CS4102 - ALGORITHMS - FALL 2019

DUE TUESDAY, OCTOBER 1, 2019 AT 11PM VIA COLLAB

H1:ER6QT

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own independently written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff.

Collaborators: zh2yn, zz9ek, zm5du

Sources: Chenghan (TA)

PROBLEM 1 *Solving Recurrences*

Prove a (as tight as possible) O (big-Oh) asymptotic bound on the following recurrences. You may use any base cases you'd like.

1. $T(n) = 3T(\frac{n}{3} - 2) + \frac{n}{2}$

Master Theorem Case 2:

$$
\begin{aligned}
if\ f(n) = \Theta(n^{log_b a}),\ then\ T(n) &= \Theta(n^{log_b a} log n) \\
Let\ a &= 3, \\
b &= 3, \\
f(n) &= \frac{m+6}{2} \\
T(m) &= m^{log_3 3} * log(m) \\
T(n-6) &= (n-6)^{log_3 3} * log(n-6) \\
T(n-6) &= (n-6)^1 * log(n-6) \\
T(n) &= n * log(n) \rightarrow T(n) \in \Theta(n * log(n))
\end{aligned}
$$

2. $T(n) = 7T(\frac{n}{5}) + n \log n$

Master Theorem Case 1:

$$
\begin{aligned}
if\ f(n) = O(n^{log_b a - \epsilon})\ for\ some\ constant\ \epsilon > 0,\ then\ T(n) &= \Theta(n^{log_b a}) \\
Let\ a &= 7 \\
b &= 5 \\
f(n) &= nlogn \\
Let\ \epsilon &> 0 \\
Case\ 1 : f(n) &= O(n^{log_b a - \epsilon}) \\
nlogn &= O(n^{log_5 7 - \epsilon}) \\
nlogn &= O(log_5(7-\epsilon)) * n \\
nlogn &\neq O(n^{1.2}) \\
nlogn &< O(n^{1.2}) \\
T(n) &\in \Theta(n^{log_b a})
\end{aligned}
$$

3. $T(n) = 4T(\frac{n}{2}) + 10n^{5/2}$

Master Theorem Case 3:

$$
\begin{aligned}
if\ f(n) = \Omega(n^{log_b a + epsilon})\ for\ some\ constant\ \epsilon > 0,\ then\ T(n) = \Theta(f(n)) \\
Let\ a &= 4, \\
b &= 2 \\
f(n) &= 10n^{5/2} \\
Let\ \epsilon &= .001 \\
10 * n^{5/2} &= \Omega(n^{log_2 4 + \epsilon}) \\
n^{5/2} &= \Omega(n^{log 2 + \epsilon} n^{5/2}) \geq n^{2+\epsilon} \\
n^{5/2} &\geq n^{2+.0001} \\
n^{5/2} &\geq n^{2.0001}
\end{aligned}
$$

We then proceed to check if $a * f(n/b) \leq c * f(n)$ for a constant $c < 1$ and a large n.

$$
\begin{aligned}
Let\ a &= 4, \\
b &= 2 \\
f(n) &= 10n^{5/2} \\
a * f(n/b) &\leq c * f(n)\ for\ some\ constant\ c \\
4 * (10n/2)^{5/2} &\leq c * 10 * n^{5/2} \\
4 * (n/2)^{5/2} &\leq c * n^{5/2} \\
4 * (n^{2.5})/(2^{2.5}) &\leq c * n^{5/2} \\
4 * (n^{2.5}/2^{2.5}) &\leq c * n^{5/2} \\
.707n^{2.5} &\leq c * n^{5/2}
\end{aligned}
$$

Thus, for any constant $c < 1$ and greater than .707, this holds true. $(1 > c > .707)$

PROBLEM 2 *Flight Planning*

You have been hired to plan the flights for Professor Floryan's brand new passenger air company, "Receding Airlines." Your objective is to provide service to n major cities within North America. The catch is that this airline will only fly you South.

