Problem Set 2

All parts are due September 20, 2019 at 6PM.

Name: Lydia Yu

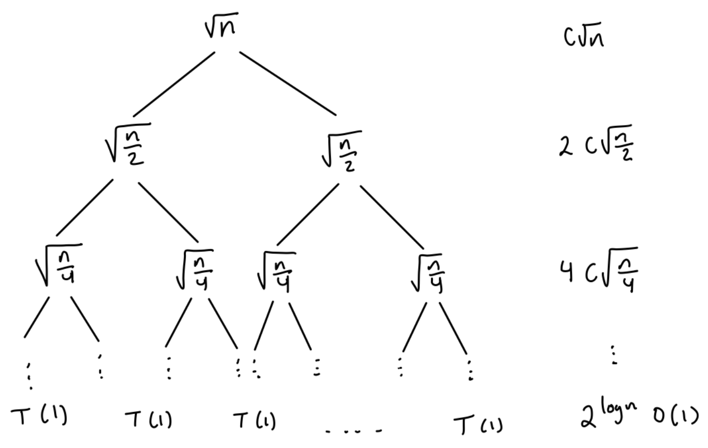
Collaborators: Terryn Brunelle, Tanya Yang, Herbie Turner, David Magrefty

Problem 2-1.

(a) $T(n) = \Theta(kn) + kT(\frac{n}{k})$

(b) Given the recurrence that we have from part a), which takes on the form of $T(n) = aT(n/b) + \Theta(n^c)$, we can use the Master Theorem case where $T(n) = \Theta(n^c \log n)$ when $c = \log_p q$ with p as the branching factor and q as the problem size reduction factor. $c = 1$ here since $T(n) = \Theta(n \log n)$, so p must equal q, which is true since both are equal to k based on the recurrence. Since c=1, we get from the recurrence form $\Theta(n)$, so kn must always be $\Theta(n)$. This means that k must always be constant, and since $k = n^a + b$, that means that a=0 to keep k a constant and b can be anything ≥ 1.

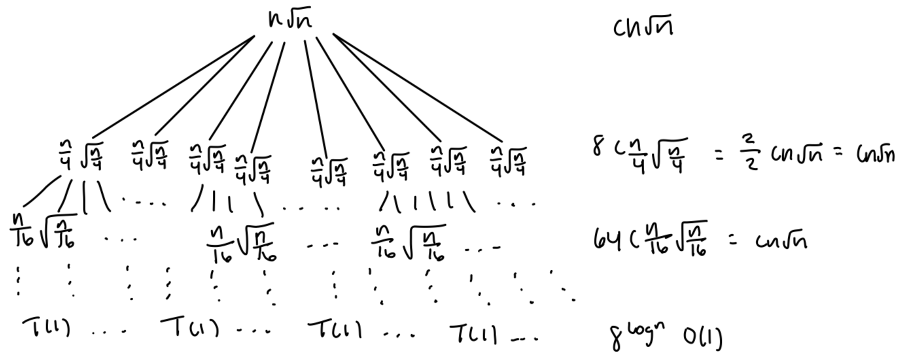
Problem 2-2.



$$c\sqrt{n}\,(1+\sqrt{2}+2+2\sqrt{2}+\ldots) = c\sqrt{n}\left(\sum_{i=0}^{\log n}\sqrt{2}^{\,i}\right) = c\sqrt{n}\;\Theta\left((1+\sqrt{2})(2^{1/2(\log n+1)}-1)\right)$$
$$= c\sqrt{n}\,\Theta(2^{1/2(\log n+1)}) = c\sqrt{n}\,\Theta(2^{1/2\log n+1/2}) = c\sqrt{n}\,\Theta(2^{\log\sqrt{n}}) = c\sqrt{n}\,\Theta(\sqrt{n})$$
$$= \Theta(n)$$

(a)

