1 Q1: Basics

1.1 Sign up for the course on Piazza!

Done.

1.2 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 $O(n^2)$?

YES.

A function $f(n)$ that is an element of $O(n \log n)$ has a growth rate with an upper bound of $n \log n$. Since n^2 is larger than $n \log n$ as n increases, n^2 is also an upper bound for the function and thus the function can also be considered an element of $O(n^2)$.

1.3 Suppose $f(n) = \Omega(n^{2.5})$. Is it true that $f(n) \in o(n^5)$?

NO.

$f(n) = \Omega(n^{2.5})$ tells us that the function $f(n)$ has a growth rate greater than or equal to $n^{2.5}$; it tells us nothing of the upper bound. Therefore, it's unknown if this function, $f(n)$, has a strict upper bounded growth rate of n^5.

1.4 Let $f(n) = n^{\log n}$. Is $f(n)$ in $o(2^{\sqrt{n}})$?

Here's some values I used to help me decide which was bigger:

$n = 10, n^{\log n}, 2^{\sqrt{n}}$

$n = 10, 2.0 \times 10^2, 9.0$

$m = 100, 1.6 \times 10^9, 1.0 \times 10^3$

$n = 1000, 5.3 \times 10^{20}, 3.3 \times 10^9$

$n = 10000, 6.9 \times 10^{36}, 1.3 \times 10^{30}$

$n = 100000, 3.7 \times 10^{57}, 1.6 \times 10^{95}$

YES.

If $n^{\log n}$ is in $o(2^{\sqrt{n}})$, then there $\lim_{n \to \text{inf}} \frac{n^{\log n}}{2^{\sqrt{n}}} = 0$. I'm assuming there is something clever we can do here with l'hopital's rule, but I haven't been able to find it. I used wolfram to check that the limit is zero in the meantime. We can also see from the growth rate in the values above that $2^{\sqrt{n}}$ starts to grow much faster thus you'd be dividing by a larger number in the limit as n grows.

2 Q2: Bubble sort basics

2.1 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.)

On each iteration of bubble sort an element can only move 1 position to the left; this is a consequence of us scanning from the left to the right – we swap to the left and then check the element immediately to the right of the swapped element. Thus, we only consider it one time. Since this is the case, we can carefully craft an array for any value $k; 1 < k < n$ where the smallest element of the array (the one that will finish farthest to the left when the array is sorted) is the kth element of the array and all other elements are sorted.

I'm not sure how to give a runtime complexity for this algorithm given that the runtime is always going to be exactly nk given how the array is crafted, but here goes:

Since the composition of $A[]$ is a known singular array (not a multitude of array types), the upper and lower bounds will be the same. Since all other elements other than the one in the kth position are sorted (if the element in the kth position was removed), when we compare any other element with the element that starts in the kth position, they will be moved to their final resting place. Thus we just need to wait for the element that starts at the kth position to move to the left until it settles into the 0th index. This takes exactly k iterations since it can only move one spot in each iteration.

Since the upper and lower bounds are both nk, running the array that I crafted through bubble sort takes time $\Theta(nk)$.

3 Q3: Deletion in prefix trees

3.1 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.

Algorithm 1 delete(string valueToDelete)

1: Move to node valueToDelete[0]
2: if length of valueToDelete greater than 1, delete(substring of valueToDelete from 1st index to end)
3: if node is leaf remove the leaf

If we modify the delete operation to recursively delete leaves that would have been set to false in the original delete algorithm, then we can save some space. This reduces the space required by making the paths that used to lead to more nodes point to null instead. By pruning these leaves off the tree recursively as described in the algorithm above, we're guaranteed that the space complexity is the same as the tree that is built just from S. This is because all words lead to intermediary nodes on the tree or to leaves. If we prune leaves we're not using, then the only ones left would be intermediary nodes, which still take up space when the tree represents S.

A consequence of this operation is that when we're querying for words that don't exist, we might have faster query paths say there was a path for "ZZZ" in the original dictionary and that there is no word that begins with "Z" in the new dictionary S. Then, if we were searching for "ZZZ", we could immediately return

"no such word" when we search the first Z, instead of having to keep traverse down the tree to the node representing "ZZZ". This reduces the number of searches we have to do in this case because we don't traverse nodes which lead to false.

The effect isn't so positive for add and delete. In the case of delete, we now have to remove a number of nodes up to the length of the word we're deleting. This is much slower than just setting the value of the terminating node to false. In fact it now takes $2 * \log(n)$ operations instead of $\log n$ For adding, we might have to add nodes that were previously deleted when removing a word. this leads to more assignments and a longer run time. In the case above for query speed, if we wanted to add "ZZZ" back to the dictionary, it would now have to rebuild 3 nodes, instead of just setting one value to true. In the worst case, that could mean adding as many nodes as there are letters in the new word, which could be costly if that happens repeatedly.

