Problem 1: *(3 points)* Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers. *Assume the base cases T(0)=1 and/or T(1) = 1.*

A) $T(n) = T(n-2) + n$

Solved with muster. T(n) = a T(n-b) + f(n)

A = 1

B = 2

F(n) = N

D = 1

If A = 1, then T(n) is theta $N^{d+1} = N^{1+1} = N^2$

T(n) = Theta(N^2) N squared.

B) $T(n) = 3T(n-1)+1$

Using muster method (answer at beginning of next page)

T(n) = a T(n-b) + f(n)

A = 3

B = 1

F (n) = 1

D = 0

If A > 1, then T(n) = Theta ($n^d * a^{n/b}$) = $1 * 3^{n/1} = 3^n$

T(n) = Theta (3^n) 3 to the nth power.

C) $T(n)=2T(n / 8)+4n2$

Using muster method.

T(n) = a T(n / b) + f(n)

A = 2

B = 8

F(n) = $4n^2$

$N^{\log}{}_b\ a= N^{\log}{}_8\ 2 = N^{1/3}$

F(n), being N^2 is more complex $N^{1/3}$, so it dominates. Thus, this is Case 3 of the master method.

T(n) = Theta (N^2) N squared .

Problem 2: *(4 points)* Consider the following algorithm for sorting.

STOOGESORT(A[0 ... n - 1])

if n = 2 and A[0] > A[1]

swap A[0] and A[1]

else if n > 2

k = ceiling(2n/3)

STOOGESORT(A[0 ... k - 1])

STOOGESORT(A[n - k ... n - 1])

STOOGESORT(A[0 ... k - 1])

Continued on next page.

a) Explain why the STOOGESORT algorithm sorts its input. (This is not a formal proof).

Trivially, the algorithm works for n = 0, 1, and 2. This is the base case. If there is 0 elements, there is nothing to sort. If there is 1 element, there only element is already sorted, as there is no other possible configuration. If there is 2 elements, the algorithm will swap those elements if the first element is greater than the last, and this action will sort two elements properly.

This algorithm recursively uses divide and conquer to sort the array. We can divide the array up into 3 thirds. A is the first third of the array. B is the second third of the array. C is the third third of the array. The first recursive function sorts the first two thirds of the array. A and B are recursively sorted and all the elements in B, the second third of the array, will be larger than the ones in A. The second recursion sorts B and C, moving the largest values in B to C. This sorts the final two thirds of the array. Finally, the first two thirds are sorted again. A and B. This will move any of the smaller values from C that were exchanged with B in the previous function to A.

b) Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)

If ceiling was replaced with floor, the algorithm wouldn't work. Imagine n = 4. Originally, ceiling(2n / 3) = ceiling (8/3) = 3. The floor of (8/3) is 2. This value is important because K is assigned its value.

If there are 4 elements, the array will go up to index 3, because it counts from 0. But if K is given a value of 2 instead of 3, then the arguments for the recursive sorting will be wrong. Going from 0 to K-1 for example will only sort indexes 0 and 1 for the initial two thirds instead of indexes 0, 1, and 2. If the algorithm isn't properly creating the partitions of the array, it will fail to combine after dividing, or give inaccurate solutions.

c) State a recurrence for the number of comparisons executed by STOOGESORT.

$T(n) = 3\ T(2/3n) + 1$. We can derive this because 1 is the constant time operation at the start. (2/3n) is because the recursive function is performed on an array 2/3rds the size of the original, and the 3 is because the recursive function is called 3 times.

d) Solve the recurrence to determine the asymptotic running time..

If N is = 0, 1 or 2, the algorithm runs in constant time.

For everything else, we can apply the master method to $T(n) = 3\ T(2/3n) + 1$

$A = 3$

$B = 3/2$ (Because n divided by 3/2 would give 2/3's N)

$F(n) = 1$

D = 0

Computing $N^{\log_b a} = N^{\log_{3/2} 3} =$ Applying change of base formula to get log 3 / log 1.5 = 2.7ish

We have case 3 where the weight of the leaves dominate because $n^{\sim 2.7}$ is much greater than n^0.

T(n) = Theta ($n^{(\log 3 / \log 1.5)}$) Approximately n to the 2.71 power.

Problem 3: *(6 points)* The quaternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-fourth.

a) Verbally describe and write pseudo-code for the quaternary search algorithm.

An algorithm that can locate an item within an already sorted array by recursively dividing the array into four equal or almost equal parts.

