COSC 311: ALGORITHMS
HW2: DIVIDE-AND-CONQUER
Solutions

Problem 1: Using the Master Theorem.

For each of the following recurrences, check whether the Master Theorem applies. If it does, give the asymptotic class of $T(n)$. If not, explain why not.

(a) $T(n) = 9T(n/3) + 4n^2$

Solution: We have $a = 9$ and $b = 3$, so $n^{\log_b a} = n^{\log_3 9} = n^2$. We also have $f(n) = 4n^2$. Comparing the two, we see that $f(n) = 4n^2 = \Theta(n^2)$, so Case 2 applies and $T(n) = \Theta(n^2 \lg n)$.

(b) $T(n) = 8T(n/2) + \frac{2n^3}{\lg n}$

Solution: We have $a = 8$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 8} = n^3$. We also have $f(n) = \frac{2n^3}{\lg n}$. We might be tempted to apply Case 1 since $f(n) = O(n^3)$, but we have to check if there's a polynomial separation between $f(n)$ and $n^{\log_b a}$. That is, in order to apply Case 1 we need $f(n)\frac{2n^3}{\lg n} = O(n^{3-\epsilon})$ for some $\epsilon > 0$. That is, we need $\frac{1}{\lg n} = O(n^{-\epsilon}) = O(\frac{1}{n^\epsilon})$. Equivalently, we need $\lg n = \Omega(n^\epsilon)$, which cannot be true for any $\epsilon > 0$. So Case 1 does not apply. Nor does Case 2 since $\frac{2n^3}{\lg n} \notin \Theta(n^3)$. We fall in between Cases 1 and 2, so we cannot solve this recurrence using the Master Theorem.

(c) $T(n) = 2T(n/2) + n\sqrt{n}$

Solution: We have $a = 2$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 2} = n$. We also have $f(n) = n\sqrt{n} = n^{3/2}$. Comparing the two, we see that $f(n) = n^{3/2} = \Omega(n^{1+\epsilon})$ for $\epsilon = 1/2$ (for example; other choices of ϵ also work). So we may be in Case 3. We still have to check the regularity condition. Here we have $af(n/b) = 2f(n/2) = 2 \cdot (\frac{n}{2}\sqrt{\frac{n}{2}}) = \frac{1}{\sqrt{2}}n\sqrt{n} \leq cn\sqrt{n}$ for $c = \frac{1}{\sqrt{2}} < 1$. The regularity condition is satisfied, so Case 3 of applies and $T(n) = \Theta(f(n)) = \Theta(n\sqrt{n})$.

(d) $T(n) = 3T(n/3) + f(n)$ where

$$f(n) = \begin{cases} n^2 & \text{if } n \text{ is a power of 9} \\ 6n^2 & \text{otherwise} \end{cases}$$

Solution: We have $a = 3$ and $b = 3$, so $n^{\log_b a} = n^{\log_3 3} = n$. We also have $f(n)$ as defined above. In both cases, we have that $f(n) = \Omega(n^{1+\epsilon})$ for $\epsilon = 1$, so Case 3 may apply. We need the regularity condition to hold for all sufficiently large values of n, so let's first check the case where n is a large power of 9. Here we have $af(n/b) = 3f(n/3) = 3 \cdot 6(\frac{n}{3})^2 = 2n^2 \not\leq cn^2$ for any $c < 1$ (in the second equality we use the "otherwise" case of the recurrence since n was a power of 9, so $n/3$ is not). The regularity condition does not hold for n's that are powers of 9, so it does not hold for all sufficiently large n. Hence we cannot apply the Master Theorem to solve this recurrence.

2) Probing binary trees.

Consider an n-node complete binary tree (by "complete," we mean that every level is full—so $n = 2^d - 1$ for some positive integer d). Each node of the tree x stores some value v_x. You can assume that all of the values are distinct, i.e., $v_x \neq v_y$ for all $x \neq y$. A node x is considered a *local minimum* if its value v_x is less than the value v_y for all nodes y that are connected to x by an edge (i.e., v_x is smaller than the values of x's parent and children if they exist).

