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 $O(n^3)$?

Answer:

YES. Since $f(n) \in O(n^2) \implies \exists c, n_0$ such that for all $n \geq n_0$, $f(n) \leq cn^2$, it does also imply that for the same n_0 and c we have that $f(n) \leq cn^3$ (since $n^3 \geq n^2$ for $n \geq 1$).

YES/NO answer	1 point
Providing a valid explaination	1 point

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

Answer:

NO. Remember that we say that a function $f(n)$ is in $\Omega(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0$$

With this definition we can see that $f(n) = n^3$ is in $\Omega(n^2)$ but not in $o(n^3)$.

YES/NO answer	1 point
Providing a valid explaination	1 point

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

Answer:

YES. Note that $f(n) = n^{\log_2 n}$ is the equal to $2^{(\log_2 n)^2}$ (given that the logarithms are base 2). Therefore, $f(n)$ is clearly in $o(2^n)$ since for any c we can always find an n_0 such that $c \cdot 2^n$ is larger than $2^{(\log_2 n)^2}$ for all $n \geq n_0$

Note: Finding an n_0 for c- If $c \geq 1$, for $n = e^x \geq n_0 = e^{18}$, we can see that $2^n = 2^{e^x} > 2^{1+x+x^2/2+x^3/6} > 2^{(\log_2 e^x)^2} = 2^{(\log_2 n)^2}$ and therefore, $c2^n > 2^{(\log_2 n)^2}$. For $c < 1$, we can see that for $n = e^x \geq n_0 = e^{f(c)}$ (where $f(c)$ is some function of c) we can get similar bounds. We can prove the existance of this function. Let $f(c)$ be the point where $y > f(c) \implies y^3/6 - ((\log_2 e)^2 - 1/2)y^2 + y + 1 - \log_2 1/c > 0$ (naively picking $y > \max\{6((\log_2 e)^2 - 1/2), \log_2 1/c\}$ works). Then, $n = e^x > n_0 = e^{f(c)} \implies x > f(c) \implies x^3/6 - ((\log_2 e)^2 - 1/2)x^2 + x + 1 - \log_2 1/c > 0$. Now we can see that $c2^n = c2^{e^x} > c2^{x^3/6+x^2/2+x+1} > c2^{((\log_2 e)^2 x^2 + \log_2 1/c} = 2^{(\log_2 n)^2}$. This shows that for any c, there is an n_0 such that for $n \geq n_0$, $c2^n > 2^{(\log_2 n)^2}$

YES/NO answer	1 point
Providing a valid explaination	2 points

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:

Consider the array B where $B[i] \leq B[j] \forall i \leq j$.
Now consider the array $A = \{B[n], B[n-1], \ldots, B[n-k+1], B[1], B[2], \ldots, B[n-k]\}$.
Note that at each iteration, the bubble sort takes the largest element and puts it in the correct

place while shifting all the others one step forward.
Each iteration takes $n-1$ comparisons since it has to compare each element with the neighbor and potentially swap and then move forward.
Since at the end of i^{th} iteration the i^{th} largest number goes to the correct location, we can see that after k iterations all k largest elements are in place and since the others are sorted the algorithm will check to see that there are no swaps and terminate on $(k+1)^{th}$ step.
Therefore, we have exactly $(n-1)(k+1)$ iterations, meaning the algorithm takes at least $(n-1)(k+1)$ iterations (lower bound) and not more than $(n-1)(k+1)$ iterations (upper bound).
Therefore the time taken is $\Theta(nk)$

Providing a valid example (for parameters n,k)	2 points
Proving the upper bound	1 point
Proving the lower bound	1 point

(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:

Note that now we have a case where we have smaller search areas each of size $n/3$ but to select the correct search area we would need to compare the x with $A[n/3]$ and $A[2n/3]$ so overall 2 comparisons each time. Therefore, we can write the recursive function for comparisons $C(n)$ as,

$$\begin{aligned} C(n) &= C(n/3) + 2 \\ &= C(n/9) + 2 + 2 \\ &\vdots \\ &= C(n/3^r) + \sum_{i=1}^{r} 2 = C(n/3^r) + 2r \end{aligned}$$

Solving this gives us, $C(n) = 1 + 2\log_3 n = 1 + \frac{2}{\log_2 3}\log_2 n$. We can see that since $2 > \log_2 3$, the number of comparisons we do are now more than $\log_2 n$ (even though assymptotically we are doing the same).

