Anthony Chui
waxaA
Adam Yin
12278

Recurrence Relation

1) a) $T(n) = 2T(\frac{n}{2}) + C_1$
$T(1) = C_1$

Closed form Solution

$T(n) = 2(2T(\frac{n}{4}) + C_1) + C_1$
$= 2 \cdot 2 \cdot (2 \cdot T(\frac{n}{8}) + C_1) + 2C_1 + C_1$
$= 2^k \cdot T(\frac{n}{2^k}) + \frac{(1 - C_1^{k+1})}{1 - C_1}$

Let $k = \lg n$

$T(\lg n) = n \cdot T(1) + \frac{(1 - C_1^{\lg n + 1})}{1 - C_1}$
$= n \cdot C_1 + \frac{(1 - C_1^{\lg n + 1})}{1 - C_1}$

$T(n) = 2^n \cdot C_1 + \frac{(1 - C_1^{n+1})}{1 - C_1}$

1) b) Base Case: # elements in A = 1
This means length = 1, left = 0, right = 1,
So the max value is returned

I.H: Assume findMaxHelper(A, left, (right+left)/2) gives max of indices left of $\frac{right + left}{2}$

and assume findMaxHelper(A, (right+left), right) gives max of indices right of $\left(\frac{right + left}{2}\right)$

Inductive Step: Given any left and right of A the function returns the max of findMaxHelper(A, left, (right+left)/2) and findMaxHelper(A, (right+left)/2, right). By I.H, the former gives the left max of (right+left)/2 and the latter gives the right max of (right+left)/2
The max of both recursive calls must give the max of left and right QED.

2)a) Invariant: The index of the smallest value of array x from index i to j is contained in iom.

Base case: Let j be equal to n. Then i is equal to j-1. This means there is exactly 1 element between index i and j. iom now contains that element, which is the smallest value of array x from index i to j.

Inductive Step: Let j be equal to n-k. Assume x[iom] is the smallest element between index i and j-1. For this iteration, x[j] is the smallest element between i and j if x[j] is smaller than x[iom]. Once iom contains j, the invariant is maintained and the next loop iteration continues.

2)b) Invariant: If i > 0, each element in array x from index i to n is greater than the element at index i-1.

Base Case: After the first iteration of the loop, i = 1, so index 0 contains the smallest element of array x. This is because in the first iteration, the element at index 0 and the smallest element swap indices. The invariant then holds. We then find the next smallest element and swap it with index i.

2)b) Inductive Step: Because index $i-1$ stores an element smaller than every element to the right of it, the invariant holds. By finding the next smallest value from index i to n and swapping it with the element at i, index i contains the smallest value from i to n too. By iterating through the loop, the invariant holds.

2)c) Using these 2 invariant, each iteration of A (which results in loop B), the array is sorted from index 0 to i.

Base Case: $i = 0$, Single element is sorted

Inductive Step: $i = k$. The array is sorted from index 0 to index $i-1$. Each iteration of A places the lowest element value from index i to $n-1$ and stores it in i. Invariant A states the array is sorted from index 0 to i. The invariant holds for $i = k+1$ to the entirety of the loop. Once $i = n$ (loop terminates), the first $n-1$ elements are sorted in increasing order.

3) Assume there exists 5 computers in the room to start off. Each computer could be connected to from one to four computers (we do not include zero).

Using the pigeonhole principle, there must be at least 2 computers with the same number of connections.

Now, let's go back to the original problem with 6 computers connected to from zero to five computers each. If one computer has no connections, then no computer can have 5 connections. This leads us back to the above scenario, where there are 5 computers with connections to from one to four computers. We already showed there exists 2 computers with the same number of connections.

If more than one computer has no connections, then that already implies two computers have the same number of connections.

If no computers have no connections, then that means there are 6 computers with 5 different possible connections. Again, by the pidgeonhole principle, there will be 2 computers with the same number of connections.

We have covered all cases of the problem.
QED.

4) Assume the situation where we have 8 buckets and we need to take 4 buckets with at least one red ball. We could achieve this by filling 5 buckets with one red ball each. This is the same as saying we fill 3 buckets without a red ball.

Out of one of the 4 buckets we took, assume we remove a bucket with a red ball and place the rest back. Now, with the 7 buckets, assume we place green balls such that taking any 3 buckets would result in at least 1 green ball. We could place a green ball in each of the buckets with red balls and 1 green ball by itself to accomplish this. We know this works because if we take 3 buckets out of 7 where only 2 do not contain green balls, we are guaranteed at least one green ball.

