CS 6150: HW1 – Data structures, recurrences

Submission date: Monday, Sep 13, 2021 (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 basics	5	
Deletion in prefix trees	6	
Binary search and test pooling	10	
Recurrences, recurrences	16	
Dynamic arrays: is doubling important?	6	
Total:	50	

Note. *When asked to describe and analyze an algorithm, you need to first write the pseudocode, provide a running time analysis by going over all the steps (writing recurrences if necessary), and provide a reasoning for why the algorithm is correct.* Skipping or having incorrect reasoning will lead to a partial credit, even if the pseudocode itself is OK.

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: YES

(b) [**2**] Let $f(n)$ be a function of integer parameter n, and suppose that $f(n) \in O(n \log n)$. Is it true that $f(n)$ is also $O(n^2)$?
Answer: YES, if $f(n) \in O(n \log n)$, because this is Big-O notation, which means upper bound running time is $O(n \log n)$ then we could say $f(n)$ is also $O(n^2)$.

(c) [**2**] Suppose $f(n) = \Omega(n^{2.5})$. Is it true that $f(n) \in o(n^5)$?
Answer: NO, because $f(n) = \Omega(n^{2.5})$, Omega means the running time is no slower than $n^{2.5}$, but we could not know the upper bound from question, for example, $f(n)$ could be n^6, so we could not say $f(n) \in o(n^5)$

(d) [**2**] Let $f(n) = n^{\log n}$. Is $f(n)$ in $o(2^{\sqrt{n}})$?
Answer: NO, counter example, if n = 4, $n^{logn} = 16$, but $2^{\sqrt{n}} = 4$, 16 greater than 4, so we could not say $f(n)$ in $o(2^{\sqrt{n}})$.

Question 2: Bubble sort basics ..... **[5]**

Recall the bubble sort procedure we saw in Lecture 1 (see 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. As I mentioned in class, the running time of the algorithm depends on the input.

Given a parameter $1 < k < n$, give an input array $A[]$ for which the bubble sort procedure takes time $\Theta(nk)$. (Recall that to prove a $\Theta(\cdot)$ bound, you need to show upper and lower bounds.)
Answer: I will start with fastest case, for example, give array A[1,2,3...n] sorted with increasing order already, then apply bubble sort on this array, since no integer need to be swapped, only things are compared integer and go through every items in this array which is n, since compared take $O(1)$, go through every items take $O(n)$ lower bound is $O(n)$, then $k = 1 + O(1)$.
Worst case is, given array A[n,n-1,n-2....1] with reverse sorted, then apply "bubble sort" sorts to increasing order, compared still take O(1), but every items need to swap with every other items, swap is O(1), every other items are n-1, so one item takes $(n-1) * 1$. There are n items in array, so running time $n * (n-1)$ which $k = n - 1 +$ some kind of compared time, then $1 < k < n$

Question 3: Deletion in prefix trees ..... **[6]**

In class, we saw how a prefix tree can be used to store a *set of strings* (which we called a dictionary) over some alphabet Σ. Specifically, we saw how to implement the `add` and `query` operations on such a data structure. (See the lecture notes for more details.) Now, consider implementing the `delete` operation. As suggested in the notes, one way to do this is to mimic the query, and for the node corresponding to the word, set the "IsWord" boolean to false. However, if we are adding and removing multiple strings, this can lead to many tree paths that were created, but don't correspond to any word currently in the dictionary.

Show how to modify the data structure so that this can be avoided. More formally, if S is the set of words remaining after a sequence of add/delete operations, we would like to ensure space utilization that is the same (possibly up to a constant factor) of the space needed to store only the elements of S. If your modifications impact the running time of the `add`, `query`, and `delete` operations, explain how.
Answer: Start from deletion Algorithm, for example, there is a word = c1+c2+c3...

start with n = root node (c1).
if(n.hasChildNode == false)
return false;
for(i = 1; $i <$ *word.length*, i++)
delete n;
n = n.child (next character);
then, mark n as invalid

Running time: the deletion is to delete every character of the word, it will be O(word.length).

Question 4: Binary search and test pooling ..... **[10]**
"Test pooling" is a trick that is used when testing for a disease is expensive or has limited availability. The idea is the following: suppose we have n people (numbered $1, 2, \ldots, n$ for convenience), instead of testing each one, samples from a subset S of the people are combined and tested, where a test runs in $O(1)$ time regardless of the size of S. If at least one of the people in S has the disease, the test comes out positive, and if none of the people in S has the disease, it comes out negative. (Let us ignore the test error for this problem.)

It turns out that if only a "few" people have the disease, this is much better than testing all n people.