Deriving the proper recurssion	2 points
Solving for the number of comparisons and arguing which one is better	2 points

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?

Answer:

Note that the number of bots needed to represent a number is $\log N^3$. Therefore, the depth of the prefix tree is $\log N^3 = 3\log N$. Therefore,
(a) If we want to add an element we can simply go down the path given by the binary representation of the element and if it is marked as an element we are done else we can mark it or add the necessary bit nodes. Since at each level any of these operations are constant time, we can do the addition in $3\log N \in O(\log N)$ time steps.

(b) Search is also $O(\log N)$ since we just go down the prefix tree trying to see if we can find the path represented by x and that only takes $3 \log N$ steps.

Deriving the correct time taken for an insert (with correct arguments)	3 points
Deriving the correct time taken for a search (with correct arguments)	2 points

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

Answer:

The binary tree is faster since it takes $\log N$ comparisons instead of $3 \log N$ comparisons, but in terms of asymptotic complexity they are the same since they both belong to $O(\log N)$

Valid arguments on why binary trees could be better than tries in this case	3 points

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:

Since $T(n) \geq 2T(n/2) + n$ we can see that using an inductive argument, $T(n) \geq n \log n$. Assume $T(n) = A\lfloor n^c \rfloor$ for some $c > 0$ (and $n \geq n_0$ for some n_0). Using the given recursion, we can get $An^c = 2A\lfloor \frac{n}{2} \rfloor^c + A\lfloor \frac{n}{3} \rfloor^c + n \leq 2A\frac{n^c}{2^c} + A\frac{n^c}{3^c} + n$ which gives $An^{c-1}\left(1 - \frac{2}{2^c} - \frac{1}{3^c}\right) \leq 1$ (for all n) and this implies $1 = \frac{2}{2^c} + \frac{1}{3^c}$ which gives $c \approx 1.3646$ which implies $T(n) \in O(n^{1.3646})$

Alternative answer:

Note that Akra-Bazzi method tells us, given $T(x) = h(x) + \sum_{i=1}^{k} a_i T(b_i x + h_i(x))$ for $x \geq x_0$ (x_0 constant), where $a_i > 0, 0 < b_i < 1$ are constants for all i, $|g'(x)| \in O(x^c)$ (for a constant c) and $|h_i(x)| \in O\left(\frac{x}{(\log x)^2}\right)$ for all i, then $T(x) \in \Theta\left(x^p\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$ where $\sum_{i=1}^{k} a_i b_i^p = 1$.

We can see that in our case $g(x) = x$ and $a_1 = 2, a_2 = 1, b_1 = 1/2, b_2 = 1/3$. This gives us $T(x) \in O\left(x^p\left(1 + \int_1^x \frac{u}{u^{p+1}} du\right)\right) \equiv O\left(x^p\left(1 + \int_1^x u^{-p} du\right)\right) \equiv O\left(x^p\left(1 + \frac{x^{-p+1}}{-p+1} - \frac{1}{-p+1} du\right)\right) \equiv O(x^p)$ where $\frac{2}{2^p} + \frac{1}{3^p} = 1$. Solving this gives us $p \approx 1.3646$. Therefore we get $T(n) \in O(n^{1.3646})$

If the big-O bound given is not tight but reasonable (with proof)	3 points
If the big-O bound given is tight	5 points

(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:

We can see that,

$$\begin{aligned} T(n) &= nT(n^{1/2})^2 \\ &= n\left(n^{1/2}T(n^{1/4})^2\right)^2 = n^2 T(n^{1/4})^4 \\ &\vdots \\ &= n^i T(n^{1/2^i})^{2^i} \end{aligned}$$

We can prove this using induction. Assuming, $T(n) = n^i T(n^{1/2^i})^{2^i}$ for all $i \leq k$. Then for $i = k+1$ we get $T(n^{1/2^k}) = n^{1/2^k} T(n^{1/2^{k+1}})^2$ and $T(n) = n^k\left(n^{1/2^k} T(n^{1/2^{k+1}})^2\right)^{2^k} = n^{k+1} T(n^{1/2^{k+1}})^{2^{k+1}}$

and since we already have the base case with the recursion, we can claim by induction, $T(n) = n^i T(n^{1/2^i})^{2^i}$ for all i.

