COSC 311: ALGORITHMS
HW2: RECURRENCES AND PROOFS
Solutions

1 Practice with Recurrences

1) An erroneous inductive proof.
Consider the following recurrence:

$$T(n) = \begin{cases} 3T(n/3) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

where n is a power of 3.

(a) Here is an *incorrect* theorem and proof about this recurrence:

Theorem 1. $T(n) \in O(n)$.

Proof. Our proof will be by induction.
Base case: $n = 3$. Then we have:

$$T(3) = 3T(1) + 3 = 3 \cdot 1 + 3 = 6 \leq cn$$

for $c = 2$.
Inductive hypothesis: For some $n \geq 3$, $T(n) \leq cn$.
Inductive step: Assume the inductive hypothesis holds. We will show that $T(3n) \leq 3cn$. We have:

$$\begin{aligned} T(3n) &= 3T(3n/3) + 3n && (1) \\ &= 3T(n) + 3n && (2) \\ &\leq 3cn + 3n && (3) \\ &= O(n). && (4) \end{aligned}$$

Hence $T(n) \in O(n)$. □

Where is the mistake in the proof?

Solution: The problem is in the last line of the inductive step. It is not enough to show that $T(n)$ is less than a function that happens to be $O(n)$, we need to show that $T(n)$ is less than the *specific* function $3cn$, for our specific choice of $c = 2$. Based on the third line of our inductive step and our choice of c, we have $T(n) \leq 3cn + 3n = 6n + 3n = 9n$. This will never be less than $3cn = 6n$, so our proof is broken.

(b) What is the correct asymptotic class for this recurrence? Prove by induction that your guess is correct.

Solution: $T(n) = O(n \ln n)$. We can guess this intuitively by recognizing that our recurrence follows a very similar form to the mergesort recurrence. We can also apply the Master Theorem, where $a = 3$, $b = 3$, and $f(n) = n$, so $n^{\log_b a} = n$, $f(n) = n = O(n^{\log_b a})$, and so case 2 of the Master Theorem applies and $T(n) = \Theta(n \lg n)$. We will prove this claim by induction.

Theorem 2. *Let $n_0 = 3$ and $c = 2$. Then for all $n > n_0$, $T(n) \leq cn \log_3 n$*

Proof. Our proof will be by induction on n.
Base case: (n=3). Then $T(n) = T(3) = 3T(3/3) + 3 = 3T(1) + 3 = 3 + 3 = 6 = cn \log_3 n$.
Inductive hypothesis: For some $n \geq n_0 = 3$, $T(n) \leq cn \log_3 n$.
Inductive step: Assume that the inductive hypothesis holds for n. We will show that $T(3n) \leq 3cn \log_3(3n)$. We have:

$$\begin{aligned} T(3n) &= 3T(n) + 3n \\ &\leq 3cn \log_3 n + 3n \\ &= 3cn(\log_3(3n) - \log_3 3) + 3n \\ &= 3cn \log_3(3n) - 3cn + 3n \\ &\leq 3cn \log_3(3n) \end{aligned}$$

Hence $T(n) = O(n \lg n)$. □

2) Does the Master Theorem apply?
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) $T(n) = 27T(n/3) + n^2$
Solution: We have $a = 27$ and $b = 3$, so $n^{\log_b a} = n^{\log_3 27} = n^3$. So $f(n) = n^2 = O(n^{\log_b a - \epsilon})$ for $\epsilon = 1$. So Case 1 of the Master Theorem applies, and $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$.

(b) $T(n) = 2T(n/4) + n$
Solution: We have $a = 2$ and $b = 4$, so $n^{\log_b a} = n^{\log_4 2} = n^{1/2}$. So $f(n) = n = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon = 1/2$. So Case 3 of the Master Theorem might apply: we first need to check the regularity condition. We have $af(n/b) = 2\frac{n}{4} \leq cn$ for $c = 1/2$, so the regularity condition holds, Case 3 applies, and $T(n) = \Theta(f(n)) = \Theta(n)$.

