CS 6150: HW1 - Data structures, recurrences

Payman Behnam (u1078370)

Submission date: Friday, Sep 6, 2019 (11:59 PM)

This assignment has 6 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Basics	7	
Bubble sort and binary search	8	
Tries and balanced binary trees	5	
Recurrences, recurrences	17	
Computing intersections	5	
Dynamic arrays: is doubling important?	8	
Total:	50	

Question 1: Basics .... **[7]**

In (b)-(d) below, answer YES/NO, along with a line or so of reasoning. Simply answering YES/NO will fetch only half the points.

(a) [**1**] Sign up for the course on Piazza!

Answer: I did it!

(b) [**2**] Let $f(n)$ be a function of integer parameter n, and suppose that $f(n) \in O(n^2)$. Is it true that $f(n)$ is also $O(n^3)$?

Answer: Yes. It is true. The notation O put an n asymptotic upper bound for the complexity of an function that may or may not be asymptotically tight. since Since $n^2 \leq n^3 when : n \geq 1$, the n^3 is another upper bound. So $f(n) \in O(n^3)$).

Formal proof: $f(n) = O(g(n)) \Leftrightarrow \exists c, n0 \geq 0 : \forall n \geq n0; 0 \leq f(n) \leq cg(n)$. In this example $g(n) = n^2$. Since $n^2 \leq n^3 when : n \geq 1$ considering $k(n) = n^3$ we have: $\exists c, n0 \geq 0 : \forall n \geq n0; 0 \leq f(n) \leq ck(n) \Leftrightarrow f(n) = O(k(n))$ and hence $f(n) = O(n^3)$

(c) [**2**] Suppose $f(n) = \Omega(n^2)$. Is it true that $f(n) \in o(n^3)$?

Answer: No. This is not correct. Ω put a n asymptotic lower bound on a function. *o-notation defines an upper bound that is not asymptotically tight for sure.*

$f(n) = o(g(n)) \Leftrightarrow \forall c, \exists n0 \geq 0 : \forall n \geq n0; 0 \leq f(n) \leq cg(n)$.

However there is no relation between upper bound and lower bound. For example in the function $f(n) = n^3 + n^2 + 30$, we have $f(n) = \Omega(n^2)$, but the statement $f(n) \in o(n^3)$ is not correct. the minimum degree of n in o-notation in this example is 4.

(d) [2] Let $f(n) = n^{\log n}$. Is $f(n)$ in $o(2^n)$?

***Answer**: Yes. This is correct. o-notation defines an upper bound that is not asymptotically tight for sure. $f(n) = o(g(n)) \Leftrightarrow \forall c, \exists n0 \geq 0 : \forall n \geq n0; 0 \leq f(n) \leq cg(n)$. In another word, we can have $f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*

$f(n) = n^{\log n}$ and $g(n) = 2^n$. We have $\lim_{n\to\infty} \frac{n^{\log n}}{2^n} = \frac{\infty}{\infty}$. If we apply log to both numerator and denominator we have $\frac{\log n \times \log n}{n \log 2}$ Since $n \geq (\log n)^2$, (plotted already) the results would be zero . $\lim_{n\to\infty} \frac{(\log n)^2}{n \log 2} = 0$. Hence, the statement is correct.

Question 2: Bubble sort and binary search .... **[8]**

We will re-visit two simple algorithms we saw in class.

(a) [**4**] Recall the bubble sort procedure (see lecture notes): while the input array $A[]$ is not sorted, go over the array from left to right, swapping i and $(i + 1)$ if they are out of order. Given a parameter $1 < k < n$, give an input $A[]$ for which the procedure takes time $\Theta(nk)$. (Recall that to prove a $\Theta(\cdot)$ bound, you need to show upper and lower bounds.)

Answer: In Bubble sort we have two nested for-loops. Worst case happens when the array is sorted in reversed order. In this case, the complexity would be O(n^2). Best case happens when the array is sorted (i.e.O(n)). The question ask the complexity of $\Theta(nk)$ which is between worst case and best case. In this situation, the second loop should iterate K times. In other words, k items is not sorted and (n-k) is sorted.

