Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — September 13, 2019
Instructors: Jason Ku, Julian Shun, and Virginia Williams — Problem Set 2

Problem Set 2

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

Name: Diego Escobedo

Collaborators: Noah Lee, Nicholas Baginski

Problem 2-1.

(a) The recurrence relation is simple. The branching factor is k (because we have k arrays), the problem size reduction factor is also k (because the problem is now divided into those k subarrays, and the work function is $O(k^2 * m)$. However, we know that $n = k * m$, so we can substitute the m for $\frac{n}{k}$. Therefore, the recurrence relation is $T(n) = kT(\frac{n}{k}) + O(kn)$.

(b) Famy sort will run in $\Theta(nlog(n))$ time only if condition 2 for Master's theorem is satisfied, which means that $O(kn) = O(n^c)$ and $c = 1$ must be true. If we expand k, we see that $O(n^{a+1} + bn) = O(n)$. This can only be true when a = 0, though b can be any constant.

Problem 2-2.

(a) For Master's Theorem, we can tell that , $a = 2$, $b = 2$, and $c = \frac{1}{2}$. Plugging this into the equation we can see that this falls under case 1, where the work done at the leaves dominates. This means that the runtime is $\Theta(n^{log_b a})$, which is simply $\Theta(n)$ because $log_b a = 1$.

For the recursion tree, we know that the work being done at each level is going to be $2^i * \sqrt{n * 2^{-i}}$, where the root node is at level 0. Because the number of nodes is being reduced by half each time, we know that there will be $log_2 n$ levels. We also know because of master's theorem that the work at the leaves will dominate the work in the rest of the tree, meaning if we add up the work on non leaf nodes it will be pointless because the terms on th leaves will dominate it anyway. Therefore, we can simply plug in $log_2 n$ for i in $2^i * \sqrt{n * 2^{-i}}$ to find the run time. This simplifies to $2^{i/2} * \sqrt{n}$. If we plug in $log_2 n$ then we get $\sqrt{n} * \sqrt{n}$, which tells us that the runtime is $\Theta(n)$.

(b) For Master's Theorem, we can tell that , $a = 8$, $b = 4$, and $c = \frac{3}{2}$. Plugging this into the equation we can see that this falls under case 2, where the work is spread throughout the tree. This means that the runtime is $\Theta(n^c log(n))$, which is simply

$\Theta(n^{\frac{3}{2}}log(n))$.

For the tree, this one is not as simple as the last one. We need to consider the work being done in every node of the tree. We know that the general formula for work done at a specific level is $n * 4^{-i} * 8^i * \sqrt{n} * \sqrt{4^{-i}}$. We need to do a summation of these terms from 0 to $logn$ (because we are quartering the number of values per node, it will take $logn$ to be split completely). Therefore, $T(n)$ =

$$\sum_{i=0}^{logn} n * 4^{-i} * 8^i * \sqrt{n} * \sqrt{4^{-i}}$$

From here, we can pull out the terms without i AND also change the base of the 8 to get

$$n^{\frac{3}{2}} \sum_{i=0}^{logn} 4^{-i} * 4^{\frac{3i}{2}} * 4^{\frac{-i}{2}}$$

The exponents here add to 0 so we're just summing 1 logn times, which is just logn. This gives us $T(n) = \Theta(n^{\frac{3}{2}}logn)$, which confirms what we got from Master's theorem.

(c) We can't solve this problem using Master's theorem in its current form. However, we can bound this problem from above or below and then estimate from that. We can bound the worst case by assuming $T(n) = 2T(\frac{n}{3}) + \Theta(n)$. If we apply Master's Theorem (a = 2, b = 3, c = 1), then we can see that this relation is $O(n)$. If we assume the best case, then we can assume $T(n) = 2T(\frac{n}{4}) + \Theta(n)$. If we apply Master's Theorem (a = 2, b = 4, c = 1), then we can see that this relation is $\Omega(n)$. Because O an Ω are the same, then we know that $T(n) = \Theta(n)$.

Because the work done at the root dominates the work done at lower levels, we just need to calculate the work at the root, which from the given recurrence is just $\Theta(n)$.

(d) Again, since we can't solve this directly, we need to find upper and lower bounds by assuming $T(n) = 3T(n-1) + \Theta(n)$ for the worst case and $T(n) = 3T(n-2) + \Theta(n)$ for the best case. In the worst case, the depth of the tree is n, so the work at each level is 3*(n) + 3^2*(n-1) + 3^3*(n-2) + ... which means that the work at the leaves will dominate the work in the rest of the tree. Therefore, we only need to worry about the work done at the bottom level. At the bottom level, each node has only 1 data point, but there are 3^n nodes. This means that the work will be 3^n, which is $O(3^n)$. In the best case, there are only half as many levels as above. Therefore, instead of tehre being 3^n nodes there are $3^{\frac{n}{2}}$ nodes, again with constant sized work. The complexity is therefore $O(\sqrt{3^n})$. Therefore, the complexity of this recurrence is bounded by above by $O(3^n)$ and from below by $O(\sqrt{3^n})$.