You recognize that in order to enable all your passengers to travel from any city to any other city (to the South) with a single flight requires $\Omega(n^2)$ different routes. Prof. Floryan says that the airline cannot be profitable when supporting so many routes. Another option would be to order the cities in a list (from North to South), and have flights that go from the city at index i, to the city at index $i + 1$. This requires $\Theta(n)$ routes, but would mean that some passengers would require $\Omega(n)$ connections to get to their destination.

1. Devise a compromise set of routes which requires no passenger have more than a single connection (i.e. must take at most two flights), and requires no more than $O(n \log n)$ routes.

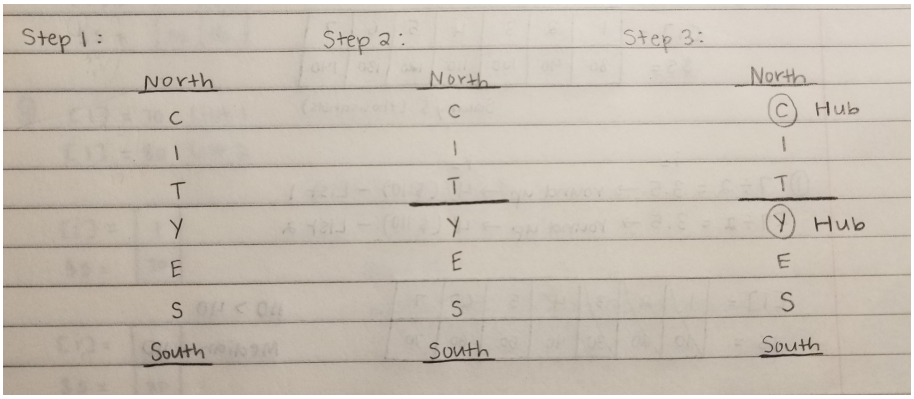

Figure 1: Algorithm for Flight Planning

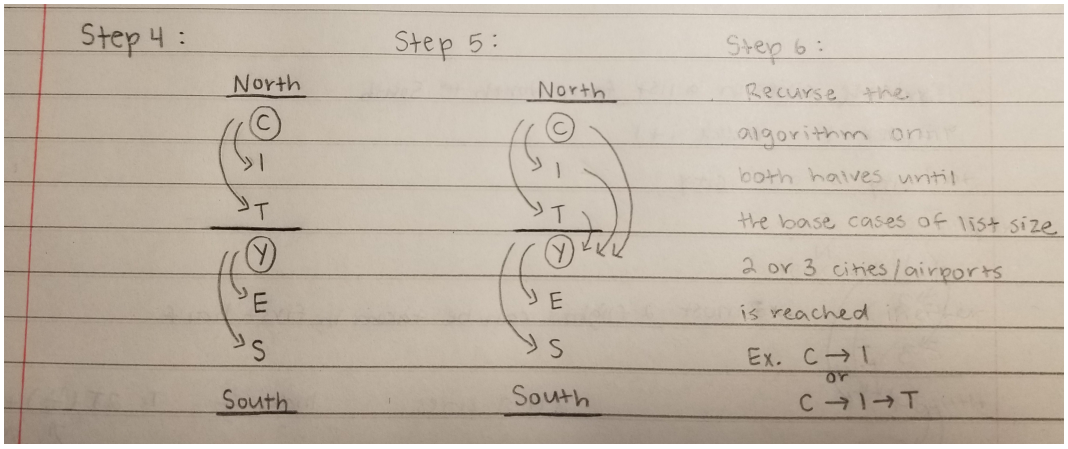

Figure 2: Algorithm for Flight Planning Continued

You start by ordering the cities in a list from North to South as shown in step 1 of Figure 1 above. You then divide that list in half as shown in step 2. If there are an uneven number of elements(cities) in the list, then you can take the ceiling when you divide the list in half. For example, if there are 7 cities, the half would be 3.5, so by taking the ceiling, you would have 4 elements in one half, and 3 elements in the other half. This would take into consideration an odd number of cities in the list. Next, note that the first city in each half is a hub(step 3) that connects to every other city in its half, which is step 4 of Figure 2 right above. As well, every city in the first half is connected to the hub of the city in the second half, as can be seen in step 5. You then recurse this algorithm on both halves of the split list till you reach your bases cases of two or three cities, in which a passenger has a single connection route between cities that take at most two flights, as explained in step 6.