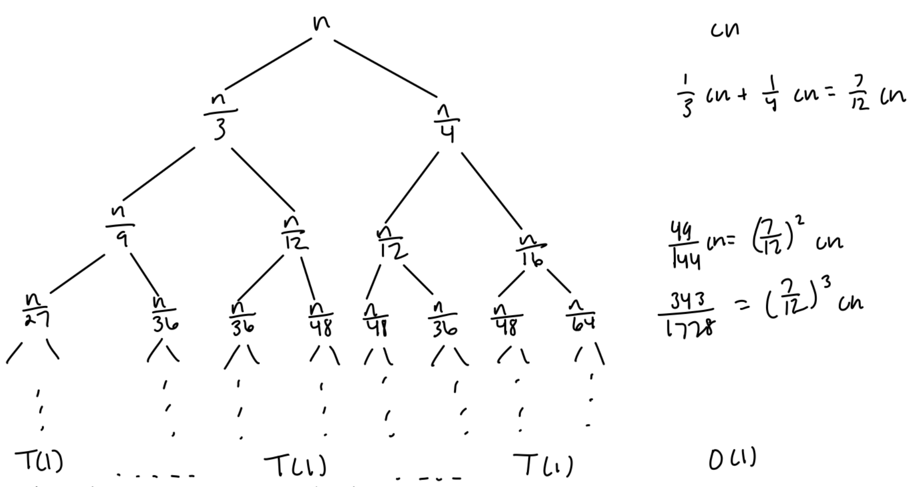
Master Theorem: this recurrence follows the form of $T(n) = aT(n/b) + \Theta(n^c)$ and therefore satisfies case 1 of the theorem because $a = 2, b = 2$, and $c = 1/2$, and $1/2 < \log_2 2 = 1$. Thus, $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.


$n\sqrt{n}$
$cn\sqrt{n}$
$\frac{n}{4}\sqrt{\frac{n}{4}}$
$8c\frac{n}{4}\sqrt{\frac{n}{4}} = \frac{2}{2}cn\sqrt{n} = cn\sqrt{n}$
$\frac{n}{16}\sqrt{\frac{n}{16}}$
$64c\frac{n}{16}\sqrt{\frac{n}{16}} = cn\sqrt{n}$
$T(1)$
$8^{\log n}\ O(1)$

$\log n$ levels total, each doing $cn\sqrt{n}$ work
↳ overall, $T(n) = \Theta(n\sqrt{n}\log n) = \Theta(n^{3/2}\log n)$

(b)

Master Theorem: this recurrence also follows the form $T(n) = aT(n/b) + \Theta(n^c)$ with $a = 8, b = 4$, and $c = 3/2$. $\log_b a = 3/2 = c$, so we use case 2: $T(n) = \Theta(n^c \log n) = \Theta(n^{3/2}\log n)$.



The $n/3$ branch represents the upper bound on the number of levels of the tree ($\log_3 n$), and the $n/4$ branch is the lower bound ($\log_4 n$)

upper bound: $cn \lim_{n\to\infty} \sum_{i=0}^{\log_3 n - 1} (\frac{7}{12})^i = \frac{12}{5}cn$

$\Rightarrow T(n) = \Theta(n)$

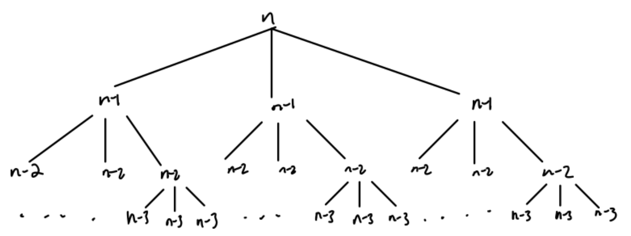
lower bound: $cn \lim_{n\to\infty} \sum^{\log_4 n - 1} (\frac{7}{12})^i = \frac{12}{5}cn$

(c)

Master Theorem: for this recurrence, $a = 1, b = 3$ or 4, and $c = 1$. Thus, $\log_b a = 0$ regardless of which value of b we use, so $\log_b a < c$. We can use case 3, which means that $T(n) = \Theta(n^1) = \Theta(n)$.

