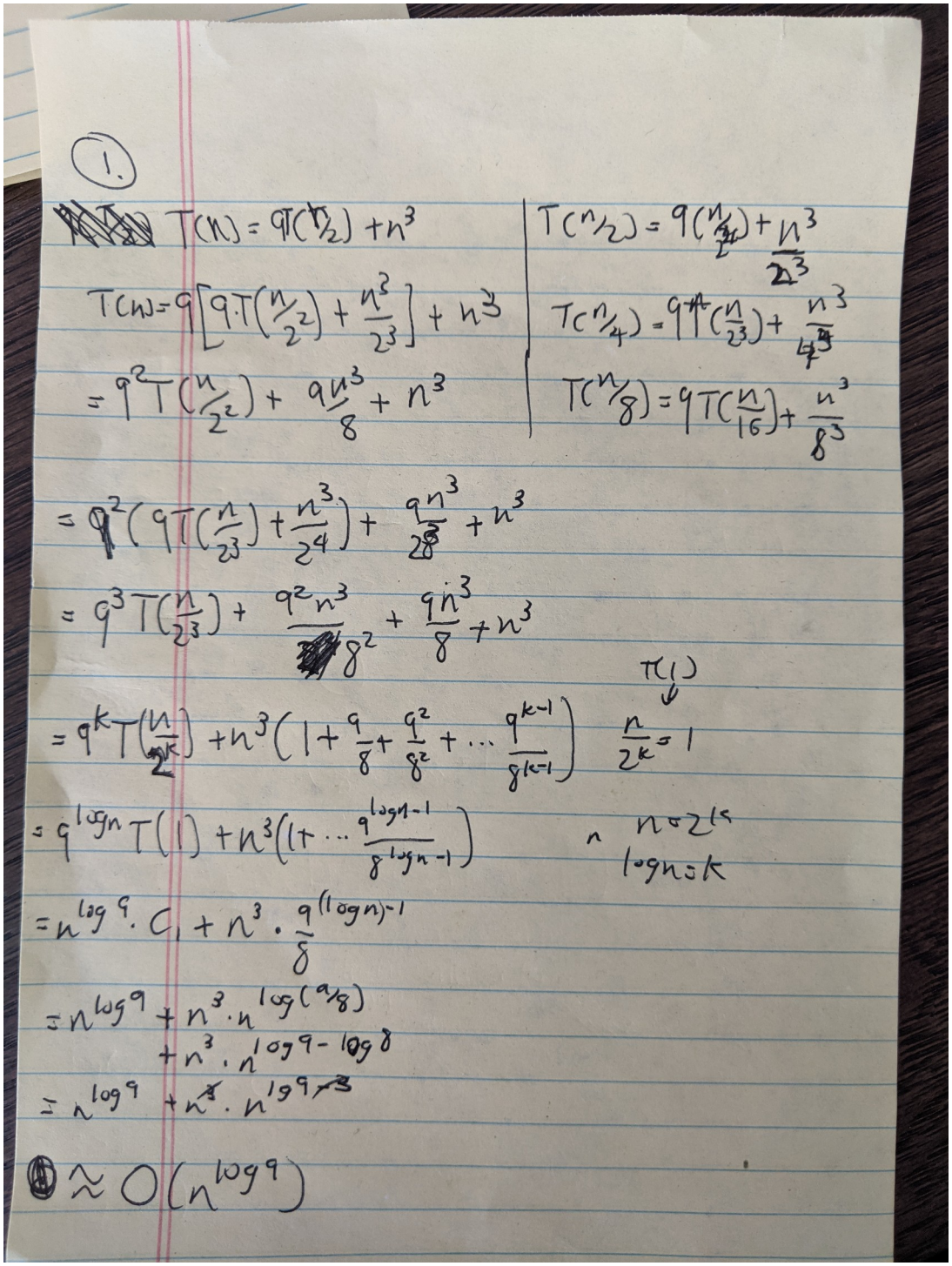
Will Frazee
HW #1
COT5405

1. Give asymptotic upper bound for T(n) in each of the following recurrences. Make your bounds as tight as possible.

- $T(n) = 9T(n/2)+n^3$



1.

