Algorithms HW1

Greyson Brothers

February 7, 2023

1

Statement of Integrity: I, Greyson Brothers, attempted to answer each question honestly and to the best of my abilities. I cited any and all help that I received in completing this assignment.

1.1 Linear Search

This algorithm steps through each item in the array iteratively and compares the search key on each step. In the best case, the item being searched for is at index 1 in the array, meaning the search takes only 1 iteration. Worst case, the item being searched for isn't in the array which would take n iterations. Average case is n/2 iterations. Worst case implies time complexity of $O(n)$

1.2 Binary Search

Binary search assumes array A is already sorted. Each iteration discards half of the search space, implying a worst case log2(n) comparisons when A does not contain the search key x. Best case, we still have log2(n) comparisons, as there is no early stopping condition if A[i] or A[j] is x. This also makes performance invariant to the location of x in A, unlike linear search. Binary search has time complexity of $O(logn)$

2 $T(n) = 3T(\frac{n}{2}) + nlgn$

Using the Master Method as explained in this YouTube video (clickable)

$$a = 3, b = 2, k = 1, p = 1$$

$$log_2(3) = 1.58496 > 1 = k$$

This implies that the first term dominates as n approaches infinity, so by the Master Method, our time complexity is strictly bounded by $\Theta(n^{log_2(3)})$

3 $T(n) = T(\sqrt{n}) + 1$

Unraveling the recurrence relation, we have:

$$T(n) = T(\sqrt{n}) + 1$$

$$= T(n^{\frac{1}{2}}) + 1$$

$$= T(n^{\frac{1}{4}}) + 2$$

$$= T(n^{\frac{1}{8}}) + 3$$

$$= T(n^{\frac{1}{16}}) + 4$$

$$\ldots$$

$$T(n) = T(n^{\frac{1}{2^k}}) + k$$

Where the above equation represents the fully unrolled k-deep recurrence. This implies then that

$$n^{\frac{1}{2^k}} = 0$$

Assuming that our function operates on sets of size n, $\sqrt{n}$ must be an integer. So assuming $n = 2^i$, $i \in N$ we can then make a substitution to get the following:

$$2^{i \cdot \frac{1}{2^k}} = 0$$

$$log_2(2^{i \cdot \frac{1}{2^k}}) = log_2(0)$$

$$\frac{i}{2^k} = 1$$

$$k = log_2 i \Rightarrow k = log_2(log_2 n)$$

Given the $T(0)$ is some constant time operation, we have see that each increment in n results in time increment of k or $log_2(log_2 n)$, so it holds that the time complexity of this function is:

$$T(n) = T(n^{\frac{1}{2^k}}) + k = T(0) + k = c + log_2(log_2 n)$$

so our time complexity is

$$\Theta(log_2(log_2 n)$$

4 $T(n) = 2T(\frac{n}{3} + 1) + n$

Using a recursion tree we get

n

$\frac{n}{3} + 1$, $\frac{n}{3} + 1$

$\frac{n}{3^2} + 1$, $\frac{n}{3^2} + 1$, $\frac{n}{3^2} + 1$, $\frac{n}{3^2} + 1$

$\dots$

$\frac{n}{3^k} + 1$,$\dots$, $\frac{n}{3^k} + 1$

So we have k steps with $n \cdot (\frac{2}{3}^k + 2^k)$ operations per step

I was not able to derive a good guess from this tree, perhaps I went wrong somewhere. I've been looking at Master's Theorem videos (here and here), and have tried to reconcile the two of them to come up with a joint case that applies to the above. I haven't had any luck and have run out of time. If I was able to continue solving this, I would take the approach presented in office hours of using substitution to find upper and lower bounds, then applying Theorem 3.1.

5 Collaborative Problem

5.1 Prove that insertion sort can sort the $\frac{n}{k}$ sublists, each of length k, in $\Theta(nk)$ worst-case time:

Insertion sort is $O(n^2)$. If we split our input set into size k sublists for insertion sort, then it follows that each sublist takes $O(k^2)$. For $\frac{n}{k}$ sublists, we have $O(\frac{n}{k} \cdot k^2) = O(nk)$

Note: I read the group discussion posts prior to writing this up, so this answer was informed by the posts there.

5.2 Prove how to merge the sublists in $\Theta(nlg(\frac{n}{k}))$ worst-case time.

We know that the merge operation takes O(n), so we want to figure out how many merges it takes to zip up all of our sublists. Since we start off with n/k sublists, after the first merge we end up with $n/2k$ sublists. After the second, $n/4k$, then $n/8k$. Each level reduces the number of sublists by a factor of two, so it follows that the total number of merges is going to be $log_2(\frac{n}{k})$. Thus the worst case merge complexity is $O(nlog_2(\frac{n}{k}))$

5.3 What is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of -notation?

We know that standard merge sort has a time complexity of $O(nlgn)$. We want to find the largest k such that $\Theta(nk + nlg(n/k))$ is bounded by $\Theta(nlgn)$. So, given

$$\Theta(nk + nlg(n/k)) = \Theta(nlogn)$$

I make the observation that this equation can only hold when $k \leq lgn$. For clarity, substituting that value gives us:

$$\Theta(nlgn + nlg(\frac{n}{lgn})) = \Theta(nlogn)$$

Thus the largest value of k for which this algorithm has the same complexity of merge sort is lgn

5.4 How should we choose k in practice?

I would suggest first finding your expected range of n, then given a constant value of n reframing the relation as an optimization problem. Here we would set $f(k) = nk + nlg(n/k)$ subject to the constraint $k <$ the upper bound found in 5.3 above. Then solve for the value of k that minimizes $f(k)$