(d) The upper bound of this function is represented by the T(n-1) term and the lower bound is represented by the T(n-2) term because T(n-1) divides the overall problem up into more levels than T(n-2) does since n is decreasing by a smaller value each time. Thus, $T(n) \leq 3T(n-1)$ and $T(n) \geq 3T(n-2)$. We can solve this recurrence using two trees, one for $T(n) = 3T(n-1) + \Theta(n)$ and one for $T(n) = 3T(n-2) + \Theta(n)$ to find the upper and lower bounds.

$T(n) = 3T(n-1) + \Theta(n)$:



$n = 3^0 n - 0 \cdot 3^0$

$3(n-1) = 3n - 3 = 3^1 - 1 \cdot 3^1$

$9(n-2) = 9n - 18 = 3^2 - 2 \cdot 3^2$

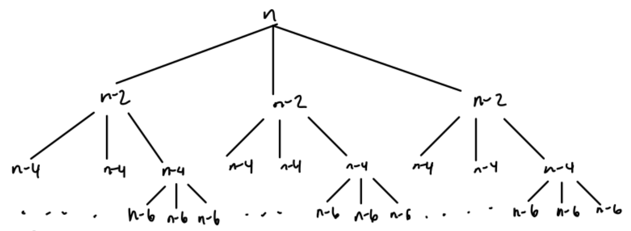
$27(n-3) = 27n - 81 = 3^3 - 3 \cdot 3^3$

n levels total because each time the problem reduces by 1, so you need n times to get to 0.

$$\sum_{i=0}^{n-1} 3^i (n-i) = \frac{1}{4}(-2n + 3^{n+1} - 3)$$

upper: $O(3^{n+1})$

$T(n) = 3T(n-2) + \Theta(n)$:



$3(n-2) = 3n - 6 = 3^1 n - 2 \cdot 1 \cdot 3^1$

$9(n-4) = 9n - 36 = 3^2 n - 2 \cdot 2 \cdot 3^2$

$27(n-6) = 27n - 162 = 3^3 n - 2 \cdot 3 \cdot 3^3$

$\frac{n}{2}$ levels total because $n - 2(\frac{n}{2}) = 0$

$$\sum_{i=0}^{n/2-1} 3^i (n-2i) = \frac{1}{2}(-n + 3^{n/2+1} - 3)$$

lower: $O(3^{n/2+1})$

Problem 2-3. In order for him to find the Stone, he first has to place bounds on the planets that he is searching because there are an infinite number of planets. The lower bound starts as the first planet, which is the one with the lowest positive integer. The upper bound can be found by going to a higher planet and asking if the Thoul Stone resides in a higher planet. If no, then this current planet becomes the upper bound for Sanos's search; if yes, then the new lower bound becomes this higher planet, and then go to the planet that has an exponentially higher number than the current one (for example, if he first went to planet 2, then go to planet 4, and then go to planet 8, etc) and repeat the same process of asking the oracle if the stone resides at a higher planet to determine if the upper bound needs to be higher and changing the lower bound. Once he has the lower and upper bounds, he can just complete a binary search with the planets within the bounds to find the one that the stone resides in - go to the planet in the middle and ask if the stone is on a higher planet; if yes, then go to the midpoint of the planets in the higher half of the original set; if no, then go to the midpoint of the planets in the lower half of the original set. Repeat this process for the new midpoint planet. The run-time of this algorithm is $O(\log k)$ because the part where he searches for the upper bound takes $O(\log k)$ since every time he goes to a new potential upper bound, the index increases exponentially (this is because we know that 2^p, where p is the number of planets he's gone to to find the upper bound, is $\geq k$, so $p \geq logk$). Thus, the upper bound is located at $2^{\log k}$ and the lower bound is located at $2^{\log(k-1)}$. Using these values, the binary search part, which divides the number of planets it's looking at in half each time, takes $O(\log(2 - 2^{\log(k-1)})) = O(\log(k - k/2)) = O(\log(k(1/2)) = O(\log(k) - \log 2) = O(\log k)$. Adding these together, we get $O(2\log k)$ which is the same as $O(\log k)$. We know that this algorithm will always return the location of the stone because you will always be able to find the bounds since the stone is located at a finite integer and there are an infinite number of planets. Once the bounds are found, we know that the stone is within the bounds, and the binary search will always return the correct planet because the oracle will tell us whether the stone is located at a higher planet, which is the information we need to go through the search.