A = $[a_0, a_1, a_2, ..., a_k, a_{k+1}, ..., a_n]$

Specific Example: A= [1 4 2 5 3 7 6 ...... 8 9 10] : N=10, k =6

The upper bound would be k=n-1 and hence 1 items is sorted and $nk = n^2 - 1$; so the complexity would be O(n^2). The lower bound would be k=2 and hence two items are non-sorted and $nk = 2n$; so the complexity would be O(n). $(1 < k < n)$

Formal Proof should goes through the following formula; however, based on a discussion with a TA the above explanation should be enough.

$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n0 \geq 0 : \forall n \geq n0; 0 \leq c1g(n) \leq f(n) \leq c2g(n)$.

(b) **[4]** Consider the binary search procedure for finding an element x in a sorted array $A[0, 1, \ldots, n-1]$. We saw that the total number of comparisons made is exactly $\lceil \log_2 n \rceil$. Suppose we wish to do better. One idea is to see if we can recurse on an array of size $< n/2$. To this end, consider an algorithm that (a) compares the query x with $A[n/3]$ and $A[2n/3]$, and (b) recurses into the appropriate sub-array of size $n/3$. How many comparisons are done by this algorithm? (You should write down a recurrence, compute a closed form solution, and comment if it is better than $\log_2 n$.)

Answer: Let's write the pseudo code for that.

```
NewSearch(int arra, int l, int r, int x)
if $r \geq 1$ then
    middle1 = l + (r - l)/3 = $\frac{n}{3}$
    middle2 = middle1 + (r - l)/3 = $\frac{2n}{3}$
    if $array[middle1] == x$ then
        middle1
    end
    if $array[middle2] == x$ then
        middle2
    end
    if $array[middle1] > x$ then
        NEWSearch(arr, l, middle1-1, x)
    end
    if $array[midddle2] < x$ then
        NEWSearch(array, middle2+1, r, x)
    end
    return NewSearch(array, middle1+1, middle2-1, x)
else
    return -1
end
```

Algorithm 1: NewSearch

Let's write a recursive formula for counting comparisons for the binary search and the Newsearch method. In Binary search we have:

$T(n) = T(n/2) + 2, T(1) = 1$

In NEWSearch, since wh have four if-then statement we have: $T(n) = T(n/3) + 4, T(1) = 1$ Following the same method for binary search for computing complexity and Master theorem (see details of the theorem in Q4-C), in NewSearch, the number of searches is $4c \log_3 n+$ c in worst case. This is $2c \log_2 n + c$ in the case of binary search.

$\frac{4c \log_3 n}{2c \log_2 n} = \frac{2 \log_3 n}{\log_2 n} = \frac{2 \log_2 n}{Log_2 3 \times \log_2 n} = \frac{2}{Log_2 3} = \frac{2}{0.47} \gg 1$. Hence, NewSearch needs more comparisons than Binary Search in the worst case.

Question 3: Tries and balanced binary trees .... **[5]**

I mentioned in class that prefix trees can be used as a "poor man's binary search tree". Let us see the sense in which this is true. Suppose we have a set of N integers all in the range $[1, N^3]$. If we treat them as binary strings (using the natural binary representation of the integers), how long does it take to (a) add a new integer in the same range to the data structure? (b) to query if some x belongs to the set?

Answer: We can form a tries in such a way that each number is represented using 0,1. Since the maximum value is N^3 the depth of the tree would be $\log N^3 = 3logN$. Since we have N numbers, building the data structure takes O(N.3logN) = O(NlogN) and for the query the complexity would be O(3log N) = O(log N) . Adding one element takes O(logN) .

(Bonus): in what sense is a "regular" binary search tree better?

Answer: In Binary search tree the height would be longer since each number is considered separately whereas in "prefix tree" some representations is shared between different numbers. So, the running time for binary search tree would be worse. However, binary search tree has some good advantages. The left(right) subtree of a node includes nodes with keys lesser(greater) than the node's key. So, it keeps the order of number. Hence, Addition, deletion and search in a self-balanced binary search tree (e.g., Red-Black Tree) is O(L Log n) where L is length of word and n is total number of words. Accordingly, operations like find min, max, and k-th smallest (largest) is faster. Tries also need more memory for storing the strings.