Problem 2-3. We initialize two pointers, call them low and high, at positions 0 and 1. We ask the oracle at high if k is lower than high's position. If it isn't then we move low to the position at high, and high to a position 2 times larger than it was before. If the oracle says it is lower, then we perform binary search between low and high. We know that this range between low and high is guaranteed to be at most 2k long because in the worst case, we reach the spot right before k, and high doubles the distance, meaning that its 2k units long. However, this is a constant so it stills runs in log k time.

In terms of runtime, this algorithm will finish in log k time because each of the two steps takes log k time. Because we are doubling the location of the high counter each time, we will reach k in logk steps. Once we have this range (which we proved above that is at most of length 2k) then we know that a binary search can find any number within this 2k long array in log(2k) time, which is just logk. Therefore, we have 2 logk time operations, which is just $O(log(k))$.

In terms of correctness, because we are putting a lower bound and an upper bound, we are guaranteed to 'capture' k within this upper and lower bound. From tehre, we know that a binary searc is correct, so we are guaranteed to find k.

Problem 2-4. We are going to use a combination of: a doubly linked list, which will represent the actual order of the images, and a dynamic array that holds the IDs in order and a two-way pointer connected to the location of the linked list node associated with that ID.

The first operation is trivial; both our data structures take any time to initialize when they're empty.

The second operation is to add an image to the top (end). We can insert it into the doubly linked list in O(n) since we traverse the whole list to place it at the end, and insert it into the sorted array in O(n) time (we learned that insertion into a sorted array is O(n) in lecture).

The third operation is to print the IDs in order of how they're stacked in the image. This is simple; we just traverse the linked list, getting the corresponding ID from the doubly linked pointers, which happens in O(n) time as we traverse the list.

The fourth operation is slightly more complicated, but manageable. We need to:
1. binary search for ID x #O(log(n))
2. Use x's doubly linked pointers to splice x-1 and x+1 together #O(1)
3. binary search for ID y #O(log(n))
4. Connect y-1 to x's node.before and y to x' node.after #O(1)
Because we are using y's doubly linked pointer to find y-1, we are guaranteed that we are placing x right before y. Also, we know that a doubly linked list can be spliced together by changing the locations on the .before and .next pointers of x-1 and x+1.

And just like that, we found a solution that works. We discussed correctness and runtime during the explanation of the solution.

Problem 2-5.

(a) The damage for this specific setup is [4, 5, 6, 3, 3, 1, 4, 1, 1, 1]

(b) The algorithm is going to do the following:
1. Taking the elements of the array 2 at a time using two pointers called 'left' and 'right', find the element where the number at left is bigger than the number at right. This is our non-special house, which we found in O(n) time.
2. We are going to create an array called D of length n, composed of all 1s. Again done in O(n) time.
3. We are going to return the left pointer to the very first element of the list, and create a pointer at non-special plus one called 'end.
4. Compare left and right. If left is bigger than right, increase D[index of left] by 1, and then move right 1 step to the right. If right is bigger than left, move left 1 step to the right, and move right to where end is. 5. Terminate when left reaches end. The procedure will do all these comparisons in at most O(n) time, because in the worst case where . This guarantees correctness because every house after the non-special one will have only 1 house knocked down. For the ones before the non-special one, we are iterating through each one, and iterating through every house after the non-special one to see if they are smaller than that one and would therefore be knocked down. Therefore, we don't leave the damage for any house uncalculated, INCLUDING the non-special one (because end is at non-special plus one).

(c) For this part of the problem ,we are going to use the same procedure as above, but also implement some characteristics of mergesort to achieve O(nlogn) runtime. The algorithm is going to do the following:
1. Create an array called D filled with all 1s that will correspond to the damage that each house causes.
2. Separate our array exactly like we would in mergesort, until we get down to the lowest level.
3. Append the two arrays together. If the left is bigger than the right, increase the corresponding damage by 1 in D. Otherwise, do nothing. 4. Merge and sort the two single length arrays. We are now going to have the second level of mergesort, where we have arrays of length 2. We are going to append all of these length 2 arrays in pairs (which will leave us with many arrays of length 4), and run our procedure from 5b on them. 5. Rinse and repeat this step: Merge and sort to make a bigger array, append two adjacent arrays to each other, run 5b on them, increase the corresponding index in D if necessary.

In terms of runtime, we know that this will run in O(nlogn) because of the way mergesort works. Mergesort, at each level, does a linear amount of work, and since you are dividing it by two at each level, tehre are logn levels. This is why mergesort take nlogn time. However, we proved above that the procedure from 5b runs in n time. If we take the combined procedures (sorting + merging + 5b) * logn levels, we can see that it will be (n + n) * logn, which is just 2nlogn, which is asymptotically the same as O(nlogn).

In terms of correctness, we know this is going to be correct because it is the same procedure as 5b, except we are splitting up a big array into multiple small ones and running the algorithm in 5b on the arrays in the smaller unmerged ones. Thus, we have a proof of cerrectness by induction.

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