Georgia Institute of Technology
College of Computing
January 29, 2015 Bhakta

CS 3510 **Name:** ________________________________

Test 1 **Id:** ________________________________

Problem	Points	Score
1	15	
2	15	
3	10	
4	30	
5	30	
Bonus Points	5	
Total	100	

1. List the following functions in increasing order according to their asymptotic growth rate. There is no need to justify your answers. (Hint: First simplify the more complicated expressions into more useful forms).

$$n, \quad \log(n^{100}), \; 2^{\log_4(4n)}, \quad n^4, \quad 7^n, \quad 7^{n\log_3 n}, \quad 7^{\log_3 n}, \quad n^2, 2^n$$

Solution: First simplify

$$2^{\log_4(4n)} = 2^{\frac{\log_2(4n)}{\log_2(4)}} = (2^{\log_2(4n)})^{\frac{1}{2}} = (4n)^{\frac{1}{2}} = 2\sqrt{n}$$

$$7^{\log_3(n)} = 7^{\log_7(n)\cdot\log_3(7)} = n^{\log_3(7)}$$

and

$$\log(n^{100}) = 100\log(n)$$

Note that $1 = \log_3(3) < \log_3(7) < \log_3(9) = 2$.
So putting things in order, first come all logarithms: $100\log(n)$,
then all polynomials in order of exponent $2\sqrt{n}, n, n^{\log_3(7)}, n^2, n^4$,
then all linear exponentials in order of base $2^n, 7^n$,
then all faster than linear exponentials $7^{n\log n}$.
You may write out your final answer in terms of the simplified functions OR the original, but the answer is:

$$\log(n^{100}), 2\sqrt{n}, n, n^{\log_3(7)}, n^2, n^4, 2^n, 7^n, 7^{n\log n}$$

.

2. Say you are designing a divide and conquer algorithm for a problem, and you come up with three possible solutions.

   (a) if you solve 2 subproblems of size $n/3$, then the cost for combining the solutions is $3n^2 + n$.

   Solution:
   $$T(n) = 2T(\frac{n}{3}) + O(n^2)$$
   $$a = 2, b = 3, d = 2$$
   $$\text{Therefore,} \quad a < b^d, d > \log_b a$$
   $$\implies T(n) = O(n^2)$$

   (b) if you solve 8 subproblems of size $n/4$, then the cost for combining the solutions is 100;

   Solution:
   $$T(n) = 8T(\frac{n}{4}) + O(1)$$
   $$a = 8, b = 4, d = 1$$
   $$\text{Therefore,} \quad a > b^d, d < \log_b a$$
   $$\implies T(n) = O(n^{\log_4(8)}) = O(n^{3/2}))$$

   (c) If you solve 3 subproblems of size $n/3$, then the cost of combining the solutions of the subproblems to obtain a solution for the original problem is $32n + 20$;

   Solution:
   $$T(n) = 3T(\frac{n}{3}) + O(n)$$
   $$a = 3, b = 3, d = 1$$
   $$\text{Therefore,} \quad a = b^d, d = \log_b a$$
   $$\implies T(n) = O(n \log n)$$

   Which divide and conquer solution do you prefer in the long run and why? Your answer should include figuring out the asymptotic runtime of each option. You may use the Master Theorem, where applicable, but show your work.

   Solution: Option c), with a runtime of $O(n \log n)$, is the most aspymtotically efficient.

3. You are given a connected graph labeled with weights on each vertex. You are trying to find the vertex with largest weight. You start at some start vertex v, and then repeatedly move to neighbor, always choosing the largest possible. You repeat this until you can no longer find a node of higher weight. Show that this can be suboptimal by giving an input where it fails.

   Solution: Consider a list of four vertices, all connected in a line. The values at each vertex are, in order, $[100, 2, 1, 3]$. Beginning at vertex 1, you would then proceed to 3 and stop, completely missing the 100.

4. If you were given a sorted array and asked to find the median. you could do this in $O(1)$ time. Instead, you are given **two** sorted arrays, each of size n. We want an algorithm to find the median of the **combined** set of elements. Assume the elements are unique.

   (a) Give an $O(n)$ algorithm for this problem. Your solution to this part doesn't have to use divide-and-conquer. Briefly explain your algorithm in words and justify its running time.

   Solution: Option 1:

   Merge the two arrays together in $O(n + n) = O(n)$ time. Then select the median element of this sorted array in $O(1)$ time.

   Option 2:

   Append the two arrays in $O(1)$ time (Or even O(n) if needed) into array A. Run the selection algorithm which we discussed in class, $Select(A, len(A)/2)$, which takes $O(n)$ time.

   (b) Design a more efficient *divide-and-conquer* algorithm for finding the median. Explain your algorithm in words, describe why your algorithm is correct, and justify its running time (this should include stating the recurrence relation and solving it). (Hint: There are at least two different divide and conquer solutions. In the first one, begin by comparing the medians of both arrays. In the second one, look at the median of the first array, and figure out how to decide if it is bigger or smaller than the true median in $O(1)$ time. Then use that idea to check the entire first array for the median. If you can't find it, try the second array.)

Solution: Solution 1: We give the following recursive dynamic programming algorithm.

Input: Arrays A, B of size n.

Compare the medians of A and B.

If the median of A is bigger than the median of B, then everything on the right half of A (larger than the median), is bigger than at least the first half of A AND the first half of B.) Thus everything to the right of the median of A can't be the median. Similarly, everything to the left of the median of B can't be the median and can be discarded. So we can throw out the right half of A and the left half of B, and recurse on the remaining two arrays, which are about half the size of the original.

If the median of A is smaller than the median of B, then do the opposite, throw out the left half of A and the right half of B.