You can look up the value of a node x by *probing* the node. Assume that probing is a constant-time operation.

(a) Describe an algorithm that finds a local minimum in time $O(\lg n)$. You may write pseudocode or clearly state the steps of your algorithm; your goal is for your description to be precise and unambiguous.

Solution: Let's start by thinking about the root of the tree. The root is a local minimum if its value is smaller than both of its children. If not, we have two cases: either the root is bigger than its left child, or it's bigger than its right child (or both). If the root is bigger than its left child, then there's a chance that the left child could be a local minimum, so we'll check it next. Since we already know it's smaller than the root, we only have to check if it's smaller than both its children. If so, it's a local minimum and we're done. If not, it must be bigger than one of its children. If it's bigger than the left child, then there's a chance that the left child could be a local minimum, so we'll check it next...and so on. This is starting to feel like a recursive algorithm. Here's some pseudocode:

```
1 FindLocalMin(root)
2     if root.left == NULL
3         return root
4     if root.value < root.left.value AND root.value < root.right.value
5         return root
6     if root.value > root.left.value
7         return FindLocalMin(root.left)
8     else return FindLocalMin(root.right)
```

Lines 2-3 are a base case that says that if we reached a leaf, we have found a local minimum (note that we only check if one child is null because we have assumed the tree is complete). Lines 4-5 handle the case in which the root is smaller than both its children; in this case we return the root. Lines 6-7 handle the case in which the root is bigger than its left child; in this case we recur on the left subtree. Finally, line 8 handles the case in which the root is bigger than its right child and we recur on the right subtree.

(b) Explain why your algorithm is correct. You may want to use induction or some other form of reasoning. Your goal here is to convince a classmate that your algorithm does the right thing.

Solution: Here's an inductive proof that the algorithm is correct. Other less formal approaches are also possible, including the intuition given in part (a) about how we came up with this algorithm.

Proof. Our proof will be by induction on the depth of the tree, d.
Base Case: (d=1). There is only one node, namely the root itself, and it must be a local minimum. The base case of our algorithm correctly returns the root (lines 2-3).

Inductive Hypothesis: For some $d \geq 1$, our FindLocalMin algorithm correctly returns a local minimum.

Inductive Step: Assume that the inductive hypothesis holds for d. We will show that for a tree of depth $d+1$, FindLocalMin correctly returns a local minimum.

When we call FindLocalMin on the root, there are three cases: (1) the root's value is smaller than that of both of its children; (2) the root's value is bigger than that of its left child; (3) the root's value is bigger than that of its right child and smaller than that of its left child. We will show that in each of these cases, FindLocalMin produces a correct result.

- **Case 1 (root is smaller than both children):** FindLocalMin returns the root in this case (lines 4-5), which is correct because the root is a local minimum by definition (it is smaller than both children and it has no parent).
- **Case 2 (root is bigger than left child):** Then we recursively call FindLocalMin on the root's left child (lines 6-7). The subtree rooted at the left child is itself a complete binary tree with depth d. By our inductive hypothesis, the recursive call to FindLocalMin correctly returns a local minimum in this subtree. If the returned node is some node x that is not the root of the subtree, then x must also be a local minimum of the entire tree, since all of the nodes connected to x are also within the subtree. If x is instead the root of the subtree (i.e., x is the left child), then we know that x is smaller than both of its children (by the inductive hypothesis and the definition of a local minimum). Since we are in the case in which the root is bigger than its left child, we know that x is smaller than all nodes connected to it, and so it is a local minimum. Hence FindLocalMin returns a correct result.
- **Case 3 (root is bigger than right child and smaller than left child):** The same argument as in Case 2 applies. Again, FindLocalMin returns a correct result.

