CS 4349 Lecture—August 30th, 2017

Main topics for #lecture include #recurrances.

Prelude

- I will hold an extra office hour Friday, September 1st from 3:00pm to 4:00pm.
- Please fill out prerequisite forms if you haven't already. The prerequisites are CS 3305 with a grade of C or better, and (CE 3345 or CS 3345 or SE 3345 or TE 3345).
- Monday, we discussed recursion, the Tower of Hanoi, and Merge-Sort. Do you have any questions?

Solving Recurrences

- Last time, I introduced you to recurrences as a first step in determining the running time of recursive algorithms.
- For example, if T(n) is the worst-case running time of Merge-Sort, we saw T(n) = T(\floor{n/2}) + T(\ceil{n/2}) + Theta(n)
- We then gave an upper bound on T(n) of O(n log n). Using a similar proof, we could have shown a lower bound as well to *solve* the recurrence. T(n) = Theta(n log n).
- Our proof was a based on guessing a solution and checking it, otherwise called the *substitution method*.
- Today, we'll learn a few more techniques for solving recurrences you are likely to see when designing algorithms.

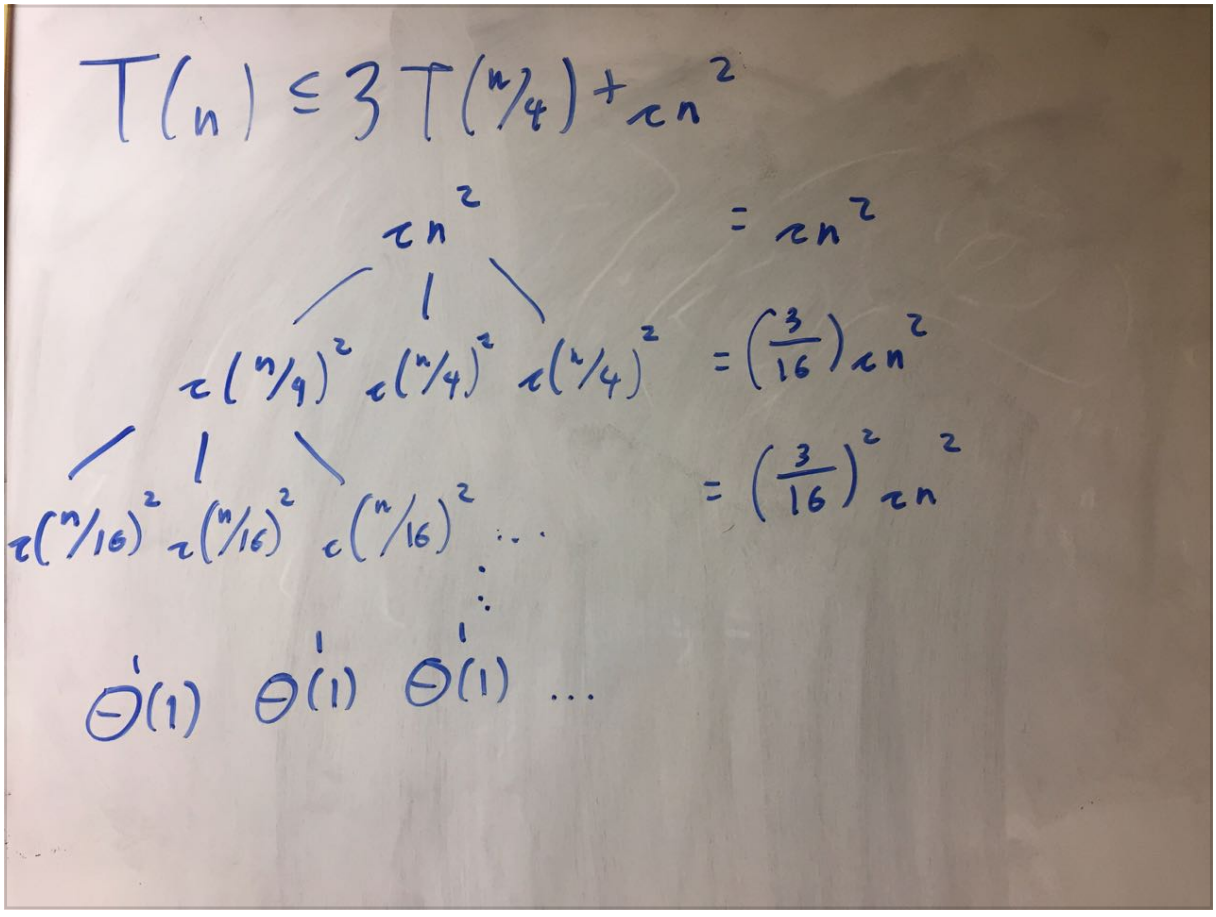
Transformations

- The first method we'll look at is changing variables or performing transformations.