Requirement One: No passenger have more than a single connection (i.e. must take at most two flights)

As mentioned above, the list of cities is split into 2 lists/subproblems, North and South. A passenger can travel from the hub(first airport in each half) to the airports/cities in its half, as explained above. The cities in the first half(North) can all connect to the hub of the second

half(South) as well. So passengers travelling within their half can take at most two flights, requiring only a single connection, and passengers travelling to the other half(South) can go the the hub in the second half(South), and then connect to any other airport in that half, again, taking at most two flights to reach their destination city/airport. In both of these ways to travel, there is only a possibility of taking one or at most two flights to reach the destination city, and therefore, only one single connection is needed. In these halves/subproblems, you are recursing until you reach the base cases of two or three cities/airports, which you can only take one or two flights to travel between. Thus, these two ways to travel prove that the algorithm fulfills the requirement that passengers can only have a single connection, taking at most two flights.

Requirement Two: No more than $O(nlogn)$ routes

Recurrence Equation: $T(n) = 2T(\frac{n}{2}) + f(n)$
There are 2 subproblems $(2T)$
Divide number of cities in list in half $(\frac{n}{2})$
n is the number of elements(cities) in the list
$logn$ is the time complexity when n number of cities are divided into 2 halves

Master Theorem Case 2:

$$
\begin{aligned}
if\ f(n) = \Theta(n^{log_b a}),\ then\ T(n) &= \Theta(n^{log_b a} logn) \\
a &= 2 \\
b &= 2 \\
f(n) &= n \\
If\ f(n) = \Theta(n^{log_2 2}), then\ \Theta(n^{log_2 2} logn) & \\
\Theta(n) &= \Theta(nlogn) \\
T(n) &= \Theta(nlogn) = O(nlogn)
\end{aligned}
$$

2. After a few years, passengers start demanding routes from South to North, and you decide to support new routes from South to North (in addition to supporting routes from North to South). Show that with routes in *both* directions, it is possible to connect all n cities with just $O(n)$ routes such that no passenger needs more than a single connection to get to their destination.

   One possible way to connect all n cities with just $O(n)$ routes such that no passenger needs more than a single connection to get to their destination with routes going in both directions is by making only one airport/city a hub rather than two. This means that one airport/city can connect to all other cities in the list. Therefore, any passenger can travel to not only that airport/city, but also use it as their single connection to get to their destination city. For example, using the cities/airports in the pictures above but modifying it, if I had cities ”C, I, T, Y, E, S,” airport ”C” would be the only hub. So if a passenger wanted to travel from city ”E” to city ”I”, he/she would travel to the hub airport ”C” first, and then travel to their destination city of ”I” from there as city/airport ”C” is connected to all other cities. This way no passenger needs more than a single connection to get to their destination.

PROBLEM 3 *Trench Warfare*

We are currently developing a new board game called “Trench Warfare.” This game works similarly to “Battleship,” except instead of trying to find your opponent’s ships on a two dimensional

board, you're trying to find a trench (i.e. a local minimum) in your opponent's one dimensional board. Each player's board will be a list of n floating point values. To guarantee that a local minimum exists somewhere in each player's list, we will force the first two elements in the list to be (in order) 1 and 0, and the last two elements to be (in order) 0 and 1.

To make progress, you name an index of your opponent's list, and she/he must respond with the value at that index. To win you must correctly identify that a particular index is a local minimum (the ends don't count). An example board is shown in Figure 4. [We will require that all values other than the first and last pairs be unique.]

1	0	-2	-5	-3	3	5	-1	0	1
0	1	2	3	4	5	6	7	8	9

Figure 3: An example board of size $n = 10$. You win if you can identify any one local minimum, in this case both index 3 and index 7 are local minima.