Let L = the smallest element in the array, M = the middle element of the array, H = the largest element of the array, Q1 = the element halfway between L and H, and Q3 = the element halfway between M and H.

quarterSearch(array, target, low, high)

Middle = (Low + High) / 2
Q1 = (Low + Middle) / 2
Q3 = (Middle + High) / 2

If target is equal to low, high, middle, Q1, or Q3, return the index.
Else If: the target is less than Q1, apply quarterSearch using L as low and Q1 as high.
Else If: the target is less than middle, apply quarterSearch using Q1 as low and middle as high.
Else If: the target is less than Q3, apply quarterSearch using middle as low and Q3 as high.
Else If: the target is greater than Q3, apply quarterSearch using Q3 as low and high as high.
Else: Return failure to find target.

b) Give the recurrence for the quaternary search algorithm

T(n) = T(N / 4) + 1. Recursive function called once, plus the constant time to return element once found.

c) Solve the recurrence to determine the asymptotic running time of the algorithm.

Using master theorem:

A = 1
B = 4
F(n) = 1
D = 0

$N^{\log_4 1} = N^0 * \text{Lg } n = 1 * \text{Lg } n = \text{Lg } n.$

Asymptotic running time: **T = O (log n)**, due to base 4, it would be **T = Theta (log 4 n)**

d) Compare the worst-case running time of the quaternary search algorithm to that of the binary search algorithm.

Binary search is theta (log 2 n) while Quartneary search is theta (log 4 n).

While they are both log n, **binary search running time is a bit less.** Anything beyond binary involves more comparisons at each level, slightly increasing the runtime of the algorithm

Problem 4: *(6 points)* Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

a) Verbally describe and write pseudo-code for the min_and_max algorithm.

findMinMax(array, sizeOfArray, low, high)

minMax is a structure that can contain both a min and max variable.

If sizeOfArray = 1: Set element as both min and max in minMax structure and return structure.
If sizeOfArray = 2: compare the two numbers. Set the greater of them as the max and the lesser of them as the min in the minMax structure, then return structure.

Else if sizeOfArray > 2:

middle = floor (sizeOfArray / 2)

findMinMax(array, (middle for new size), low index, middle index) // call recursively on left half

findMinMax(array, (sizeOfArray – middle to calculate new size) , middle index, high index) // call recursively on right half

// The size calculation formulas might be a bit off.

// These two comparisons will recursively combine the min and max solutions from the recursive calls.

Compare the min found by the left and right recursive calls and set the smaller number as trueMin.

Compare the max found by the left and right recursive calls and set the smaller number as trueMax.

Set min and max in the structure that holds both min and max variables to the values of trueMin and trueMax.

Return the structure.

b) Give the recurrence.

T(n) = 2 T(n / 2) +2 . 2 because we call the function recursively twice during running. N/2 because each call divides the problem by 2. +2 for the constant steps when N = 1 and N = 2.

c) Solve the recurrence to determine the asymptotic running time of the algorithm.

Using master theorem:

A = 2

B = 2

F(n) = 2

D = 0

$N^{\log_b a} = N^{\log_2 2} = N^1 = N$. The weight of the leaves dominates the constant f(n).

T(n) = Theta(n)

d) Compare the running time of the recursive min_and_max algorithm to that of an iterative algorithm for finding the minimum and maximum values of an array.

Both iterative and recursive algorithms are O(n), so I would imagine the running times would be pretty similar.

Problem 5: *(6 points)* An array A[1 . . . n] is said to have a majority element if more than half of its entries are the same. The majority element of A is any element occurring in more than n/2 positions (so if n = 6 or n = 7, the majority element will occur in at least 4 positions). Given an array, the task is to design an algorithm to determine whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from an ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (Think of the array elements as GIF files, say.) Therefore you cannot sort the array. However you can answer questions of the form: "does A[i] = A[j]?" in constant time.

a) Give a detailed verbal description for a **divide and conquer** algorithm to find a majority element in A (or determine that no majority element exists).

This algorithm recursively divides the array and attempts to find the majority element in each sub array. A second function called frequency counts the number of elements for each candidate majority and compares it to the ceiling of the number of elements in the array divided by 2.

b) Give pseudocode for the algorithm described in part a)

N = # of elements in array
findMajority(array[1 ….. N])

// If N =1, there is only 1 element in the array, so you just return the element.

If n = 1: return index 0 of array.

middle = ceiling(N / 2)

// Returns the element that has the majority in each recursively divided array if applicable.
leftElement = findMajority(array[1…. middle])
rightElement = findMajority(array[middle+1 ….. n])

// We now introduce a function called frequency that calculates the number of times a given element is shown in O(n) time.

frequency(array, element to calculate the number of)
leftCount = frequency(array, leftElement)
rightCount = frequency(array, rightElement)

if leftCount > middle + 1: return leftElement
else if rightCount > middle + 1: return rightElement

else: return No majority

c) Give a recurrence for the algorithm

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

d) Solve the recurrence to determine the asymptotic running time.

T(n) = O(n log n)

Note: A O(n) algorithm exists but your algorithm only needs to be O(nlgn).