(c) $T(n) = 4T(n/2) + n \sin n$
Solution: We have $a = 4$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 4} = n^2$. We need to compare $f(n) = n \sin n$ to n^2. The function $\sin n$ is always between -1 and 1, so we can say $n \sin n \leq n$ for all n. However, our big-O definition says that $f(n) = O(g(n))$ if there are constants c and n_0 such that for all $n \leq n_0$, $0 \leq f(n) \leq cg(n)$. It turns out that this is one of those cases where the $0 \leq f(n)$ part of the inequality is important: $f(n) = n \sin n$ is not asymptotically positive, so we can't apply the

Master Theorem.

As a "sanity check," think about what it would mean in a divide-and-conquer algorithm for the cost to split the problem into subproblems and then combine the results to be $n \sin n$. This function is sometimes negative, so we'd be saying that it takes *negative* time to combine the results of our subproblems. This doesn't make very much sense!

(d) $T(n) = 8T(n/2) + 4n^3$
Solution: We have $a = 8$ and $b = 2$, so $n^{\log_b a} = n^{\log_2 8} = n^3$. So $f(n) = 4n^3 = \Theta(n^{\log_b a})$. So Case 2 of the Master Theorem applies, and $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^3 \lg n)$.

(e) $T(n) = 3T(n/3) + \frac{n}{\lg n}$
Solution: We have $a = 3$ and $b = 3$, so $n^{\log_b a} = n^{\log_3 3} = n$. We need to compare $f(n) = \frac{n}{\lg n}$ to n. We can see immediately that $\frac{n}{\lg n}$ grows more slowly than n, so our best candidate is Case 1. In order for Case 1 to apply, we need $f(n) = O(n^{1-\epsilon})$ for some ϵ; this means we would need to find a c and n_0 such that for all $n \geq n_0$,

$$0 \leq \frac{n}{\lg n} \leq cn^{1-\epsilon}$$
$$0 \leq \frac{1}{\lg n} \leq cn^{-\epsilon}$$
$$0 \leq n^{\epsilon} \leq c \lg n$$

which isn't true for any ϵ. So the Master Theorem doesn't apply.

2 Selection

In the *selection* problem, our input is an array `a` of ints (in no particular order). Our goal is to find the kth smallest element in the array (where $k = 0$ is the smallest element). For example, if our array is `a = <3 7 9 1 6 8>` and $k = 3$, the algorithm should give 7 as the output since 7 is the 3rd smallest element (again, where $k = 0$ is the smallest element).

Clearly, one way to solve the selection problem is to sort the array and then return `a[k]`. But as we have seen, sorting requires $\Omega(n \lg n)$ time and we would like to solve the selection problem faster than that.

3) Modified quicksort.
Design an algorithm called quickselect that uses a modified version of (deterministic) quicksort to solve the selection problem in average time $\Theta(n)$.

(a) Write pseudocode for your algorithm.

Solution:

```
QuickSelect(int[] a, int lo, int hi, int k)
    if lo == hi
        return a[lo]
    part → partition(a, lo, hi)
    q = part − lo + 1
    if q == k, return a[part]
    else if q < k
        return QuickSelect(a, lo, part−1, k)
    else
        return QuickSelect(a, part+1, hi, k−q)
```

(b) Assume your choice of pivot always produces a partition that has pn elements to the left of the pivot (i.e., smaller than the pivot) and $(1-p)n$ elements to the right of the pivot (i.e., bigger than the pivot). Write a recurrence for the runtime of your algorithm, $T(n)$ (assume that $p \geq 1/2$).

Solution: Our algorithm takes time $O(n)$ to do the partitioning (this is the exact same partition method we used for quicksort). We're then allowed to assume that the partition method splits our array into one piece of size pn and one piece of size $(1-p)n$. We don't know which side our kth element is on, but since $p \geq 1/2$ we know that our recursive call is on an array of size *at most* pn. So we can write:

$$\begin{aligned} T(n) &\leq T(\lfloor pn \rfloor) + c_1 n && n > 1 \\ T(n) &= c_2 && n = 1 \end{aligned}$$

Note that we use an inequality to express our recurrence; this is fine because in part (c) we are looking fo a big-O result, which is an upper bound. If we wanted an Ω or Θ result, we could also lower bound $T(n)$ by noting that our recursive call is always on an array of size *at least* $(1-p)n$.