Question 4: Recurrences, recurrences .... **[17]**

Solve each of the recurrences below, and give the best $O(\cdot)$ bound you can for each of them. You can use any theorem you want (Master theorem, Akra-Bazzi, etc.), but please state the theorem in the form you use it. Also, when we write $n/2$, $n/3$, etc. we mean the floor (closest integer less than the number) of the corresponding quantity.

(a) **[5]** $T(n) = 2T(n/2) + T(n/3) + n$. As the base case, suppose $T(0) = 0$ and $T(1) = 1$.

Answer:

If we try Akra-Bazzi theorem we have: $2(1/2)^p + (1/3)^p = 1$ if p=1 then we have 2(1/2)+(1/3) >1; if p=2 then we have 2(1/4)+(1/9) <1. So 1 < p < 2. I plotted the graph and got p= 1.365.

$T(x) = \theta(x^p(1+\int_1^x \frac{g(u)du}{u^{p+1}}) = \theta(x^p(1+\int_1^x u^{-p}du) = \theta(x^p + \frac{x}{1-p} - \frac{x^p}{1-p})$. So the asymptotic complexity would be $\theta(x^p)$ when 1 < p < 2. So, we have $O(x^{1.365})$.

I also tried guess and prove. It showed that $O(n \log n) <$ O(T(n)) $\leq O(n^2)$ which means that what I get should be correct!

(b) **[6]** $T(n) = nT(\sqrt{n})^2$. This time, consider two base cases $T(1) = 4$ and $T(1) = 16$, and compute the answer in the two cases.

Answer:

$T(n) = nT(\sqrt{n})^2 = nT(n^{\frac{1}{2}})^2 = n[n^{\frac{1}{2}}(T(n^{\frac{1}{4}}))^2]^2 = n^2[T(n^{\frac{1}{4}})]^4 = n^3[T(n^{\frac{1}{8}})]^8 = ... = n^r[T(n^{\frac{1}{2^r}})]^{2^r}$

Since we have $T(n^{\frac{1}{2^r}})$, we cannot get T(1) any how.($n^{\frac{1}{2^r}} \neq 1$). We need T(2). $T(2) = 2T(\lfloor\sqrt{2}\rfloor)^2 = 2T(1)^2$. Since we have two different values for T(1) we can have :

$$\begin{cases} T(2) = 2 \times 16 = 32 ... if T(1) = 4 \\ T(2) = 2 \times 256 = 512 ... if T(1) = 16 \end{cases}$$

$n^{\frac{1}{2^r}} = 2 \Longrightarrow \log_2 n = 2^r \Longrightarrow r = loglog_2 n$

$T(n) = n^{\log\log n}[T(2)]^{2^{\log\log n}}$

Depending the initial value we have:

$T(n) = n^{\log\log n}(2^5)^{\log n} = n^{\log\log n}(2)^{\log n^5}$

or:

$T(n) = n^{\log\log n}(2^9)^{\log n} = n^{\log\log n}(2)^{\log n^9}$

We can rearrange the exponential representation to logarithmic and write the O based on that!

(c) **[6]** Suppose you have a divide-and-conquer procedure in which (a) the divide step is $O(n)$ (for an input of size n, irrespective of how we choose to divide), and (b) the "combination" (conquer) step is proportional to the product of the sizes of the sub-problems being combined. Then, (i) suppose we are dividing into two sub-problems; is it better to have a split of $(n/2, n/2)$, or is it better to have a split of $n/4, 3n/4$? (or does it not matter?) Next, (ii) is it better to divide into two sub-problems of size $n/2$, or three sub-problems of size $n/3$? (Assume that the goal is to minimize the leading term of the total running time.)