- Let's try to find an asymptoticly tight solution to T(n) = 2T(\floor{\sqrt{n}}) + lg n with T(n) = Theta(1) for all n at most some constant c.
- For simplicity, we'll assume \sqrt{n} is always an integer.
- This kind of looks like the Merge-Sort recurrence, but not quite. For Merge-Sort, we had a variable drop by half inside the recursive use of T.
- But \sqrt{n} is like an *exponent* dropping by half. And lg n is like taking the value of that exponent.
- So set m so that n = 2^m. We have m drop by half inside the recursive use.
- m = lg n. We have T(2^m) = 2T(2^{m/2}) + m.
- Or if we want to write this using functions of m directly, we set S(m) = T(2^m). Solving, we see S(m) = 2S(m/2) + m.
- Oh, but we already solved this one! S(m) = Theta(m lg m).

- Which implies T(n) = T(2^m) = S(m) = O(m log m) = O(lg n lg lg n).
- Similar to guessing, figuring out which substitutions may be useful requires a bit of practice.

Recursion Trees

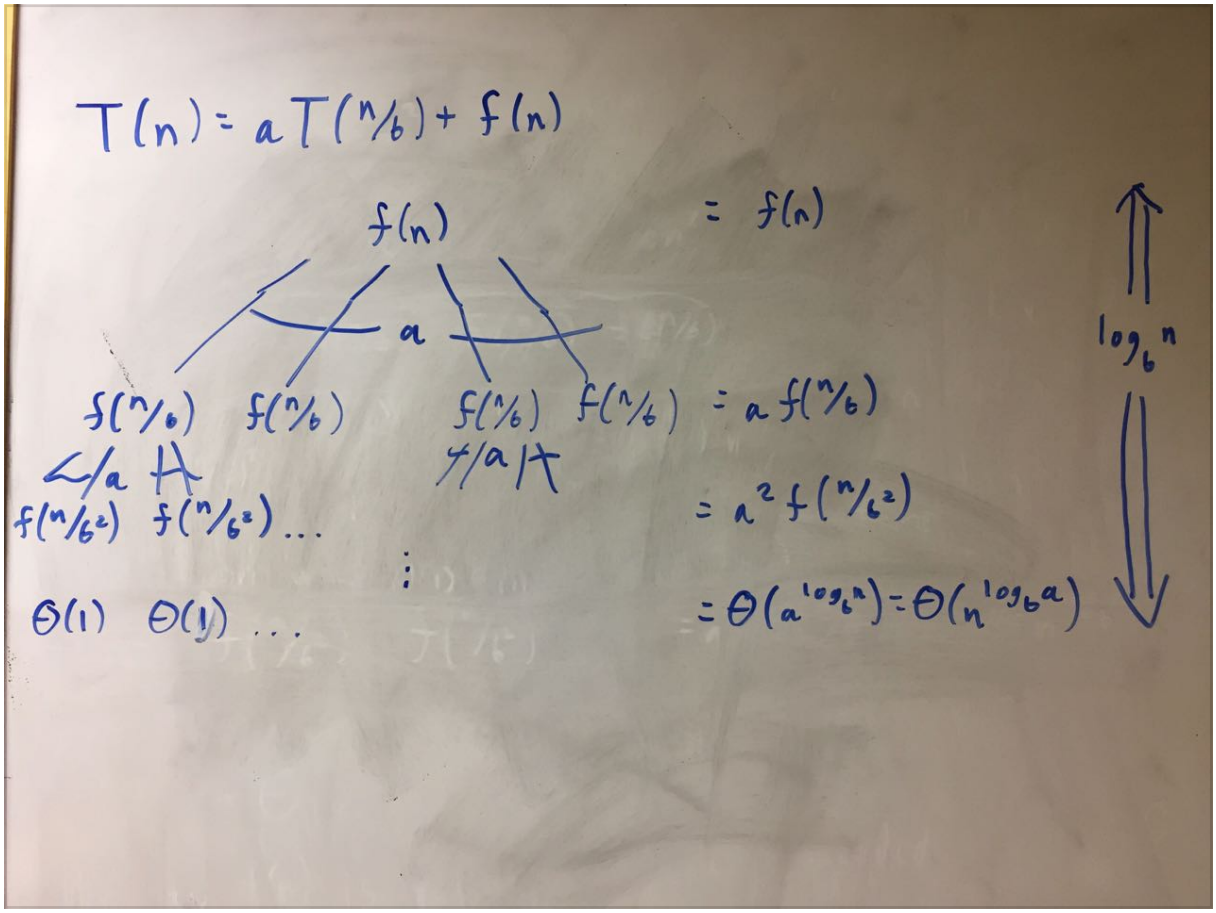
- But doing transformations to get the Merge-Sort recurrence can only take us so far. Not every algorithm looks like Merge-Sort.
- Say we want to solve the recurrence T(n) = 3T(n/4) + Theta(n^2) where T(n) = Theta(1) for all n ≤ n0. For an upper bound, we'll solve T(n) ≤ 3T(n/4) + cn^2.
- We'll use something called a recursion tree.