Note that we can do this reduction until we get close to 2, i.e., when $n^{1/2^i} \geq 2$ and $n^{1/2^{i+1}} < 2$, we can see that, $n \geq 2^{2^i}$ and $i \leq \log\log n$. Therefore the time complexity $T(n) = n^{\log\log n}(2T(1)^2)^{\log n}$. Therefore, for $T(1) = 4 = 2^2$,

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

and with $T(1) = 16 = 2^4$,

$$T(n) = (2 \times (2^4)^2)^{\log n} n^{\log\log n} = 2^{9\log n} n^{\log\log n} = n^{\log\log n + 9}$$

Solving the recurssion formula to get a valid solution	4 points
Solving for the case where $T(1) = 4$	1 point
Solving for the case where $T(1) = 16$	1 point

(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(i):

So the key point to compare the running time performance of two split strategy is to calculate the corresponding conquer time!

$\left(\frac{n}{2}, \frac{n}{2}\right)$ split:

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

$$T(n) = 2^{\log n} T(\frac{n}{2^{\log n}}) + c\frac{n^2}{2^{\log n+1}} + c\frac{n^2}{2^{\log n}} + \cdots + c\frac{n^2}{2^2} + O(n)$$

$$T(n) = nT(1) + cn^2(\frac{1}{2} - \frac{1}{2^{\log n+1}}) + O(n)$$

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

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

$\left(\frac{n}{4}, \frac{3n}{4}\right)$ split:

From the recursion running time equation, the running time upper-bound should be in this form:

$$T(n) \leq \alpha n^2 + O(n)$$

We have the recursion running time equation:

$$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + c\frac{n}{4}\frac{3n}{4} + O(n)$$

Substitute the guess upper-bound into the recursion equation, we have:

$$T(n) \leq \alpha(\frac{n}{4})^2 + O(n) + \alpha(\frac{3n}{4})^2 + O(n) + c\frac{n}{4}\frac{3n}{4} + O(n)$$

$$\alpha\frac{n^2}{16}+\alpha\frac{9n^2}{16}+c\frac{3n^2}{16}+O(n)\le\alpha n^2+O(n)$$
$$\alpha\ge\frac{1}{2}c$$

So the guess of running time upper-bound is: $\frac{1}{2}cn^2+O(n)$
Prove: We know according to the problem, $T(0)=T(1)=T(2)=T(3)=0$.
Base case:

$$T(4)=T(1)+T(3)+3c+O(4)=3c+O(4)\le\frac{1}{2}c4^2+O(4)=8c+O(4)$$

Base case works.
Suppose if $k<n$, the running time satisfies $T(k)\le\frac{1}{2}ck^2+O(k)$, Then, we have:

$$T(n)=T(\frac{n}{4})+T(\frac{3n}{4})+c\frac{n}{4}\frac{3n}{4}+O(n)$$

we know that $\frac{n}{4}<n$, $\frac{3n}{4}<n$, So

$$T(\frac{n}{4})\le\frac{1}{2}c(\frac{n}{4})^2+O(\frac{n}{4})$$
$$T(\frac{3n}{4})\le\frac{1}{2}c(\frac{3n}{4})^2+O(\frac{3n}{4})$$
$$T(n)\le\frac{1}{2}c(\frac{n}{4})^2+O(\frac{n}{4})+\frac{1}{2}c(\frac{3n}{4})^2+O(\frac{3n}{4})+c\frac{n}{4}\frac{3n}{4}+O(n)$$
$$T(n)\le\frac{1}{2}cn^2+O(n)$$

The guess upper-bound running time is correct.
In the prove above, if the sign $\le$ was replaced with $\ge$, the logic also worked. So $\frac{1}{2}cn^2+O(n)$ is a tight upper-bound running time.
we can see that the running time upper-bound of two splits share similar dominant term. So two splits enjoy similar running time performance.
In this case, the split method does not matter.
Answer(ii):
$\left(\frac{n}{3},\frac{n}{3},\frac{n}{3}\right)$ split:

$$T(n)=3T(\frac{n}{3})+c\frac{n^3}{3^3}+O(n)$$

It's clear that $T(n)\ge n^3$, the dominant running time is $o(n^3)$. So it's worse than $\left(\frac{n}{2},\frac{n}{2}\right)$