Let's say that out of the 3 buckets we choose, we remove a bucket with a green ball and place back the other buckets. Again, we place into the buckets a new blue ball such that by taking 2 buckets, one is guaranteed to have a blue ball. ~~Depending on whether we removed a bucket with red and green or just green in the previous situation,~~ We can place a blue ball with each green bucket and a blue ball into an empty bucket. In the case that we previously remove a bucket with red and green balls, one bucket will contain only green and blue.

4) Either way, with only one non-blue (empty)
cont. bucket, we are guaranteed a blue bucket by taking 2 buckets.

Once again, we take 2 buckets, remove the one with blue, and place the other back. We introduce a yellow ball and to guarantee taking 1 bucket will have a yellow ball, we place a yellow ball into all 5 remaining buckets. At this point we have a valid solution

1: R 2: RG 3: RGB ~~*~~ 4: RGBY 5: RGBY
6: GBY 7: BY 8: Y

By analyzing the minimum balls we needed of each colour, we conclude that we need 5 balls of each colour to guarantee taking 4 buckets from 8 would result in all colours in a bucket.

1

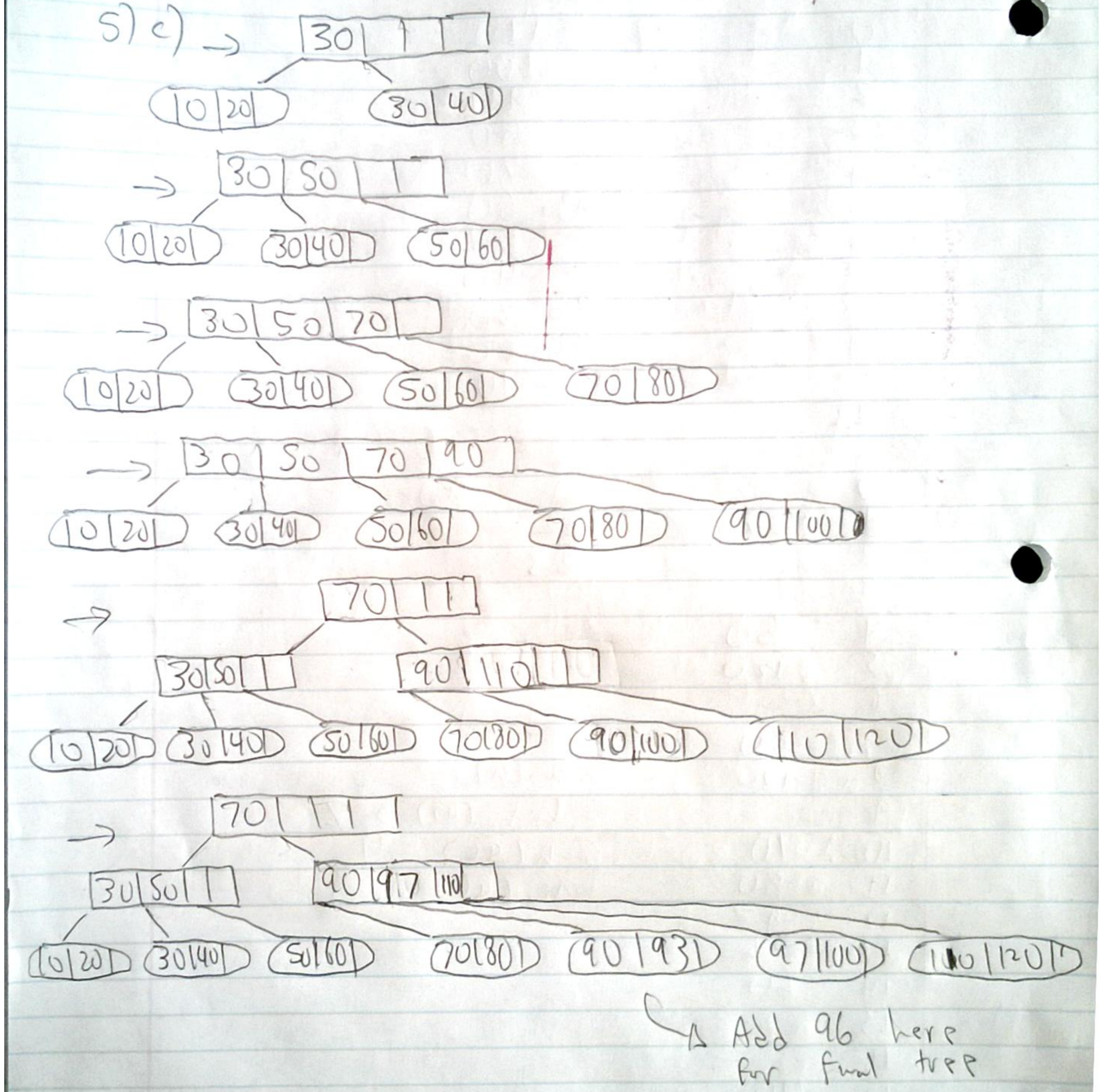
5) a)