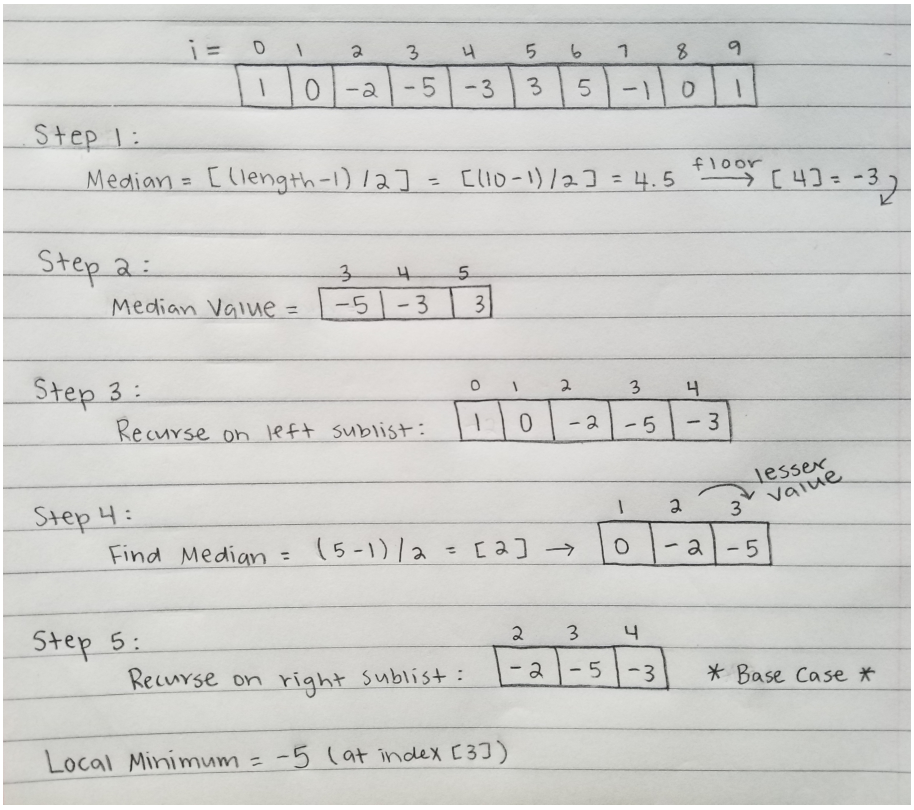

Figure 4: Finding Local Minimum

1. Devise a strategy which will guarantee that you can find a local minimum in your opponent's board using no more than $O(\log n)$ queries, prove your run time and correctness.

Strategy

To guarantee that you find a local minimum in your opponent's board, you would first get the median index in the list in order to find the middle element/value. In order to get the median index and, therefore, the current middle value, you would do [length-1/2]. For an even length of elements you would have to take the floor of the value returned after doing [length-1/2]. Therefore, in the example board list given on the homework, the middle index returned would be 4.5, so you would take the floor of it to return 4 as the index holding the current middle value, as shown in step 1 of Figure 4. You then proceed to look at the values to the left and right of the middle value found, as shown in step 2 of Figure 4. If the value to the left of the middle element's value is less than(smaller) than the middle value, you recurse on the left sublist, making sure to include the middle element, as shown in step 3 of Figure 4. However, if the value to the right of the middle element's value is less than(smaller) than the middle value, you recurse on the right sublist, making sure again to include the middle element found in the beginning of the algorithm, an example of which can be shown in step 4 - 5 of Figure 4. Now if the current middle element's value is less than both its left and right values, then you have found your local minimum. The base case in this strategy is when you have a list of 3 elements, in which the local minimum is the middle value, as can be seen in step 5. This strategy bounds the middle element by higher values to it's left and right so that it is guaranteed we find a local minimum.

