CS 6150: HW1 – Data structures, recurrences

Submission date: Tuesday, Sep 12, 2023 (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 (a)-(c) below, answer YES/NO, along with a line or so of reasoning. Simply answering YES/NO will fetch only half the points.

(a) [**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 $O(n^3)$? **Ans) Yes**

$f(n) <= c * g(n) \ \forall \ \ c, n >= n_0$

let c=1

$n^2 <= 1 * n^3 \ \forall \ n >= 1$

$f(n) \in O(n^3)$

Since Big − oh represents upper − bound, polynomial above n^2 *like* n^3, n^4 *are considered to be loose upper bounds so* $f(n)$ *belongs to* n^3.

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

Ans) No

$f(n) >= c * g(n) \forall c, n >= n_0$

$Let f(n) = n^4, \ c = 1$

$n^4 >= n^4$

But $f(n)$ *has lower − bound* $O(n^2)$, *the loose lower bounds would be* $n, 1$

$Therefore \ f(n)$ *can be* $O(n^2)$ *and below*

$Hence \ false.$

(c) [**3**] Let $f(n) = n^{\log_2 n}$. Is $f(n)$ in $o(2^n)$?

Ans) Yes

$Let f(n) <= c * g(n) \forall c, n >= n_0$

$n^{\log_2 n} <= 2^n$

Taking log both sides

$(\log_2 n)^2 <= n$

This is always true.

$Therefore f(n) \in o(2^n)$

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.)

Ans)

Lets consider an array of size n where n-k is sorted and k is unsorted.

We know that the maximum swap that can happen in an array is n-1.

Maximum iterations would be k-1 as k part is unsorted.

Therefore total time would be

$f(n) = (n-1) * (k-1)$

$f(n) = n - k - n + 1$

$We\ know\ 1 < k < n$

Therefore we ignore 1 and get f(n)=nk-k-n

$We\ know\ f(n)\ \in\ \theta(nk)$

$c_1 * g(n) <= f(n) <= c_2 * g(n)$

Lets solve for upper bound

$f(n) <= c_2 * g(n)$

$let\ c_2 = 1\ and\ g(n)\ be\ nk$

$nk - n - k <= nk$

$n + k >= 0$

This is true hence Lower bound is satisfied.

Lets solve for lower bound

$f(n) >= c_1 * g(n)$

$Let\ c_1 = 1/4\ and\ g(n)\ be\ nk$

$nk - n - k >= nk/4$

$3nk/4 - n - k >= 0$

Since K has to be an integer let $k >= 4$

$3n - n - 4 >= 0$

$2n - 4 >= 0$

$n >= 2$

This is true hence Upper bound is satisfied.

Since upper bound and lower bound both are satisfied for f(n), hence proved.

(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 $\log_2 n$.)

Ans)

$T(n) = T(n/3) + 2$

$T(n/3) = T(n/9) + 2$

$Therefore\ T(n)\ = T(n/3^i) + 2i$

This will occur till $T(1) = 1$ (*base case*)

$n/3^i = 1$

$n = 3^i$

$Taking\ \log_3\ both\ sides$

$i = \log_3 n$

Loop will iterate for $2i$ *times.*

There would be $2\log_3 n$ *comparisons.*

$Therefore\ final\ comparisions\ is\ 2\log_3 n.$

$We\ are\ comparing\ 2 * \log_3 n\ and \log_2 n$

Say we consider it for a value n=10

$2 * \log_3 10 = 2 * 2.09 = 4.18$

$\log_2 10 = 3.32$

Therefore we can see that log base 2 has lower value.

Therefore log base 2 is faster and is better.

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

As mentioned in class, the 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?

(Bonus: **[3]** extra points): in what sense is a "regular" binary search tree better?

Ans)

$No. = 1\ Binary = 1\ bits = 1$

$No. = 2\ Binary = 01\ bits = 2$

$No. = 3\ Binary = 11\ bits = 2$

$No. = 4\ Binary = 100\ bits = 3$

$No. = 8\ Binary = 1000\ bits = 4$

$We\ see\ that\ 2^n = (n+1)\ bits$

$Therefore\ 2(n-1) = n$

$Taking\ log\ both\ sides$

$n - 1 = \log_2 n$

$n = \log_2 n + 1\ bits$

$for\ example\ for\ n = 7, bits\ would\ be\ \log_2 7 + 1 = 3$

A) Whenever we add binary number in the prefix tree we see that the height of the tree for a particular number is equivalent to the number of binary bits.

$Since\ the\ range\ is\ [0, N^3]$

$Time\ taken = \log_2 N^3$

$= 3 \log_2 N$

$O(\log_2 N)$

B) Query searching time would be the same as number of bits.

$Therefore\ time = 3 \log_2 N$

$O(\log_2 N)$

Bonus)

We can see that poor man's binary search tree can only have two options like 0 or 1.

Regular Binary search tree on the other hand have better domain. They can take arrays as branches which can store more values, has more size.

Therefore Regular Binary search tree is better.

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 $T(1) = 1$.