$T(n) = 9T(n/2) + n^3$

$T(n) = 9\left[9 \cdot T\left(\frac{n}{2^2}\right) + \frac{n^3}{2^3}\right] + n^3$

$= 9^2 T\left(\frac{n}{2^2}\right) + \frac{9n^3}{8} + n^3$

$= 9^2\left(9T\left(\frac{n}{2^3}\right) + \frac{n^3}{2^4}\right) + \frac{9n^3}{2^3} + n^3$

$= 9^3 T\left(\frac{n}{2^3}\right) + \frac{9^2 n^3}{8^2} + \frac{9n^3}{8} + n^3$

$= 9^k T\left(\frac{n}{2^k}\right) + n^3\left(1 + \frac{9}{8} + \frac{9^2}{8^2} + \cdots \frac{9^{k-1}}{8^{k-1}}\right)$

$T(1)$

$\frac{n}{2^k} = 1$

$= 9^{\log n} T(1) + n^3\left(1 + \cdots \frac{9^{\log n - 1}}{8^{\log n - 1}}\right)$

$n = 2^k$

$\log n = k$

$= n^{\log 9} \cdot C_1 + n^3 \cdot \frac{9^{(\log n)-1}}{8}$

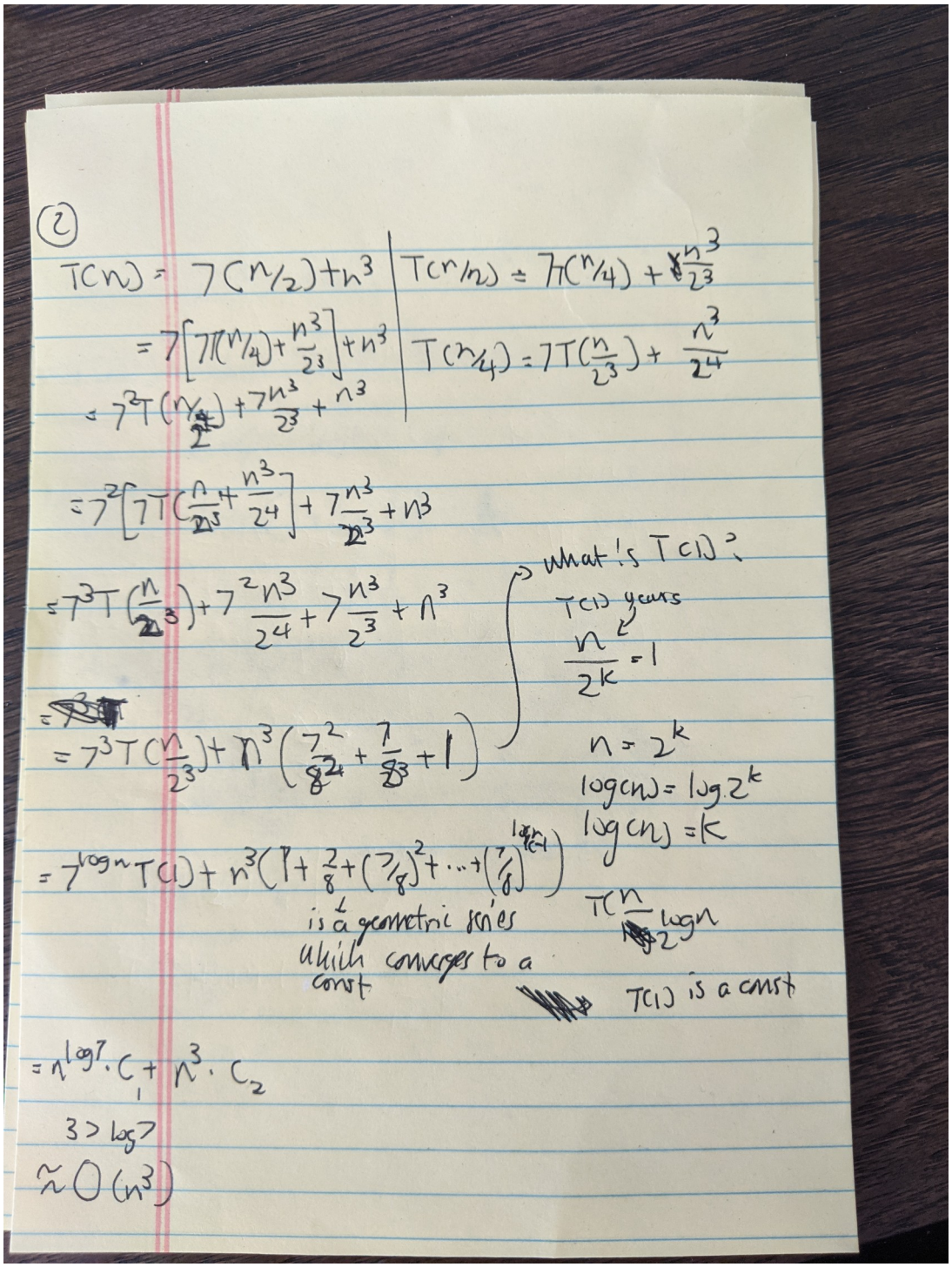
$= n^{\log 9} + n^3 \cdot n^{\log(9/8)}$