In all three cases, FindLocalMin correctly finds a local minimum in the tree. Hence FindLocalMin is correct. □

(c) Write a recurrence for the runtime of your algorithm, $T(n)$, and solve your recurrence using the Master Theorem.

Solution: Let $T(n)$ be the time it takes to run FindLocalMin on a tree with n nodes. Our FindLocalMin algorithm makes up to four constant-time comparisons to check which case we fall in, and then makes one recursive call on a subtree. The subtree has depth $d-1$, so it has $\frac{n-1}{2}$ nodes. We'll call this a subproblem of size $n/2$; the additional -1 won't make a difference asymptotically. So our recurrence is:

$$T(n) = T(n/2) + O(1).$$

To apply the Master Theorem, we have $a = 1$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. We also have $f(n) = O(1)$, meaning that the time to combine our subproblems is some constant-time function. Comparing $f(n)$ to $n^{\log_b a}$, we see that our constant-time combine function is $\Theta(1)$, so we fall into Case 2 of the Master Theorem and conclude that $T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$.

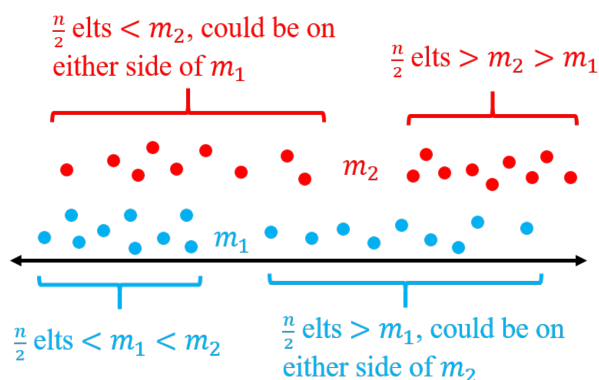
3) Database queries.
You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You want to determine the median of this set of $2n$ values, which we will define to be the nth smallest value.

The only way you can access these values is through queries to the database. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

(a) Describe an algorithm that finds the median value using at most $O(\log n)$ queries. You may write pseudocode or clearly state the steps of your algorithm; your goal is for your description to be precise and unambiguous.

Solution: We start with two databases, each with n elements. We are trying to find the nth smallest element out of all $2n$ elements. We will assume that n is a power of 2, and we will label the elements starting at 1 (so if we query a database for element 1, the result will be the smallest element stored in that database).

Suppose we first query each database for its median—that is, we query for element $n/2$. Let m_1 be the median from database 1 and let m_2 be the median from database 2. Suppose, without loss of generality, that $m_1 < m_2$. We know that $n/2$ elements in database 1 are less than or equal to m_1, and therefore must also be less than m_2. We don't know how the remaining $n/2$ elements in database 1 are ordered with respect to m_2. Similarly, we know that $n/2$ elements in database 2 are greater than m_2 and therefore also greater than m_1, but we don't know how the remaining $n/2$ elements in database 2 are ordered with respect to m_1. Here's a picture of what we know:



We can say for sure that the median of all $2n$ elements lies somewhere in either the largest $n/2$ elements in database 1, or the smallest $n/2$ elements in database 2. In fact, we can say that the median of all $2n$ elements is the median of the largest $n/2$ elements in database 1 and the smallest $n/2$ elements in database 2, since we have removed from consideration the same number of elements in both databases. So we can view these subsets as two "smaller databases," and recursively find the median of these smaller databases.