Ans)

$T(n) = 2T(n/2) + T(n/3) + n$

Using Akra Bazzi

$\Sigma_{i=1}^{k} a_i T(b_i n) + g(n)$

$k = 2, a_1 = 2, b_1 = 1/2, a_2 = 1, b_2 = 1/3, g(n) = n$

$\Sigma_{i=1}^{k} a_i * {b_i}^p = 1$

$a_1(b_1)^p + a_2(b_2)^p = 1$

$2(1/2)^p + 1(1/3)^p = 1$

$p \simeq 1.3646$

$T(n) = \theta(n^p(1 + \int_1^n g(x)/x^{p+1}))dx$

$T(n) = \theta(n^{1.3646}(1 + \int_1^n x/x^{1.3646+1}))dx$

$T(n) = \theta(n^{1.3646}(1 + \int_1^n x^{-1.3646}))dx$

$T(n) = \theta(n^{1.3646}(1 + [x^{-0.3646}/1 - 1.3646]_1^n))dx$

$T(n) = \theta(n^{1.3646} - \frac{n}{0.3646} + \frac{n^1.3646}{0.3646})$

$T(n) = \theta(n^{1.3646})$

(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.

Ans)

$T(n) = n * T(\sqrt(n))^2$

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

$T(n) = n[\sqrt{n}T(n^{\frac{1}{2^2}})^2]^2$

$T(n) = n^i * [T(n)^{\frac{1}{2^i}}]^(2^i)$

We need to equate this to some base case. We can't equate it to 1 as else i would be tending to infinity.

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

$1/2^i * \log_2 n = 1$

Therefore we consider T(2) as base case. Now we calculate the value of T(2).

$T(2) = 2T(1.414)^2$

We consider T(1.414) as T(1)

$T(2) = 2 * 4^2 = 32$

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

Taking log both sides.

$1/2^i * \log_2 n = 1$

$2^i = \log_2 n$

$i = \log(\log_2 n)$

Finally T(n) would be

$n^{log_2 log_2 n} * 32^{2^{log_2 log_2 n}}$

For T(1)=16

T(2)=2*16*16=512

Finally T(n) would be

$n^{log_2 log_2 n} * 512^{2^{log_2 log_2 n}}$

(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.)

Ans) a)

$T(n) = 2T(n/2) + c(n/2)^2$

$T(n) = 2T(n/2) + c_1n^2 + c_2n$

$f(n) = c_1n^2 + c_2n, O(n^2)$

We apply master theorem.

$d = 2, a = 2, b = 2$

$\log_2 2 = 1, d = 2$

$d > \log_b a$

$O(n^2)$

b)

$T(n) = T(n/4) + T(3n/4) + c(n/4)(3n/4)$

$T(n) = T(n/4) + T(3n/4) + c_1n^2 + c_2n$

$f(n) = c_1n^2 + c_2n, O(n^2)$

We apply Akra bazzi

$\Sigma_{i=1}^{k} a_i T(b_i n) + g(n)$

$k = 2, a_1 = 1, b_1 = 1/4, a_2 = 1, b_2 = 3/4, g(n) = O(n^2)$

$\Sigma_{i=1}^{k} a_i * {b_i}^p = 1$

$a_1(b_1)^p + a_2(b_2)^p = 1$

$1(1/4)^p + 1(3/4)^p = 1$

$p = 1$

$T(n) = \theta(n^p(1 + \int_1^n g(x)/x^{p+1}))dx$

$T(n) = \theta(n^1(1 + \int_1^n x^2/x^2))dx$

$T(n) = \theta(n^1(1 + \int_1^n 1))dx$

$T(n) = \theta(n^1(1 + n - 1)dx$

$T(n) = \theta(n^2)$

$O(n^2)$

c)

$T(n) = 2T(n/2) + c(n/2)^2$

Since the recurrence is same so the answer would also be the same.

$O(n^2)$

d)

$T(n) = 3T(n/3) + c(n/3)^3 + O(n)$

$T(n) = 3T(n/3) + c_1 n^3 + c_2 n_2^2$

$f(n) = c_1 n^3 + c_2 n_2^2, O(n^3)$

We apply master theorem

$d = 3, a = 3, b = 3$

$\log_3 3 = 1, d = 3$

$d > \log_b a$

$O(n^3)$

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

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

Ans)

Let's consider set S1 S2 and S3 as sets which have the sizes s1,s2 and s3, s1 being the smallest.

Find the set with minimum size. Here it is S1 set with size s1.

Considering all the sets are in sorted order.

After finding the minimum set, we apply binary search to each set.

We know time for binary search for a set of size s_2 *and* s_3 *would be* $\log s_2$ *and* $\log s_3$

Therefore time $= s1(\log s_2 + \log s_3)$

This would be the time required for this example.

We can see this approach would be better than $O(s_1 + s_2 + \cdots + s_t)$ as we are taking the smallest set which reduces the time for searching.

Similarly for the question answer would be

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

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 amortized (average) running time for N consecutive add operations.

Ans) For n elements time taken would be n.

$n \to n$

$n + 100 \to n + 100 + n + 100[2n + 200]$

Here n+100 is creating time, n is copying time, 100 is adding elements time.

$n + 200 \to n + 200 + n + 100 + 100[2n + 400]$

Here n+200 is creating time, n+100 is copying time, 100 is adding elements time.

$n + 300 \to n + 300 + n + 200 + 100[2n + 600]$

$n + rd \to 2n + 200r$

Here r is the number of iterations and d=100.

$Considering N >> n$

Iterations r will stop when $n + rd >= N$.

$n + 100r >= N$

$r >= (N - n)/100$

Since N is very big.

$r \simeq N$

adding all the time for r iterations

$n + 2n + 200 + 2n + 400.....2n + 200r$

$n + 2nr + 100r + 100r^2$

Replacing r to N as $r \simeq N$

$n + 2nN + 100N + 100N^2$

Since N^2 *is the biggest term, all other terms are ignored when compared.*

$100N^2$

$Ammortized = Total\ time\ for\ N\ operations/N = 100N^2/N$

$100N$

$O(N)$