$+ n^3 \cdot n^{\log 9 - \log 8}$

$= n^{\log 9} + n^3 \cdot n^{\log 9}$

$\approx O(n^{\log 9})$

- $T(n) = 7T(n/2)+n^3$



(2)

$T(n) = 7(n/2)+n^3$

$= 7\left[7T(n/4)+\frac{n^3}{2^3}\right]+n^3$

$= 7^2T(n/4)+\frac{7n^3}{2^3}+n^3$

$= 7^2\left[7T\left(\frac{n}{2^3}\right)+\frac{n^3}{2^4}\right]+\frac{7n^3}{2^3}+n^3$

$= 7^3T\left(\frac{n}{2^3}\right)+\frac{7^2n^3}{2^4}+\frac{7n^3}{2^3}+n^3$

$= 7^3T\left(\frac{n}{2^3}\right)+n^3\left(\frac{7^2}{8^4}+\frac{7}{8^3}+1\right)$

$= 7^{\log n}T(1)+n^3\left(1+\frac{7}{8}+\left(\frac{7}{8}\right)^2+\cdots+\left(\frac{7}{8}\right)^{\log n}\right)$

is a geometric series which converges to a const

$= n^{\log 7}\cdot C_1+n^3\cdot C_2$

$3 > \log 7$

$\sim O(n^3)$

$T(n/n) = 7T(n/4)+\frac{n^3}{2^3}$

$T(n/4) = 7T\left(\frac{n}{2^3}\right)+\frac{n^3}{2^4}$

what is $T(1)$?