$$\begin{tikzcd}
\boxed{0} \arrow[r] & 110 \\
\boxed{1} \arrow[r] & 100 \\
\boxed{2} \arrow[r] & 90 \\
\boxed{3} \arrow[r] & 80 \\
\boxed{4} \arrow[r] & 70 \\
\boxed{5} \arrow[r] & 93 \arrow[r] & 60 \\
\boxed{6} \arrow[r] & 50 \\
\boxed{7} \arrow[r] & 40 \\
\boxed{8} \arrow[r] & 96 \arrow[r] & 30 \\
\boxed{9} \arrow[r] & 97 \arrow[r] & 20 \\
\boxed{10} \arrow[r] & 120 \arrow[r] & 10
\end{tikzcd}$$

b)

0:		21:	90
1:	70	22:	
2:		23:	
3:			
4:	50		
5:	120		
6:			
7:	30		
8:	100		
9:			
10:	10		
11:	80		
12:	93		
13:	96		
14:	60		
15:			
16:			
17:	40		
18:	110		
19:	97		
20:	20		

5) c)


30
10 | 20
30 | 40
30 | 50
10 | 20
30 | 40
50 | 60
30 | 50 | 70
10 | 20
30 | 40
50 | 60
70 | 80
30 | 50 | 70 | 90
10 | 20
30 | 40
50 | 60
70 | 80
90 | 100
70
30 | 50
90 | 110
10 | 20
30 | 40
50 | 60
70 | 80
90 | 100
110 | 120
70
30 | 50
90 | 97 | 110
10 | 20
30 | 40
50 | 60
70 | 80
90 | 93
97 | 100
110 | 120
Add 96 here for final tree

5) b)

10	20	30	40	50	60	70	80	90	93	96	97	100	110	120

6)a) $N(0) = 1 \quad N(1) = 2$

$N(h) = 1 + N(h-1) + N(h-2)$

6)b) Fibonacci recurrence relation

$F(h) = F(h-1) + F(h-2)$

$F(0) = 0 \quad F(1) = 1$

Comparing $N(h)$ and $F(h)$, $N(h)$ seems similar to $F(h+3)-1$

We will prove this by induction.

Base Case:

$F(0) = 1, \; F(3) = 2 \quad$ So $F(0) = F(3) - 1$

Inductive hypothesis:

Assume that $N(h) = F(h+3) - 1$ is true for $h = 1, \dots, k$

Inductive Step: Let $h = k+1$

$$\begin{aligned} N(k+1) &= N(k) + N(k-1) + 1 \\ &= F(k+3) - 1 + F(k+2) - 1 + 1 \\ &= F(k+4) - 1 \end{aligned}$$

Replacing $k+1$ with the equivalent h results in $N(h) = F(h+3) - 1$.

Because solving the Fibonacci recurrence relation results in the Binet formula, then assuming that is derived, plugging $h+3$ into the formula and subtracting 1 results in the min nodes of height h of an AVL tree.

6) c) The worst case height of an AVL tree is roughly $1.44\log_2 n$. This means that the total number of nodes including empty nodes is $2^{1.44\log_2 n+1}-1 = 2n^{1.44}-1$. Because the array for the AVL tree needs to account for empty nodes as well, its size is $\in O(n^{1.44})$. An AVL tree only needs to store $2n$ space for pointers and n space for data value, so it size is $\in (n)$. For low heights ($h \leq 2$) the array saves space but asymptotically the AVL tree saves space. This makes sense since a leaf node that doesn't reach the height of the tree can't store any extra pointers that do reach the height, whereas an array needs to maintain a size as if the tree was complete and full. In the best case scenario where a tree is complete & full, the AVL tree would be storing more space needed ($3n$) but the array stores n elements with no space in the array wasted.

Array
$\boxed{2},4.7.3$
[illegible]

3 [illegible]
4 [illegible]