Correctly arguing the solution of part (i)	3 points
Correctly arguing the solution of part (ii)	3 points

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

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

Answer:

Algorithm 1 Compute Intersections

Let s be the smallest InvIdx set, set $A = I = \arg\min_i |InvIdx_i|$ be the set of initial intersections.
for d in A **do**
  for i from 1 to t **do**
    Use binary search on $InvIdx_i$ to find d
    if d not in $InvIdx_i$ **then**
      Remove d from I
    end if
  end for
end for
Return I as the intersection.

Assumptions: The InvIdx sets are sorted (inside) and we have access to a size property to figure out the smallest InvIdx set (or sorted by their size). Let $InvIdx_i$ indicate the inverted index for word w_i. We know that $s_i = |InvIdx_i|$

Runtime: Note that in the algorithm 1, we start with an intersection $I = \arg\min_i |InvIdx_i|$ ($|I| = s$). For each document d, we run t steps checking each InvIdx set. For a fixed document we will take $O(\log s_1 + \log s_2 + \cdots + \log s_t)$ time to search all the inverted indices. The size of intersection does not increase (only decrease or stay the same). Since we go over s elements (in the loop over A), overall time complexity is $O\left(s\left(\log s_1 + \log s_2 + \cdots + \log s_t\right)\right)$

Correctness: Note that any document in the intersecion should be in the samllest set $A = I = \arg\min_i |InvIdx_i|$ (if document d is not in I then it is not in the intersection by definition). The algorithm goes over each document in A. Each time we see a document d in A, we go over each $InvIdx$ set and search for d. Note that we remove d from I if we do not find it in a invertex index set. Note that if the document is in all sets then it has to be in the intersection as well. And if it is not in at least one set it will be removed from I. Therefore, at the end all the documnets left in I are in all the sets and there cannot be any document that is in all the sets and not in I. Therefore, this gives us the intersection.

Providing a correct algorithm	2 points
Proving that the algorithm has the desired runtime	2 points
Proving that the algorithm is correct	1 point

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.

Answer:
Let's first define running time of every operation:
Let m be the number of elements in the array when it is full, then in our step of increasing the array size, creating a new array takes $O(1)$ time and copying elements takes $O(m)$ time.
Pure append operation (without array resize) consumes $O(1)$ time.
Then, Suppose the initial size of array is $100K_0$, with $K_0 \geq 1, K_0 \in Z^+$;
Thus, to hold N elements, the size of the array must be at least $100\lceil \frac{N}{100} \rceil$.
Let's set $\hat{K} = \lceil \frac{N}{100} \rceil$; then we need to resize array $\hat{K} - K_0$ times.

So for the ith size array, with $1 \leq i \leq \hat{K} - K_0$, the running time of each operation is following:
Creating a new array of size $100(K_0 + i)$: $O(1)$;
Copying elements of number $100(K_0 + i - 1)$: $O(100(K_0 + i - 1))$;
Append new element: $O(1)$.
So the total running time of add N elements can be shown as:

$$= \hat{K} - K_0 + N + \sum_{i=1}^{\hat{K}-K_0} 100(K_0 + i - 1)$$

$$= \hat{K} - K_0 + N + 100 \cdot (\hat{K} - K_0) \cdot (K_0 - 1) + \sum_{i=1}^{\hat{K}-K_0} 100i$$

$$= \hat{K} - K_0 + N + 100 \cdot (\hat{K} - K_0) \cdot (K_0 - 1) + 100\frac{(\hat{K} - K_0 + 1) \cdot (\hat{K} - K_0)}{2}$$

$$= N + 50 \cdot (\hat{K}^2 - K_0^2) - 49 \cdot (\hat{K} - K_0) = O(N + 50\lceil\frac{N}{100}\rceil^2 - 49\lceil\frac{N}{100}\rceil) = O(N^2)$$

The amortized running time for N consecutive add operations is $\frac{O(N^2)}{N} = O(N)$

Arguing the time complexities of different types of operations	4 points
Calculating the total work and averaging to get the ammortized time	4 points