$T(1)$ years

$\frac{n}{2^k} = 1$

$n = 2^k$

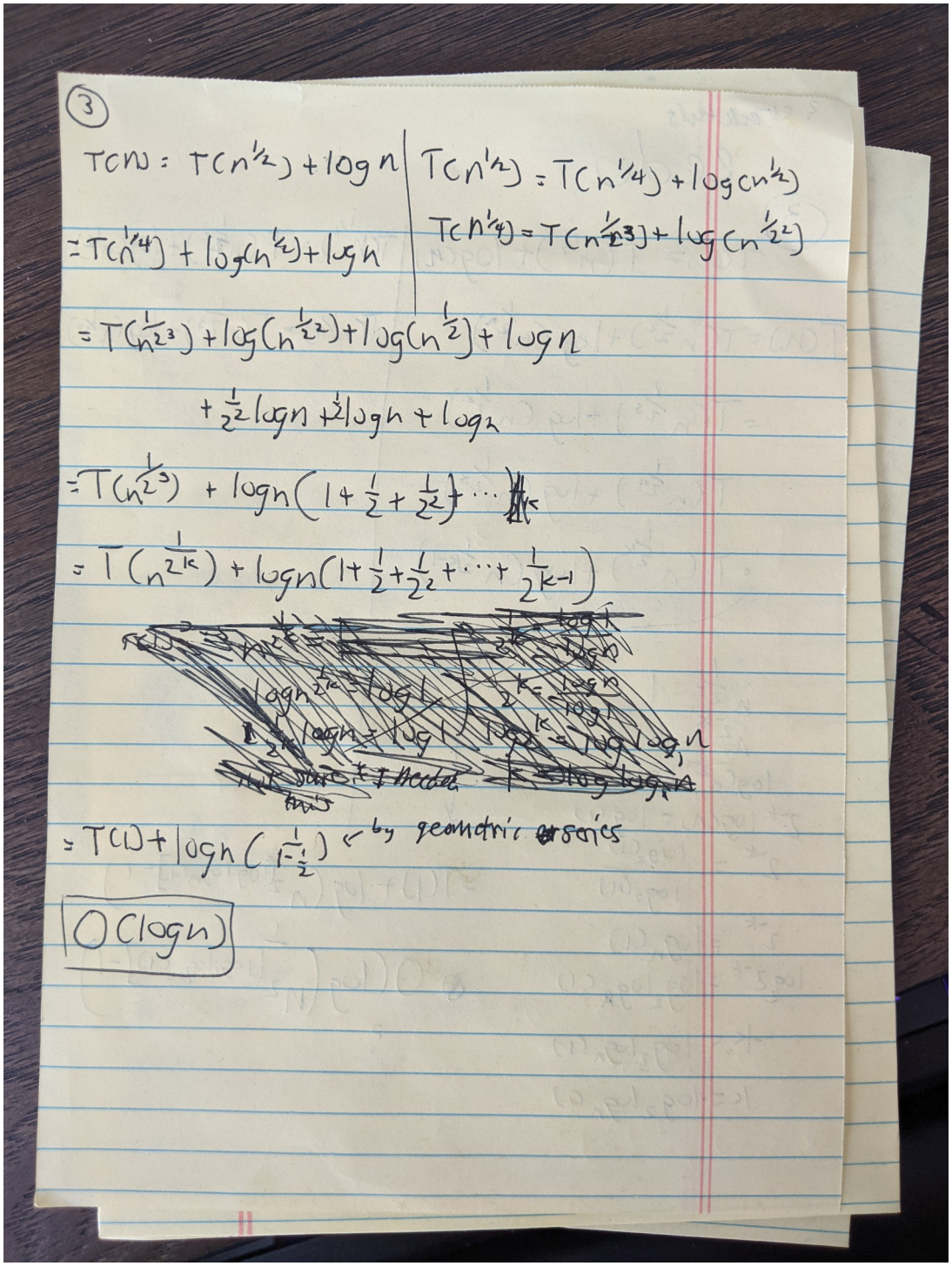
$\log(n) = \log 2^k$

$\log(n) = k$

$T\left(\frac{n}{2^{\log n}}\right)$

$T(1)$ is a const

- $T(n) = T(\text{sqrt}(n))+\log n$



- T(n) = sqrt(n)T(sqrt(n))+ n

(4) $T(n) = n^{1/2} T(n^{1/2}) + n$

$T(n^{1/2}) = n^{\frac{1}{2^2}} T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}}$

$T(n^{1/4}) = n^{\frac{1}{2^3}} T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2^2}}$

$T(n) = n^{\frac{1}{2}}\left(n^{\frac{1}{2^2}} T(n^{\frac{1}{2^2}}) + n^{\frac{1}{2}}\right) + n$

$= n^{\frac{3}{2^2}} T(n^{\frac{1}{2^2}}) + n^{2/2} + n$

$= n^{\frac{3}{2^3}}\left(n^{\frac{1}{2^3}} T(n^{\frac{1}{2^3}}) + n^{\frac{1}{2^2}}\right) + n^{2/2} + n$

$= n^{\frac{7}{2^3}} T(n^{\frac{1}{2^3}}) + n^{\frac{2^2}{2^2}} + n^{2/2} + n$

$= n^{\frac{2^k-1}{2^k}} T(n^{\frac{1}{2^k}}) + \ldots + 3n$

$T(2^m)$? $\rightarrow$ $n = 2^m$

$T(2^m)$? $\rightarrow$ $T(2^{\frac{m}{2^k}})$ $\rightarrow = 1$

$2^{\frac{m}{2^k}}$ $\quad \frac{m}{2^k} = 1$ $\quad m = 2^k$

$\log m = k$ $\quad n = 2^m$

$\log\log n = k$ $\quad \log n = \log 2^m$ $\quad m = \log n$

$= n^{\frac{\log n - 1}{\log n}} \cdot c + n \log\log n$

$n^{2^k - 1}$

$n^{\frac{2^{\log\log n} - 1}{\log\log n}}$

$n^{\frac{\log n - 1}{\log n}}$

this is just under linear

this is more than linear

$\approx O(n \log\log n)$

- T(n) = 3T(n/3)+n/3

(5)