Finding Local Minimum

As shown in Figure 4 and explained above, the process of finding the local minimum is by first calculating and finding the middle index in the list and using the value at that index to check if it is a local minimum (if it is less than it's left and right values). If it follows the rules mentioned above for being a local minimum, then the current middle value is the local minimum. If it is not, that means that you have to recurse on either the left sublist or right sublist and include the middle element. If either the left or right element is smaller, you recurse on the sublist of the smaller value's side. If both elements on the left and right are less than the middle value, you look to the value that is lesser than the middle. If it's the left value, you recurse on the left sublist and if it's the right value, you recurse on the right sublist. You keep recursing this algorithm until you reach your base case of 3 elements for which the middle value is less than both of its adjacent values, guaranteeing that a local minimum is found. You would then return the index of that local minimum. Thus, taking this strategy into account, we can prove that we are guaranteed to find a local minimum in our opponent's board.

No more than O(log n) queries

Recurrence Equation: $T(n) = T(\frac{n}{2}) + 2$
We are recursing on either the left sublist or right sublist, splitting the list of elements in half each time: $(\frac{n}{2})$
We are doing a constant comparison of 2 each time we check the values to the left and right of the middle element: 2

Master Theorem Case 2:

$$\begin{aligned} if\ f(n) &= \Theta(n^{log_b a}),\ then\ T(n) = \Theta(n^{log_b a} logn) \\ a &= 1 \\ b &= 2 \\ f(n) &= 2 \\ If\ f(n) &= \Theta(n^{log_2 1}),\ then\ \Theta(n^{log_2 1} logn) \\ \Theta(1) &= \Theta(logn) \\ T(n) &= \Theta(logn) = O(logn) \end{aligned}$$

2. Now show that $\Omega(\log n)$ queries are required by *any* algorithm (in the worst case). To do this, show that there is a way that your opponent could dynamically select values for each query as you ask them, rather than in advance (i.e. cheat, that scoundrel!) in such a way that $\Omega(\log n)$ queries are required by *any* guessing strategy you might use.

   Given that the opponent could cheat and dynamically change values for each query you ask them, there would need to be $\Omega(logn)$ queries to find the local minimum. If we had an algorithm from which we gave them an index value that is not in the middle of the list, they would choose the smaller value to be on the side that would require us to recurse on the sublist that is larger. For example, if the left sublist consisted of $1/3$ of the elements, and the right sublist consisted of $2/3$ of the elements, the opponent would choose values that would require us to recurse on the right sublist. Therefore, we would need to go through more queries to find the local minimum rather than just choosing the middle element. Also, if you start near one of the edges of the list, the opponent will cheat and choose values that will require us go through the whole list until we reach the opposite edge with end pairs of either "1 and 0" or "0 and 1", guaranteeing that a local minimum exists, which would entail log(n) queries. Therefore, the opponent will cheat in a way that makes it impossible to win without going through the whole algorithm, thus making $\Omega(logn)$ queries required by any algorithm (in the worst case).

PROBLEM 4 *Bazinga!*

Theoretical Physicist Sheldon Cooper has decided to give up on String Theory in favor of researching Dark Matter. Unfortunately, his grant-funded position at Caltech is dependent on his continued work in String Theory, so he must search elsewhere. He applies and receives offers from MIT and Harvard. While money is no object to Sheldon, he wants to ensure he's paid fairly and that his offers are at least the median salary among the two schools' Physics departments. Therefore, he hires you to find the median salary across the two departments. Each school mantains a database of all of the salaries for that particular school, but there is no central database.

Each school has given you the ability to access their particular data by executing *queries*. For each query, you provide a particular database with a value k such that $1 \leq k \leq n$, and the database returns to you the k^{th} smallest salary in that school's Physics department.

You may assume that: each school has exactly n physicists (i.e. $2n$ total physicists across both schools), every salary is unique (i.e. no two physicists, regardless of school, have the same salary), and we define the *median* as the n^{th} highest salary across both schools.