4 Q4: Binary search and test pooling

4.1 '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.

4.2 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.)

Algorithm 2 log n testing

1: Split the samples into to groups of size n/2 and run the test
2: One group will be positive, take the positive group back to step 1 and repeat until you're down to a group with only one sample that tests positive (this is the base case).

This runs in $\log n$ time for the same reason that binary search does. We're able to discard half of the remaining samples each time we test. The number of samples in each group as a function of the number of tests done (k) is $n/2^k$. We're down to one sample when $n/2^k = 1; k = log_2 n$.

4.3 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.

Let's use the same algorithm as above with a slight modification. When we split the sample and both positive samples are in one group keep testing just the one group. When there are 2 groups keep splitting up the two groups and testing each group.

There are 2 cases to consider, either one of the groups has both cases, or the test is split among 2 groups. When the test is split among two groups, run the above algorithm on each of the groups individually. Since that algorithm ran in $\log n$ time, the algorithm here will run in $2 \times \log n \in O(\log n)$ time. If the positive samples are in just one group, continue with the algorithm until they're split apart into two separate groups. We know that this second case will also run in $\log n$ time since we're not continuing our algorithm on the group that all tested negative, thus we only need to continue testing half of the samples. This continues until they're split up and then we have $2 \times \log n$ from the time that they're split. This would mean the entire algorithm would run in $\log n$ time.

Since we've considered the two possible cases, this shows that our algorithm is correct and runs in $\log n$ time regardless of the case.

5 Q5: Recurrences, recurrences

5.1 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.

5.2 $T(n) = 3T(n/3) + n^2$. As the base case, suppose $T(n) = 1$ for $n < 3$.

Using the master theorem, we can see this is $O(n^{log_3 3}) + n^2 = O(n^2)$

5.3 $T(n) = 2T(n/2) + T(n/3) + n$**. As the base case, suppose** $T(0) = T(1) = 1$**.**

Using Akra-Bazzi we end up with $2(1/2)^p + (1/3)^p = 1; p \approx 1.3646$, this isn't a nice number, but we can do some clever symbol pushing to simplify the integral in the Akra-Bazzi formula a little to make things nice for us:

$\Theta(n^p + n^p \int_1^n u/u^{p+1})$
$\Theta(n^p + n^p \int_1^n u^{-p})$
$\Theta(n^p + n^p(\ln p - \ln 1)$
$\Theta(n^p + n^p \ln p) = \Theta(n^{1.3646})$

5.4 $T(n) = 2(T(\sqrt{n}))^2$**. As the base case, suppose** $T(1) = 4$**.**

After plugging and chugging, I've found that the recurrence takes the form $2^{k-1}T(n^{1/2^k})^{2^k}$. Take the base case as $n = 2; t(n) = 32$ (by plugging in n = 1 into the formula). Thus when $n^{1/2^k} = 2$, rearranged to $k = \log(1/log_n 2)$.

At this point I'm a bit lost as to how to proceed...

6 Q6: Dynamic arrays is doubling important?

6.1 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.

Algorithm 3 delete(string valueToDelete)

1: if array is full, make new array that is 32 slots larger and copy values from prior array
2: Add value to array

In the above algorithm, we need to copy the array we have every time we add 32 new elements. This means that if we're adding N elements, we need to increment the size of the array $N/32$ times. The runtime of copying the array

will be dependent on the size of the current array, since we need to read each element currently stored and move it to the new array. Thus if the array is size k at some given time (and is fill), it will take roughly (a constant factor of) k operations to copy the array when we add a new element.

Let's work backwards to determine how many times we need to expand the the array when adding N elements. Say we got lucky and the last element we added filled the array. In this case we expanded the array at $N - 32$ element, $N - 64$ elements, etc. This will continue until we have $N - x * 32 < 32$ elements, our base case. In this case we have to expand the array $x = \lfloor N/32 \rfloor$.

We also need to copy elements each time the array expands. Let's calculate how many elements need to be copied at each expansion (again backwards). At the expansion when the array his $N - 32$ elements, we need to copy those $N - 32$ elements to a new array. Then at $N - 64$ elements, we need to copy those $N - 64$ elements to a new array. This pattern continues in a general pattern, we need to copy $N - x * 32$ elements each time we hit our capacity where x is the number of iterations until our array is large enough to hold all the values.

That is we need to do $\lfloor N/32 \rfloor$ expansions where each expansion copies $N - x \times 32$ for $0 \leq x \leq N/32$. That is, this algorithm takes $\lfloor N/32 \rfloor * (N - x * 32)$ time, which is $O(n^2)$.