$T(n) = 3T(n/3) + n/3$

$= 3[3T(n/3^2) + n/3^2] + n/3$

$= 3^2 T(n/3^2) + 3 \cdot n/3^2 + n/3$

$= 3^2[3T(n/3^3) + n/3^3] + n/3 + n/3$

$3^3 T(n/3^3) + n/3 + n/3 + n/3$

$= 3^3[3T(n/3^4) + n/3^4]$

$= 3^4 T(n/3^4) + 4(n/3)$

$= 3^k T(n/3^k) + k(n/3)$

$= 3^{\log_3 n} T(1) + \log_3 n \cdot n/3$

$= n^{\log_3 3} \cdot c + n/3 \log_3 n$

$= n + n/3 \log_3 n$

$= n + n/3 \cdot \log n / \log 3 = 1/(3 \log 3) \cdot n \log n + cn$

$\therefore O(n \log n)$

$T(n/3) = 3T(n/3^2) + n/3^2$

$T(n/3^2) = 3T(n/3^3) + n/3^3$

$T(n/3) = 3T(n/3^4) + n/3^4$

$T(1)$?

$n/3^k = 1$

$n = 3^k$

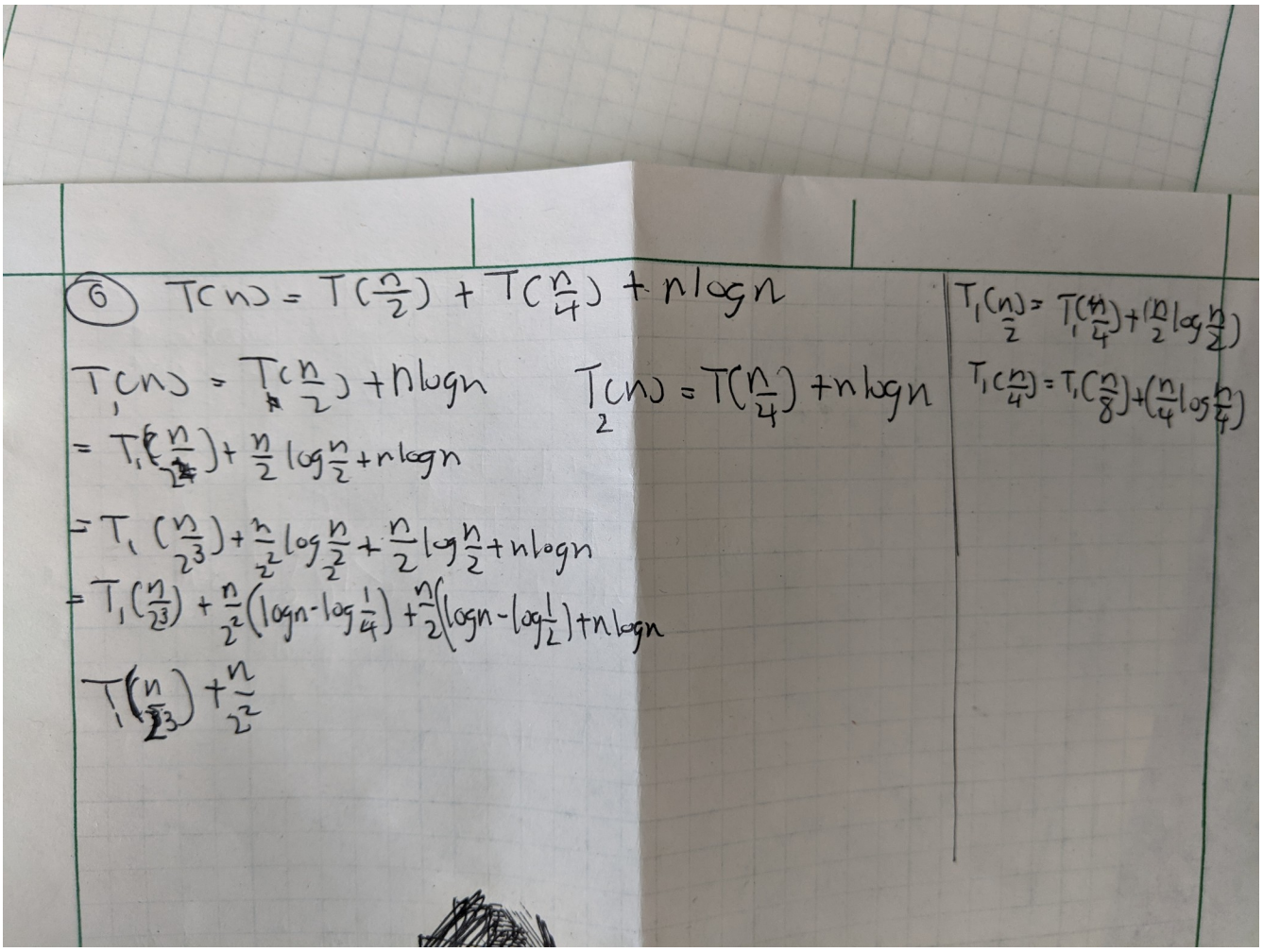
$\log n = \log 3^k$

$\log n = k \log 3$

$k = \log n / \log 3$

$k = \log_3 n$

- T(n) = T(n/2)+T(n/4)+ n log n



I am not quite sure what to do with this one. I'm not sure what a good guess world be, and when I tried solving the recurrences one by one, I became stuck.

2. Given an array A[1,n] of n numbers, compute the k-th smallest element in A for k=1,2,4,8,16,32 ... (i.e. all power of 2 up to n) in total time O(n).

Do median of medians, except when you get to the step of comparing K to the length of either subarray, compare all the Ks that you're interested in. Then run the select function on either array, depending on the result of the comparison. The trick here is that you don't need to do a select call for every single k at this step. For example, say that you have A_1 and A_2 and you're running for K=1,2,4,8. You determine that K=1,2,4 <= $|A_1|$ and that K=8>$|A_1|$. Just run two select calls: Select(A_1,1,2,4) and Select(A_2,8).

3. The following question has two parts:

- **Your friend, John, is holding an integer array A[1,n], and he tells you that the array is sorted, and contains all the integers from 0 to n except one. John challenges you to find the missing integer. He allows you to ask him questions in the form: What is the j-th bit of A[i]?" and he is going to answer you honestly. So,**

you can vary the values of i and j in each question you ask him. Design an algorithm to find out the missing integer using O($\log^2$ n) questions.

My solution works for when n is a power of 2. I'm sure that I could generalize it, but it just makes the math cleaner. Also, I think bits are 0 indexed? I can't quite remember but that's what I'm using.

If A[i] = i-1, then it plus every number before it is in the correct place. Take the example of n=4, where the number 2 is missing: [0,1,3,4] A[1] = 0 = 1-1, A[2]=1=2-1, A[3]=3, which means that it and any number after it is not in the right place. Another way to say this is that if A[i]=i, the missing digit occurs before A[i].

How can we use our question to help, along with this knowledge? We basically just need to confirm that every bit is in the right place.

Another way to think about it is that we are counting down in binary by the powers of 2, starting with n, and are building up our number. So, the algorithm:

missingno = 0
for j=n..0:
    If missingno = 0, Ask about the floor(sqrt(2^j)) bit of A[i] where i=2^j.
    If 1, continue because our missing number is less than the number we're currently examining. If missingno > 0, ask about A[i – 2^j-1], because we've already overshot to the right.
    If 0, our missing number is after A[i], add 2^j to missingno. On the next loop, instead ask about A[i+ 2^j-1], because we've overshot to the left (using the current bit).

I think this algorithm is only log(n), because the recurrence is of the form T(n/2)+c, same as binary search. I can't figure out a way to get it any faster.