Answer:

We have a master theorem like the following:

$T(n) = aT(n/b) + cf(n) where a \geq 1$ and $b > 1 \Rightarrow$:

- $f(n) = O(n^{\log_b a - e})$ for some $e > 0$, then $T(n) = \Theta(n^{\log_b a})$
  if $f(n) = cn^k$ we have the same if a> b^k
- $f(n) = \Theta(n^{\log_b a})$ for some $e > 0$, then $T(n) = \Theta(n^{\log_b a} logn)$
  if $f(n) = cn^k$ we have the same if a= b^k

- $f(n) = \Omega(n^{\log_b a+e})$ for some $e > 0$, and af(n/b) $\leq$ cf(n), for some c ¡ 1 and for all n greater than some value then n' $T(n) = \Theta(f(n))$
  if $f(n) = cn^k$ we have the same if a $<$ b^k

We can write recursive formulas for the each case:

a) $T(n) = O(n) + O(n/2 \times n/2) + 2T(n/2)$. since k=2 and a=b=2; so we have a $<$ b^k and accordingly time complexity would be O(f(n)) which is $O(n^2)$

b) $T(n) = O(n) + O(n/4 \times 3n/4) + T(n/4) + T(3n/4)$.

Let's try Akra-bazi again.

$(1/4)^p + (3/4)^p = 1$ and hence p= 1.

$T(x) = \theta(x^p(1 + \int_1^x \frac{g(u)du}{u^{p+1}}) = \theta(x(1 + \int_1^x du) = \theta(x^2)$. So the asymptotic complexity would be $O(x^2)$ which is the same as part a. Please notice that the part a is better if we consider the terms with lower degree (e.g., for small values of n)

c) $T(n) = 3T(n/3) + O(n) + O(n/3 \times n/3 \times n/3)$. since k=3 and a=b=3 we have we have a $<$ b^k and accordingly time complexity would be O(f(n)) which is $O(n^3)$

According to obtained complexity, it is matter how we define conquer steps. Seems that the divide into two parts gives us the better answer!

Question 5: Computing intersections .... **[5]**

When answering search queries using an "inverted index", we saw that the key step is to take the intersection of the InvIdx sets corresponding to the words in the query. Suppose that these sets are of size $s_1, s_2, \ldots, s_t$ respectively (say we have t words in the query). Naïvely, computing the intersection takes time $O(s_1 + s_2 + \cdots + s_t)$, which is not great if one of the s_i is large (e.g., if one of the words in the query is a frequently-occurring one). Give an alternate algorithm whose running time is

$$O\big(s(\log s_1 + \log s_2 + \cdots + \log s_t)\big), \quad \text{where } s = \min_{1\leq i\leq t} s_i.$$

Answer:

The algorithm can be based on a binary search in sorted sets. We should pick a set with minimum members $s = \min_{1\leq i\leq t} s_i$. Pick an element of that set and search it in other sets using binary search. Searching in another set is O(logs) and we have to do it for all t sets. So, it takes $O\big((\log s_1 + \log s_2 + \cdots + \log s_t)\big)$. Since we have to do it for all member of selected set (e.g., $s = \min_{1\leq i\leq t} s_i$) , The complexity would be $O\big(s(\log s_1 + \log s_2 + \cdots + \log s_t)\big)$

Question 6: Dynamic arrays: is doubling important? .... **[8]**

Consider the 'add' procedure for dynamic arrays (DA) that we studied in class. Every time the add operation was called and the array was full, the procedure created a new array of twice the size and copied all the elements, and then added the new element.

Suppose we consider an alternate implementation, where the array size is always a multiple of some integer, say 100. Every time the add procedure is called and the array is full (and of size n), suppose we create a new array of size $n + 100$, copy all the elements and then add the new element.

For this new add procedure, analyze the asymptotic running time for N consecutive add operations.

Answer: In this case, we have: *n*, *n+100*, *n+200*, and so on in each step of new creation (instead of *2n* that we had in the class). for N consecutive add operations when the array is full we have:

$n + (n + 100) + (n + 200) + (n + 300) + \ldots\ldots$ Which is arithmetic progression with N element. So the summation would be $\frac{N(100+100N)}{2} = 50N + 50N^2 + Nn$. So, asymptotic running time for N is $O(N^2)$.