Here's some pseudocode:

```
1 FindMedian(D1, d_1^ℓ, d_1^h, D2, d_2^ℓ, d_2^h)
2     if d_1^ℓ == d_1^h
3         m1 = query(D1, 1)
4         m2 = query(D2, 1)
5         return min(m1, m2)
6     m1 = query(D1,(d_1^ℓ + d_1^h − 1)/2)
7     m2 = query(D2,(d_2^ℓ + d_2^h − 1)/2)
8     if m1 < m2
9         return FindMedian(D1, m1+1, d_1^h, D2, d_2^ℓ, m2)
10     else
11         return FindMedian(D1, d_1^ℓ, m1, D2, m2+1, d_2^h)
```

If we imagine the elements of each database as being stored in an array (indexed starting at 1) in increasing order, d_i^ℓ and d_i^h represent the subarray to which we are restricting our attention, and `query(D1,j)` looks up the `j`th smallest element in the database (i.e., the element in position j in our imagined array). Our initial call is to `FindMedian(D1, 1, n, D2, 1, n)`.

(b) Explain why your algorithm is correct. You may want to use induction or some other form of reasoning. Your goal here is to convince a classmate that your algorithm does the right thing.

Solution: We'll give an inductive proof using induction on n, assuming that n is a power of 2. To clarify our argument, we will imagine each database as being an array indexed from 1 to n, where the array is sorted so that the element at position 1 is the smallest value in the array and the element at position n is the largest value in the array.

Proof. Base case: ($n = 1$). There is a single element in each database, so by definition the median is the smaller of these two elements. The base case of our algorithm queries both databases for their single element, and correctly returns the smaller.

Inductive hypothesis: For some $n \geq 1$, our algorithm returns the correct median of all elements in both databases.

Inductive step: Assume the inductive hypothesis holds when there are n total elements (i.e., $n/2$ elements in each database). We will show that our algorithm is correct when there are $2n$ total elements (i.e., n elements in each database).

We first find the median value in each database, `m1` and `m2`. If `m1 < m2`, then we know that the

elements in positions `1,...,m1-1` of database 1 are less than both medians, and the elements in positions `m2+1,...,n` of database 2 are greater than both medians. The true median of all $2n$ elements must lie either in positions `m1+1,...,n` of database 1 or in positions `1,...,m2-1` of database 2. Why? We can see this by contradiction. Suppose, without loss of generality, that the median is some element $x < \texttt{m1}$. Then there are at least $n/2+1$ elements in database 1 that are greater than x (the $n/2$ elements greater than `m1`, plus `m1` itself). There are also at least $n/2$ elements in database 2 that are greater than x (the $n/2$ elements, including `m2`, that are greater than `m1`). So there are at least $n+1$ elements greater than x, which contradicts our assumption that x was the median.

Furthermore, the true median is the median of the elements in these two subarrays, since we have discarded the same number of "too small" and "too big" elements. Each of the subarrays is itself a database consisting of $n/2$ elements; the two subarrays collectively form a pair of databases with n total elements. By our inductive hypothesis, our recursive call to

$$\texttt{FindMedian(D1, m1+1, } d_1^h\texttt{, D2, } d_2^\ell\texttt{, m2)}$$

correctly returns the median of the n elements in these subarrays. Hence our algorithm is correct in this case.

A similar argument holds when $\texttt{m1} \geq \texttt{m2}$. Hence our algorithm produces the correct result on a pair of databases with $2n$ total elements. □

(c) Write a recurrence for the runtime of your algorithm, $T(n)$, and solve your recurrence using the Master Theorem.

Solution: We start off with two databases, each with n elements, for a total of $2n$ elements. Our algorithm makes one recursive call on a problem of size n (we cut both arrays in half). Finally, we make two (constant-time) queries, a comparison, and some arithmetic: this is $O(1)$. Our base case make a comparison, does two queries, and returns a result: this is also $O(1)$. Putting it all together, we get:

$$T(2n) = \begin{cases} T(n) + c_1 & n > 1 \\ c_2 & n = 1 \end{cases}$$

Writing this as $T(n)$, we have:

$$T(n) = \begin{cases} T(n/2) + c_1 & n > 2 \\ c_2 & n = 2 \end{cases}$$

This is in fact the same recurrence we had in problem 5, so the answer is the same: we are in Case 2 of the Master Theorem, and $T(n) = \Theta(\lg n)$.