(a) **[4]** Suppose we know that *exactly one* of the n people has the disease and our aim is to find out which one. Describe an algorithm that runs in time $O(\log n)$ for this problem. (For this part, pseudocode suffices, you don't need to analyze the runtime / correctness.)
Answer: I will write the preudocode code, given array people with size S

public findDisease(array people)

if(people.size = 1)
return this people with disease;

leftArrayPeople = left half of people array;
rightArrayPeople = right half of peopel array;
if(leftArrayPeople.test == true)
findDisease(leftArrayPeople);
else
findDisease(rightArrayPeople);

(b) **[6]** Now suppose we know that *exactly two* of the n people have the disease and our aim is to identify the two infected people. Describe **and analyze** (both runtime and correctness)

an algorithm that runs in time $O(\log n)$ for this problem.
Answer: Algorithm:

public findDisease(array people)

if(people.size = 1)
return this people with disease;

leftArrayPeople = left half of people array;
rightArrayPeople = right half of peopel array;

if(leftArrayPeople.test == true and rightArrayPeople == true)
findDisease(leftArrayPeople);
findDisease(rightArrayPeople);
return;

if(leftArrayPeople.test = true)
findDisease(leftArrayPeople);
else
findDisease(rightArrayPeople);

Correctness:
Base case is when array only have one people which is the people with disease, and also in my algorithm, I cover special case after array divide into left and right, they both have positive, run left and right Synchronously, then find them separately. size S will always divide by 2 to find the case.

Running time:
Because the size S will always divide in to half, and test runs in O(1) regardless of size S, just keep splitting S into 2, until S = 1, we can find that case, which take O(log n). The worst case is having both left and right one positive after first split, so worst case running time will be 2*O(log n).

Question 5: Recurrences, recurrences ..... **[16]**
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 or equal to the number) of the corresponding quantity. **Please show how you obtained your answer.**

(a) **[4]** $T(n) = 3T(n/3) + n^2$. As the base case, suppose $T(n) = 1$ for $n < 3$.
Answer: I will use master theorem, a = 3, b = 3, f(n) = n^2, then $n^{log_b^a}$ = n, then we need to calculate ϵ compare to f(n), which is $n^{log_b^a+\epsilon} = n^2$, then $\epsilon = 1$, apply to case 3 of master theorem, we get T(n) = $\Theta(n^2)$, which is $O(n^2)$

(b) **[6]** $T(n) = 2T(n/2) + T(n/3) + n$. As the base case, suppose $T(0) = T(1) = 1$.
Answer: I will use master theorem for this question, get a f(n) = n, for $T(n) = 2T(n/2)$,

a =2, b =2, we need to calculate $n^{log_b^a+\epsilon} = n$, $\epsilon = 0$,apply case 2, we get T(n) = $\Theta(nlogn)$, then analyze $T(n/3)$ with a = 1, b =3, f(n) = n, then calculate $n^{log_b^a+\epsilon} = n$, we get $\epsilon = 1$ because $log3^1 = 0$, then apply case 3, we have T(n) = $\Theta(n)$, combine $\Theta(nlogn)$ and $\Theta(n)$ together, we can get $O(nlogn)$.

(c) [**6**] $T(n) = 2(T(\sqrt{n}))^2$. As the base case, suppose $T(1) = 4$.
Answer: I will plug and chug for this question, because the steps are really hard to show on overleaf, so I will combine a individual steps at next page. But the result I will put it right here, I have k depth, then I got $2^{2^k-1} * (T(n^{1/2^k}))^{2^k}$, which means $n^{1/2^k} = 1$, then $n = 1^{2^k}$, which means n =1, then we can get O(1), constant time.

Question 6: Dynamic arrays: is doubling important? ..... **[6]**
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 32. Every time the add procedure is called and the array is full (and of size n), suppose we create a new array of size $n + 32$, 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: I will analyze the running time of creating a new array of twice size first. Growing the array size by scaling by any constant factor will be sufficient to get the run time to be O(n), the total work done growing the array will be (1 + n + n^2 + ... + $n^{1+log_n^k}$) until it reaches to size of k. So we can see, the number of quantity creating array is about O(log n), so that the total running time would be "running time of creating new array in memory * O(log n)."
But if we consider this alternate implementation method, the total number of quantity creating array will be $(k-1)/32$ until size of k, which is about O(n), then the total running time would be "running time of creating new array in memory * O(n)." which is clearly taking more time compared to double size of array, so doubling size is important.

$K=1$ $\quad T_n = 2(T(\sqrt{n}))^2 = 2(T(n^{\frac{1}{2}}))^2$

$K=2$ $\quad 2\cdot(2\cdot T(n^{\frac{1}{2^2}})^2)^2$

$= 2^3 \cdot T(n^{\frac{1}{2^2}})^{2^2}$

$K=3$ $\quad 2\cdot[2^3\cdot T(n^{\frac{1}{2^3}})^{2^2}]^2$

$= 2^7 \cdot T(n^{\frac{1}{2^3}})^{2^3}$

$K=4$ $\quad = 2\cdot[2^7\cdot T(n^{\frac{1}{2^4}})^{2^3}]^2$

$= 2^{15}\cdot T(n^{\frac{1}{2^4}})^{2^4}$

$K=5$ $\quad = 2\cdot[2^{15}\cdot T(n^{\frac{1}{2^5}})^{2^4}]^2$

$= 2^{31}\cdot T(n^{\frac{1}{2^5}})^{2^5}$

We can conclude

$$2^{2^{K-1}-1}\cdot T(n^{\frac{1}{2^K}})^{2^K}$$