Why did I use a floor function? In order to guarantee that we'll recur on a subproblem of integer size, we need to take either a floor or a ceiling. Suppose $n = 5$ and $p = 9/10$. Then $pn = 45/10 = 4.5$. If we take a ceiling, our subproblem will still be of size 5, which is no good! So we'll take a floor instead, which guarantees that our subproblems always get smaller. This also helps us deal with the issue of choosing a base case: once n gets small enough that $pn > n - 1$, our subproblem will get smaller by 1 on each recursion. Eventually, we'll get down to $n = 1$.

(c) Prove by induction that $T(n) = O(n)$.

Solution:

Theorem 3. *Let* $c = \frac{c_1}{1-p} + c_2$ *and* $n_0 = 2$. *For all* $n \geq n_0$, $0 \leq T(n) \leq cn$.

Proof. We will use strong induction.

Base case: ($n = 2$). Then

$$\begin{aligned} T(n) = T(2) &= T(\lfloor 2p \rfloor) + 2c_1 \\ &= T(1) + 2c_1 \\ &\leq c_2 + 2c_1 \\ &\leq c_2 + \frac{c_1}{1-p} \\ &\leq cn \end{aligned}$$

Inductive hypothesis: For all k such that $1 \leq k < n$, $T(k) \leq ck$.
Inductive step: Assume the inductive hypothesis holds. We will show that $T(n) \leq cn$. We have:

$$\begin{aligned} T(n) &= T(\lfloor pn \rfloor) + c_1 n && (5) \\ &\leq c\lfloor pn \rfloor + c_1 n && (6) \\ &\leq cpn + c_1 n && (7) \\ &= (cp + c_1)n && (8) \\ &= \left(\left(\frac{c_1}{1-p} + c_2\right)p + c_1\right)n && (9) \\ &= \left(\frac{c_1 p + c_1(1-p)}{1-p} + c_2 p\right)n && (10) \\ &= \left(\frac{c_1}{1-p} + c_2 p\right)n && (11) \\ &\leq \left(\frac{c_1}{1-p} + c_2\right)n && (12) \\ &= cn. && (13) \end{aligned}$$

The first inequality is due to the strong inductive hypothesis, and the remaining steps are algebraic. We have shown that $T(n) \leq cn$, where $c = \frac{c_1}{1-p} + c_2$, for all $n \geq n_0 = 2$. Hence $T(n) = O(n)$. □

4) Worst case linear-time selection.
Unfortunately, in the worst case the quickselect algorithm from problem (4) takes time $O(n^2)$ in the worst case. We would like to design a deterministic selection algorithm that takes time $O(n)$ in the worst case to find the kth smallest element. Consider the following selection algorithm:

1. Divide the n elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, and possibly one additional group with $n \mod 5$ elements.

2. For each group, find its median. Do this by insertion sorting the elements in the group and choosing the middle element from the sorted list.

3. Recursively find the median of the $\lceil n/5 \rceil$ medians (if $\lceil n/5 \rceil$ is even, use the smaller of the two median elements). Call this median-of-medians x.

4. Partition the original input array (as in quicksort), using x as the pivot. Let q be the index of x in the partitioned array (so there are $q-1$ elements less than x and $n-q$ elements greater than x).

5. If $k = q$, then x is the kth smallest element and we are done. Otherwise recursively find the kth smallest element on the "small" side if $k < q$, or find the $(k-q)$th smallest element on the "big" side if $k > q$.

(a) For each step of the above algorithm, give the runtime of that step (if the step involves a recursive call, express that step's runtime in terms of the runtime of the smaller subproblem). Put the pieces together to write a recurrence that expresses the runtime of the entire algorithm, $T(n)$.

Solution: First let's write this algorithm as pseudocode to make it clearer what's going on.

```
Select(A[], lo, hi, k)
   if A.length < 140
      insertionSort(A[lo,...,hi])
      return A[k]
   for i → 0 to ⌊n/5⌋ + 1
      insertionSort(A[5i,...,5(i+1)−1])
      set groupMedians[i] to A[5i+2]
   set x to Select(groupMedians, ⌈n/5⌉/2)
   set q to partition(A, lo, hi, x)
   if k==q return A[q]
   else if k < q return Select(A, lo, part−1, k)
   else return Select(A, part+1, hi, k)
```

Let $T(n)$ be the time it takes to run Select on an input array of size n. $T(n)$ consists of the following components:

1. We imagine mentally grouping our array into sub-groups with 5 elements each. This doesn't actually require any computation, so it takes time $O(1)$.