1. Design an algorithm that finds the median salary across both schools in $\Theta(\log(n))$ total queries.

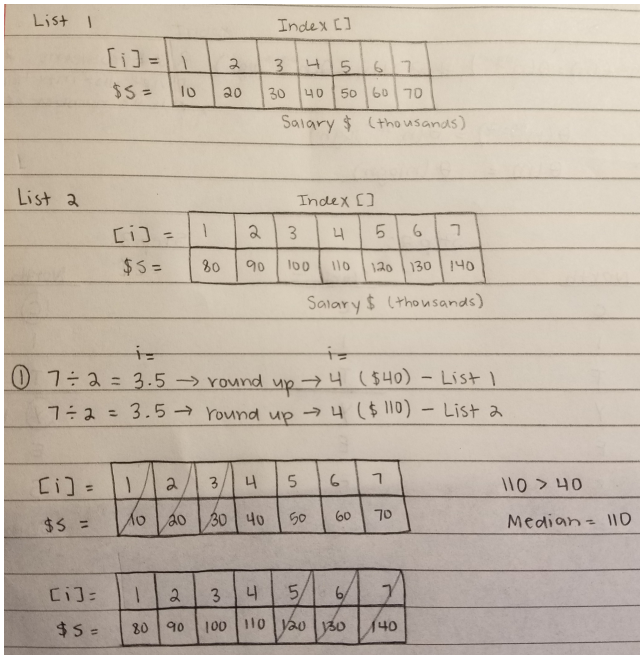

Figure 5: Algorithm for Median Salary

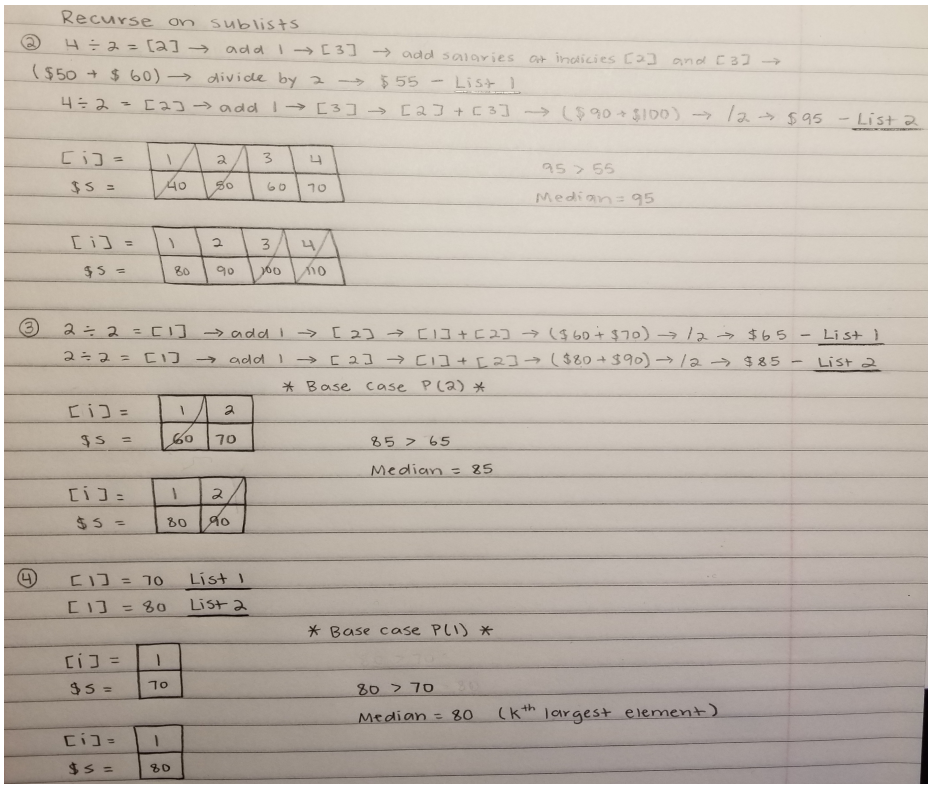

Figure 6: Algorithm for Median Salary (Continued)