- **Now, John has changed the input, and the array A becomes unsorted. Design an algorithm using O(n) questions to find out the missing integer.**

From i=1..n, determine the value of A[i] by summing up the value of 2^jth bit of each number. Then, add up all the numbers from 1..n as Sum_2. Then, subtract Sum_2 from Sum_1. This will get us the missing integer.

O(nlog(n)), I think. Not log(n), but I'm stumped.

4. Define maximum partial sum problem (MPS) as follows. Given an array A[1,n] of integers, and values of i and j with $1 \le i \le j \le n$ such that A[i]+A[i+1]+A[i+2]+…+A[j] is maximized. For example, for the array [3,-5,6,7,8,-10,7], the solution to MPS is i = 3 and j = 5 (sum is 21). Design an O(n log n) time algorithm. [Hint: Divide and Conquer approach].

Divide A into subarrays of n/2 length until we have subarrays of length 2.

As we climb back up the recursion, we will have two subarrays, A_L and A_R. At each step, we need to compare three sums:

The MSP of A_L

The MSP of A_R

A sum acquired as follows:

Merge A_L and A_R IN ORDER from left to right. Then iterate over the combination, starting at the index where the MSP of A_L begins. Keep track of three variables:

Sum: The sum from the index where the MSP of A_L starts to our current index.

Local Max: The largest sum that we've summed as we iterate. We update this when Sum $\geq$ Local Max.

K: This is the index where the Local Max ends. We update this to the current index when Sum $\geq$ Local Max.

After comparing all three sums, we keep track of what the greatest sum was (the MSP of the merged array), where it started (i) and where it ended (j). We return these variables plus the merged A_{L+R} to the next level of recursion.

A subarray of length 2 is considered trivial to discover the MSP, i and j.

This algorithm is T(n) = 2T(n/2) (for dividing into 2 smaller subproblems of n/2 size) + O(n) (for the merging + sum aspect). This recurrence is the same as Mergesort, which is O(nlogn).

This algorithm is correct because we always know the greatest MSP of any possible subarray at every step of the recursion. Given that the problem is all about finding the subarray with the greatest MSP, we are guaranteed to be correct.

5. Consider two sums, $X = x_1 + x_2 + \ldots + x_n$ and $Y = y_1 + y_2 + \ldots + y_m$. Give an algorithm that finds indices i and j such that swapping x_i with y_j makes the two sums equal, that is, $X - x_i + y_j = Y - y_j + x_i$, if they exist. Analyze your algorithm,

Assume that the input is given to us in the form of two arrays, X and Y. N is the length of X plus the length of Y.

First we need to find the difference by which we need to bring the sums together by. If we don't know the sums right away, sum together the arrays. When we have the sums, take the absolute value of their difference. Then, divide by 2. This is the difference by which the lower sum needs to rise, and the higher sum needs to fall.

Then, sort.

Then, for each number in the lower value sum array, do a binary search for that value plus our derived difference in sum value in the array of the higher sum. If we hit a match, we can swap those values for equal sums.

The sorting is O(nlogn). The binary search is O((length of one of the arrays)log(n)). This gives us an O(nlogn) runtime.

The correctness of this algorithm depends on the correctness of the way we calculated the difference between the two sums. If it is correct, then we would just need to swap x_i and $y_{j,}$ because that would subtract an amount *diff* from the larger number and add an equal amount *diff* to the larger number, making them equal.

6. Given a 2D array A[1,n][1,n] of n^2 numbers such that, in each row (from left to right) as well as in each column (from top to bottom), the elements are in the sorted order. Design an O(n) time algorithm to check if A contains a given number "x". Also, prove that it is not possible to design an o(n) time algorithm [i.e., a Ω(n) lower bound].

So, the solution is just the algorithm described in video lecture 2.3.

Start in A[1][n].

If X < A[i][j], move to A[i][j-1], with the knowledge that we don't need to check column j any longer, because it's guaranteed to be larger than a[i][j]. So we can move to to check the column to the left.

If X > A[i][j] then we move to check A[i+1][j], with the knowledge that we don't need to check the row I anymore, because it's guaranteed to be smaller than A[i][j]. So we can move on to check the lower row.