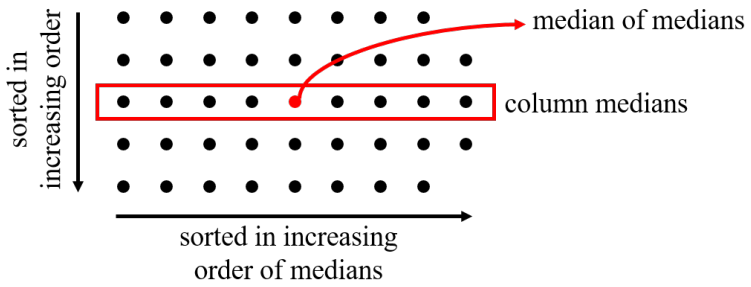
2. Running insertion sort on an array of 5 elements takes constant time (why? because we can count up the total number of operations required to do this. For small enough n, we can say that running an algorithm on a *particular* choice of n takes constant time). We do this for $\lfloor n/5 \rfloor$ groups, so the total time is $O(n)$.

3. Next we make a recursive call to Select. The array we pass to Select in this call has size $\lceil n/5 \rceil$: there is one element for each of the $\lceil n/5 \rceil$ groups. So the time required to run this recursive call is $T(\lceil n/5 \rceil)$.

4. Partitioning our array takes time $O(n)$, the same as it did in quicksort.

5. Finally we make another recursive call to Select. This time we need to do a bit more work to bound the size of the subproblem. Imagine grouping our n elements into $\lceil n/5 \rceil$ columns, with 5 elements in each column (the last column might have fewer than 5 elements, if n is not evenly divisible by 5). We then imagine sorting each column so that the smallest element is

at the top and the biggest element is at the bottom, and the median is in the middle. Finally, we imagine sorting our columns by their median values. We end up with the following situation:



We used the median of medians as our pivot in step 4; the goal was to ensure that a certain fraction of the elements ended up on each side of the partition. We now need to compute that fraction. Since our columns are sorted in increasing order of medians, we know that all medians to the right of our median-of-medians are bigger than the median-of-medians. And since each column is sorted in increasing order, we know that all items below a column median are bigger than it. Hence the items that are below and to the right of the median-of-medians must be bigger than the median-of-medians, so these items must end up on the right side of the pivot after partitioning. We can say that *roughly* a quarter of our elements fall into this case, so in the worst case the larger side of the partition contains at most $3/4$ of the elements.

But we can do a bit better. At least half of our columns have medians that are greater than or equal to the median-of-medians: a total of $\frac{1}{2}\lceil n/5 \rceil$ columns. If we look at an individual column to the right of our median-of-medians, this column contributes at least 3 elements to the right side of the pivot after partitioning (the column median, and the two elements below it. The elements above the column median might also be greater than the median-of-medians, but we don't know this for sure). The two exceptions are the column including the median-of-medians, which only contributes 2 elements, and the last column, which might contribute fewer than 3 elements if the column contains fewer than 5 elements. Putting this together, we have that the number of elements that end up to the right of the pivot is at least:

$$3 \cdot \left(\frac{1}{2} \left\lceil \frac{n}{5} \right\rceil - 2 \right) \geq \frac{3n}{10} - 6.$$

We can make a similar argument to show that there must be at least $3n/10 - 6$ elements to the left of the pivot. So in the worst case, the larger side of the partition contains at most $7n/10 + 6$ elements.

Hence we can say that the last recursive call to Select takes time at most $T(7n/10 + 6)$.

Putting all of this together, we have:

$$T(n) \leq \begin{cases} T(\lceil n/5 \rceil) + T(7n/10) + c_1 n & n \geq 140 \\ c_2 & n < 140 \end{cases}$$

Why 140? It turns out that this is a "nice" value for the base case of our algorithm that makes the inductive proof in part (b) a little easier. Other choices for the base case will also work.

(b) Prove inductively that $T(n) = O(n)$.