The recurrence relation is $T(n) = T(n/2) + O(1)$, for a $O(\log n)$ running time.

Solution 2:

Start at the median of A. At any position i of A, you are larger than i elements of A, and smaller than $n - i - 1$ elements of A. if that element was the true median, it would have to be larger than $n - i - 1$ elements of B, and smaller than i elements of B. In otherwords, this element should be smaller than the element at position $n - i$, but larger than the element at position $n - 1 - 1$. If it satisfies both of these, then it is the median. If it is larger than both of these, it is too big to be the median: the median must be smaller. If it is smaller than both, it must similarly be too small. Therefore, we can perform a binary search like algorithm:

Starting with array A, search for a median using the above check to decide if we have found the median, or should search in the left or right halves of A. If we never find a median in A, then check B. The above check takes $O(1)$ time, just like a binary search check. Thus the recurrence is $T(n) = T(n/2) + O(1)$, for a $O(\log n)$ running time.

5. Suppose you have k *sorted* arrays, each with n elements, and you want to merge them into a single sorted array of kn elements.

   (a) Here's one strategy: Using the merge procedure described in class, merge the first two arrays, then merge the third with the result, then merge the fourth with that result, and so on. What is the time complexity of this algorithm, in terms of k and n? Here k and n are both parameters, and neither are constants. You may give your answer in Big-O notation.

   Solution: Recall that merging arrays of size m and n takes $\theta(m+n)$ time. We can say that the runtime is bounded by $c(m+n)$ from some constant c. The first operation will take at most $c(n+n)$. The next will take $c(2n+n)$ steps, then next $c(3n+n)$ steps, and so on. The total runtime will be $\sum_{i=1}^{k} c(i+1)n = n\sum_{i=1}^{k}(i+1) = O(k^2n)$.

   (b) Give a more efficient solution to this problem, using *divide-and-conquer*. Explain your algorithm in words, describe breifly why your algorithm is correct, and find its running time. You may give a strategy that is not based on divide and conquer, provided it does at least as well as the divide and conquer strategy in terms of run time. Your solution must be faster asmyptotically than the approach in part a.

   Solution: Instead of merging the arrays one at a time, we will instead merge pairs of them into $k/2$ arrays of size $2n$, then repeat until we have $k/4$ arrays of size $4n$, etc. until we have merged into one large array. This is the bottom-up view of our algorithm.

   The top-down view of the algorithm is: Given a list of k arrays of size n, split the list of arrays into two halves, recursively merge the first $k/2$ arrays into one array of size $kn/2$, and merge the secong half into another array of size $kn/2$, and the merge these two arrays in $O(kn/2 + kn/2) = O(kn)$ time, using our merge procedure. This top-down view will give us the recurrence equation $T(k) = 2T(k/2) + O(kn)$. By drawing a tree, or by realizing that you want to find the solution to $T(k) = 2T(k/2) + O(k)$ multiplied by $O(n)$, you will arrive at a $O(kn\log(k))$ runtime.

   Alternatively, you could try to modify the merge algorithm to handle k arrays instead of two. Done naively, this would still result in a $O(k^2n)$ algorithm, since you get an extra factor of k by needing to find the smallest element among the pointers in each of the k arrays at each step. However, by using binary search/insertion or a heap data structure to maintain a *priority queue*, you can more efficiently find these minimums, and acheive a $O(kn\log k)$ runtime, which would be an acceptable solution.

6. [**BONUS POINTS**] You are a preschool teacher in charge of n kids, but when your back was turned, the *bad kids* stole the cookies from the cookie jar! You can, one at a time, ask a person x if another person y stole a cookie. You know that the *good kids* in your class will tell the truth if you ask them about the other kids, but the *bad kids*, who stole the cookies, might lie. The problem is, you don't know which of the kids are good or bad! For example, if you ask Raj, who is a good kid, whether or not David stole a cookie, he would respond truthfully. But Prateek, who is a bad kid, could say whatever he wanted.

   (a) If you knew that more than half of the kids are good, give an $O(n \log n)$ divide and conquer algorithm to find out which kids are good and which are bad, with proof of correctness and analysis of running time. (Hint: First make sure you can find an $O(n^2)$ brute force algorithm).

   Solution:
   Adapt the "more than half" algorithm from homework. To perform the equivalent of an "equality" test, we do the following. Given two kids, we ask each kid whether the other stole a cookie. If both reply that the other **did not** steal a cookie, then we know that either **both are good** kids *OR* **both are bad**. When this happens, we say that the children have "matched." In any other situation, we will say that the children have "mismatched." We can never mismatch two good children, and if we mismatch two bad children, this doesn't hurt our "more than half" algorithm - it will only help us ignore the bad kids who are not part of the majority.

   We can now run the "more than half" algorithm out of the box, returning at each step an example child from what we think is the majority group. This takes $O(n \log n)$ time. If there are more good kids than bad, this should result in a good kid, and we can ask them about the entire class in additional $O(n)$ time (Or check who they matched with in the last step of the recursion).

   (b) Prove that if more than half of the kids are bad, then there is no way for you to find out exactly which of the kids are good or bad. In other words, give a strategy that the bad kids can use to fool you.

   Solution: Say only $k < n/2$ of the kids are good, and say that you had all of the $O(n^2)$ peices of information about what each kid says about all of the other kids. The good kids will respond truthfully that they are the only good kids and everyone else is bad. If some k of the bad kids collaborated and similarly acted *as if* they were the only good kids and everyone else was a bad kid, there would be no way for you to distinguish between the real k good kids and the fake k "good" kids.

This page is meant for use as scratch paper.

This page is meant for use as scratch paper.