Ω(n) is the case, because: Let's assume that the worst case is that our target is in the bottom left of the array. The quickest path to that square is by going diagonally, from our starting square of A[1[n]. The number of comparisons that we would make on the way would be equal to the length of a row or column, n. Thus, Ω(n).

7. We say that an array A[1,n] is k-sorted if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks and smaller than the elements in later blocks. The elements within each block need not be sorted.
For example, the following array is 4-sorted:1,2,4,3|7,6,5|10,11,9,12|15,13,16,14

a) Describe an algorithm that k-sorts an arbitrary array in time O(n log k)

I'll just assume that n/k is an integer to make it clean.

Use median of medians to get the median value of the array O(n) time. Then, we iterate over every value in the array and slot it into two sub arrays. If a value in the array is less than the median, it goes into A1. Else, A2. Then, we run this algorithm on A1 and A2. We repeat until we have our base case of arrays of size n/k and then merge the arrays.

T(n)=T(n/2) + n. T(n/2) is because we're dividing our subproblem in half each time, and n because there's an n amount of steps at worst for the median of median algorithm to get the median, plus some array rearrangement. The n represents some number of constant operations performed n times.

This works out to the form: $T(n)=(n/2^i)+i*n$.

The cause when the recurrence stops occurs when $n/2^i = n/k \Rightarrow 2^i=k \Rightarrow i=\log(k)$.

Then we have: T(n) = n/k +nlog(k).

nlog(k) > n/k because k >= 1.

Thus O(nlog(k))

b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n\log k)$ comparisons in the worst case.

We're still dealing with comparison based methods, so our algorithm can be described as a binary decision tree. I'm not quite sure how to count the number of leaf notes in this particular tree, however, so I can't really construct the rest of the proof.

c) Describe an algorithm that completely sorts an already k-sorted array in time $O(n\log(n/k))$.

This would only work for the algorithm that I described in a) but we could keep track of where each subdivision begins and ends as we merge them together. Then, we can do a mergesort of each subdivision, which would be n/k*log(n/k).

8. Cinderella's stepmother has newly bought n bolts and n nuts. The bolts and the nuts are of different sizes, each of them is from size 1 to size n. So, each bolt has exactly one nut just fitting it. These bolts and nuts have the same appearance so that we cannot distinguish a bolt from another bolt, or a nut from another nut, just by looking at them. Fortunately, we can compare a bolt with a nut by screwing them together, to see if the size of the bolt is too large, or just fitting, or too small, for the nut. Cinderella wants to join the ball held by Prince Charming. But now, her wicked stepmother has mixed the bolts, and mixed the nuts, and tells her that she can go to the ball unless she can find the largest bolt and the largest nut in time. An obvious way is to compare each bolt with each nut, but that will take n^2 comparisons (which is too long). Can you help Cinderella to find an algorithm requiring only $o(n^2)$ comparisons? For instance, O(n) comparisons?

Keeping the nuts and bolts in separate lists. (N for nut and B for bolt) Compare B[i] to N[i], from 1..n. There are three possibilities, during a comparison:

1. Bolt > Nut:

Toss the nut, keep the bolt. Place the bolt in a new bolt list.

2. Bolt < Nut

Toss the bolt, keep the nut. Place the nut in a new nut list.

3. Bolt = Nut:

We are unable to count out either nut or bolt, as it’s possible that this could be the largest pair. Put the nut and bolt in the new lists. If an equality does occur, we’ll want to ensure that we don’t compare the nuts and bolts again in the next pass. So, in the next pass, we will need to begin comparing in the order of B[1] = N[n-however many nuts are gone]. Basically, starting from left to right with bolts and right to left with nuts.

Repeat. If one list is less than the other, than just make comparisons using the last nut or bolt, whichever runs out first.

The algorithm ends when we have one pair left – the largest pair.

This algorithm is deterministic, because there are two possibilities in the arrangement of the lists: One where the comparisons result in eliminations, or one where it results in all equivalences. In the case of all equivalences, we know there will be eliminations in the next pass because we’re reversing the order comparisons are made against nuts.

The running time of this algorithm is a little difficult to determine, because the size of n is decreasing by a variable amount on each pass. But I’m pretty sure that it’s $o(n^2)$ since eliminations are guaranteed to occur, such that it’s better than comparing every nut with every bolt.