Solution: Let $c = 20c_1$ and $n_0 = 140$. We will show that for all $n \geq n_0$, $T(n) \leq cn$. Our proof will be by strong induction on n.
Base case: ($n = 140$). Then $T(n) = T(140) = T(28)+T(98)+140c_1 = 2c_2+140c_1 < 140c = cn$.
Strong inductive hypothesis: For some $n \geq 140$, $T(k) \leq ck$ for all $140 \leq k < n$.
Inductive step: Assume that the inductive hypothesis holds for some $n \geq 140$. We will show that $T(n) \leq cn$. We have:

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{7n}{10} \right\rceil\right) + T\left(\frac{7n}{10}\right) + c_1 n \\ &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7n}{10} + 6\right) + c_1 n \\ &\leq c\left(\frac{n}{5} + 1\right) + c\left(\frac{7n}{10} + 6\right) + c_1 n \\ &= \frac{cn}{5} + c + \frac{7cn}{10} + 6c + c_1 n \\ &= cn \cdot \frac{9}{10} + 7c + c_1 n \\ &= cn \cdot \frac{9}{10} + 7c + \frac{cn}{20} \\ &= c\left(\frac{19}{20} n + 7\right) \\ &\leq cn \end{aligned}$$

Hence $T(n) = O(n)$.

3 Divide and Conquer

5) Probing binary trees.
Consider an n-node complete binary tree, where $n = 2^d - 1$ for some 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.

You can look up the value of a node x by *probing* the node. Assume that probing is a constant-time operation. Your goal in this problem is to come up with an algorithm that finds a local minimum in time $O(\lg n)$.

(a) Give a precise description of your algorithm. You may write pseudocode or state the steps of your algorithm as in problem (5) above.

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:

```
FindLocalMin(root)
   if root.left == NULL
      return root
   if root.value < root.left.value AND root.value < root.right.value
      return root
   if root.value > root.left.value
      return FindLocalMin(root.left)
   else return FindLocalMin(root.right)
```

The first line is a base case that says that if we reached a leaf, we have found a local minimum. The next if statement handles the case in which the root is smaller than both its children; in this case we return the root. The next if statement handles the case in which the root is bigger than its left child; in this case we recur on the left subtree. Finally, in our last case the root is bigger than its right child and we recur on the right subtree.

(b) Prove, using induction, that your algorithm is correct.

Solution:

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.
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, 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. 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, $T(n)$. Use the Master Theorem to show that $T(n) = O(\lg n)$.

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 $T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$.

6) Database queries.

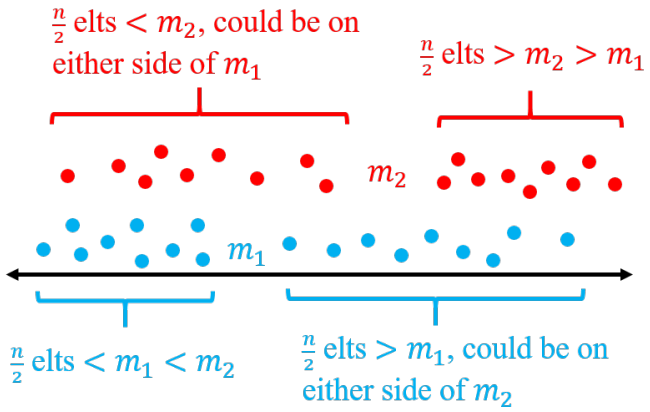
You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values—so there are $2n$ values total—and you may assume that no two values are the same. You want to determine the median of this set of $2n$ values, which we will define to be the 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) Give an algorithm that finds the median value using at most $O(\log n)$ queries.

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 at this point:



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:

```
FindMedian(D1, d1^ℓ, d1^h, D2, d2^ℓ, d2^h)
    if d1^ℓ == d1^h
        m1 = query(D1, 1)
```

```
    m2 = query(D2, 1)
    return min(m1, m2)
m1 = query(D1,(d_1^ℓ + d_1^h − 1)/2)
m2 = query(D2,(d_2^ℓ + d_2^h − 1)/2)
if m1 < m2
    return FindMedian(D1, m1+1, d_1^h, D2, d_2^ℓ, m2)
else
    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) Prove, using induction, that your algorithm is correct.

Solution: Our proof will be by 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 $x <$ `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

`FindMedian(D1, m1+1,` d_1^h`, D2,` d_2^ℓ`, 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 `m1` $\geq$ `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, $T(n)$. Use the Master Theorem to show that $T(n) = O(\log n)$.

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 $T(n) = \Theta(\lg n)$.