In order to find the median in a list or sublist of an even length, you first get the length of the list and divide it by 2. The resulting number is the index in the list who's value you will be using $(\frac{n}{2})$, but because it's an even case, there is no clear median in the list, so you add 1 to the index you got to get to the next index $(\frac{n}{2}+1)$. You then add the values at those 2 indices together and divide it by 2 to get the median. An example of this can be shown in step 2 of Figure 6. In an odd case, when the list of elements is of an odd length, you divide the length by 2, and round up/take the ceiling of the resulting number to get the index, who's corresponding value is the median salary. An example of this can be shown in step 1 of Figure 5. Given two lists of salaries with no duplicate salary amount, you first find the median of both lists, you then compare them, and cross out the values less than the smaller median and greater than the larger median, which can be seen in step 1 of Figure 5. From there, you can recur on the smaller sublists, as shown throughout Figure 6, that are left after you cross out values less than or greater than the median, as explained before. When there is only one element/value left in each sublist (a base case), you compare the two values and take the bigger value as the median, as shown in step 4 of Figure 6. *Note: if the median exists in the list(in an odd case), you keep the number, if the median was calculated(in an even case) the number does not exist anymore*. Another base case would be if the medians in both lists are the same at the end. This would mean you would have to find the number to the right of the median in both lists, compare the values, and the smaller number would be the median of the lists.

P.S. In code, you would pass in the first and last indices to find the middle so that the median does not shift (TA Explained)

2. State the complete recurrence for your algorithm. You may put your $f(n)$ in big-theta notation. Show that the solution for your recurrence is $\Theta(\log(n))$.

Recurrence Equation: $T(n) = T(\frac{n}{2}) + \Theta(1)$
We recursively split both lists in half each time we find the current median: $(\frac{n}{2})$
The constant time it takes to find the median: $\Theta(1)$

Master Theorem Case 2:

$$
\begin{aligned}
&if\ f(n) = \Theta(n^{log_b a}),\ then\ T(n) = \Theta(n^{log_b a} log n)\\
&a = 1\\
&b = 2\\
&f(n) = \Theta(n)\\
&T(n) = T(\frac{n}{2}) + \Theta(1)\\
&\Theta(n) = \Theta(n^{log_2 1}),\ then\ T(n) = \Theta(n^{log_2 1} log n)\\
&\Theta(1) = \Theta(1) \qquad\qquad\qquad\quad = \Theta(log n)
\end{aligned}
$$

3. Prove that your algorithm above finds the correct answer. *Hint: Do induction on the size of the input.*

Strong Induction

Given $\forall k < n$, our algorithm finds the median for 2 lists of size k
Goal: $\forall k \in \mathbb{N}, P(k)$ holds

Base cases: $P(1)$ and $P(2)$
P(1) - When there are 2 lists with one element in each, you compare the two numbers, and the larger value of the two becomes the median, the kth largest element. (Step 4, Figure 6)
P(2) - Same algorithm as mentioned before: when there are 2 elements remaining in a list, find median in both lists, compare them, cross out values smaller than small median and larger than large median, then look at the remaining sublists, which in this case would only have one element left in each after recursing, and apply base case one. (Step 4-5, Figure 6)

Hypothesis: $\forall_x \leq x_0, P(x)$ holds
Inductive Step: show $P(1)$,...,$P(x_0) \Rightarrow P(x_0 + 1)$

$M1$ is the median of both lists combined
$M2$ is the median of both lists after dividing in half

$P(x_0 + 1) = M1$
$P(\frac{x_0}{2}) + 1) = M2$

For the reduction step in the inductive step, we know that M1 = M2 because the median is the same even when we divide the list of each school's salaries in half ($\frac{k}{2}$). The reasoning being, when we divide the lists in half, we are removing the extreme-most values on the end of each list, not the values in the middle, hence why the median remains the same. Thus, our algorithm does in fact find the correct median.

Extra:

For $P(n)$, there are 2 lists of size n, from which you get the median salary of both lists and compare the two values, from which you then cross out numbers smaller and larger than the medians to get your smaller sublists, and then recursively call the algorithm until you reach the base case(all described before).

For $P(n + 1)$, there are 2 lists of size $n + 1$, from which you find the median of both lists, same as the way explained before, by finding the length and dividing by 2, compare the medians, cross out the numbers smaller than the smaller median and greater the bigger median, and since the new sublists of size k are $<$ n, we know our algorithm will work on the smaller sublists because our inductive hypothesis was proven to be true.