Problem 2-4. We can use a sorted dynamic array whose slots are the nodes of a linked list - the sorted array stores the images by the sorted order of their IDs while the linked list keeps track of the order of how the images were added to the document. The head of the linked list is the first image that was added (at the "bottom"), and each image that is added is at the "top". Whenever a new image is added, its information is added to the sorted dynamic array and then connected to the previous node in the linked list (and when the array runs out of space, it completes the process of adding on the new empty slots).

1. `make_document()` is completed just by creating an empty dynamic array and an empty linked list that contains a pointer to the head, both of which are valid data structures. This takes O(1) time because it is just creating two empty data structures, which is done in constant time.

2. `import_image()` is completed by inserting the image at the correct sorted location based on its ID in the array, which runs in worst-case O(n) time because adding the new image to the correct location takes O($\log n$) since it just completes a binary search on the existing IDs to find the right location, but if the array has run out of space, then it would take O(n) to add the additional slots by copying the array into a new, larger one. Then, the image is added to the linked list at the very end

(it's added to the end so that `display()` can be done in O(n) time), which takes worst-case O(n) time because you start at the head and have to run through all of the nodes and their pointers to find the last node - then have the previous last node point to this new image. We know that inserting is a valid function for both linked lists and arrays. Both the sorted order of the images' IDs and the order in which they were added to the document are preserved with this method.
3. `display()` is completed by returning the images in the linked list. Since the linked list's nodes are already each pointing to the image that was added after it, we just start at the head and traverse through the entire list to output each image in the right order. This is completed in O(n) time because you just go through each node in the list, of which there are n total.
4. `move_below()` is completed by looking at the sorted array and using a binary search to find both x and y. Once x and y are found, x is moved to the position below y in the linked list by changing y's pointer to point to x and changing x's pointer to point to the element that was previously after y. Also, the node that used to be before x now has to point to the node that came after x. Finding x and y takes O($\log n$) time because it's just two binary searches, which overall always take O($\log n$). Switching pointers takes constant time because it doesn't depend on the size of the linked list and just involves reassigning the pointer directions for each node affected. Thus, the overall running time is worst-case O($\log n$) for this operation. We know that it is possible to find elements in a sorted array, and we also know that it is possible to add elements to a linked list. Changing pointers in the list is also possible by removing a pointer and reassigning it to a new node.

Problem 2-5.

(a) House 1 (34): damage = 4
House 2 (57): damage = 5
House 3 (70): damage = 6
House 4 (19): damage = 3
House 5 (48): damage = 3
House 6 (2): damage = 1
House 7 (94): damage = 4
House 8 (7): damage = 1
House 9 (63): damage = 1
House 10 (75): damage = 1

(b) First, we should find the non-special house by going to each house and checking whether the house adjacent to it to the east has a lower number of bricks. We don't need to worry about the second condition for special houses (no easterly neighbor) because every house is going to have an easterly neighbor until you get to the end of the list, and the house at the end is special. Once we find the non-special house, we remember its location and now view the row of houses in two sections: 1) the houses that come before the non-special house and 2) the houses including the non-special and afterwards. We already know that the houses in the two sections will not contribute to each other's damage within the sections because there is only one non-special house, so all of the houses are increasing in the number of bricks as you move to the east (except for when you reach the non-special house). This means that we only need to compare the houses from section 1 with the houses in section 2 to see if those houses have fewer bricks. To do this, we should start at the first house in the first section and compare it to the houses in the second section from west to east. Each time a house in the second section has less bricks than the current house in the first section, we add to the total damage. We stop comparing once we reach a house that does not have fewer bricks than the current house in section 1, and we remember the number of houses that contributed to the damage and the house in section 2 that we stopped at. When we move to the next house in section 1, we know that the houses from section 2 that contributed to the damage of the first section 1 house will also contribute to the damage of this next house because we know that this next house has a higher number of bricks (the houses from section two that already contributed to the damage will have fewer bricks than this new house). We can automatically add to this new house's damage with the houses that we already know will be blown down from the previous iteration. Now, we just have to start comparing the new house with the house that we stopped at in section 2 and repeat the same process of comparing remaining houses from section 2 with the current house of interest in section 1 until we get to a house that has more bricks. This process is done for each house in section 1 until we get to the one right before the non-special house, and the damage for each house in section 1 is returned. The houses in section 2 each only have a damage of 1 because, as explained above, all the houses after them have greater numbers of bricks. Finding the non-special house takes O(n) time because you are traversing the list of n houses and just comparing each one with the next one. Each comparison takes O(1) and you go through n houses, so the overall time is O(n). Comparing each house in section 1 with the section 2 houses also takes O(n) because each house is only passed by once, since we are saving the number of houses that have already contributed to the damage in each iteration. Thus, the overall running time is just O(n).

(c) For each house in the array, we want to keep track of both its brick number and position number in the array, which can be done using tuples. We also want an additional array that will keep track of the damage for each house in the order of the neighborhood. Then, we want to essentially run a merge sort on the entire array and keep track of the damage for each house while sorting - we will split the array in half into smaller sub-arrays until we get down to arrays with only one house in them. Then, begin to merge the array back in the sorted order while noting the damage for each house as we make comparisons. We look at each pair of houses starting from the left and going to the right - if the house on the right has less bricks than the left, we add a damage point to the left house's score in the damage array at its index from the tuple. These two houses are then combined to an array of size two, and this comparison/merging is repeated for the next pairs of houses (if there is an odd number of houses, the last group of houses will not be a pair and will instead merge into a group of 3, but the same comparison and adding damage strategy applies). Now, we have sorted arrays of two or three houses. To compare the arrays of houses and find the damage, take the rightmost house (the one with the highest number of bricks in the array) and compare it to the rightmost house in the adjacent array to the right. If the highest house in the left array is larger than the highest house in the right array, you can automatically increase the damage of the left array house by the size of the array on the right because they are sorted, so that house will have more bricks than every house in the right array. That highest house then becomes the largest one in the merged sub-array. If it it does not have more bricks, add the highest right-array house to the merged sub-array and compare the left-array house with the next highest right-array house. Once a house on the left-array is added to the merged sub-array (after we've determined that it has more bricks than all of the remaining houses in the right-array), we move our focus to the house in the left-array with the next highest number of bricks. Repeat this process of comparing bricks of houses in one array with the bricks of the houses in the array adjacent to it, adding to the damage score if applicable, and merging the houses in the sub-arrays, until we have finished merging the entire array back together, sorted from lest number of bricks to most. We will also end up with an array of the damages of each house in the order that they were originally found in. The running-time of this algorithm is $O(n \log n)$ because creating the tuples of the bricks and index for each house takes O(n) time, sorting the houses using merge sort takes $O(n \log n)$, and updating the array of damages takes O(1) time for each comparison made because we know the original index for each house and do not need to iterate through any houses. The $O(n \log n)$ time for the merge sort dominates, so the overall running-time is $O(n \log n)$.

(d) Submit your implementation to `alg.mit.edu`.