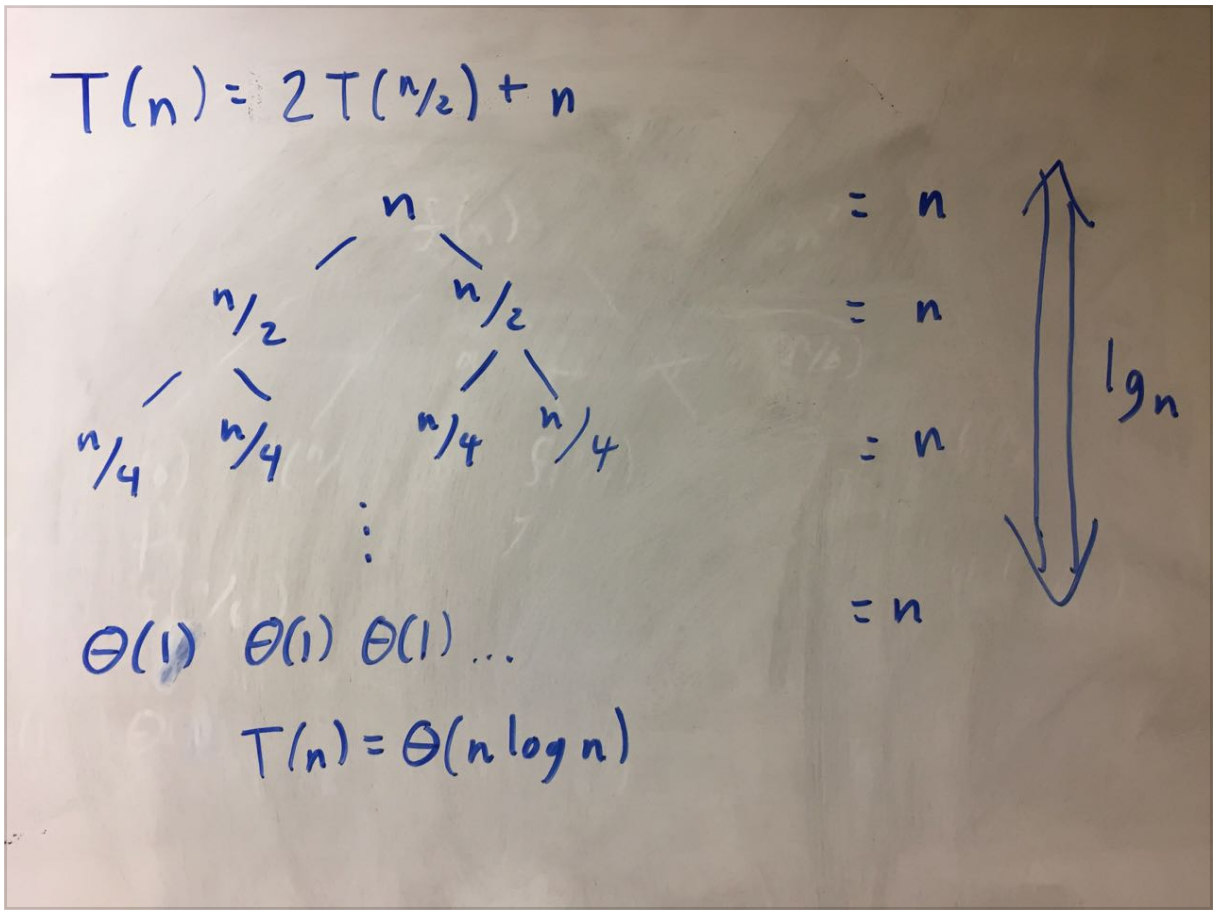
- At the root we write the time spent doing work outside of the recursion, so cn^2.
- The root has three children, representing the three recursive calls. Each of these does c(n/4)^2 work, so we write that value down.
- In general, each node represents a problem size n' and the non-recursive part of T(n'). We write down this non-recursive part. If the problem size is bigger than the base case, we give the node three children of the smaller size used in recursion.
- The leaves all get Theta(1) since T(n) = Theta(1) at the base cases.
- To solve for T(n), we simply need to add up all the non-recursive parts. It's easiest to do this row by row in the recursion tree.

- In this case, we see the sums at each row go in a decreasing geometric series. The biggest term dominates, so T(n) = O(cn^2) = O(n^2).
- We could use the same argument to show T(n) = \Omega(n^2) or just observe that the top level call alone takes \Omega(n^2) time.

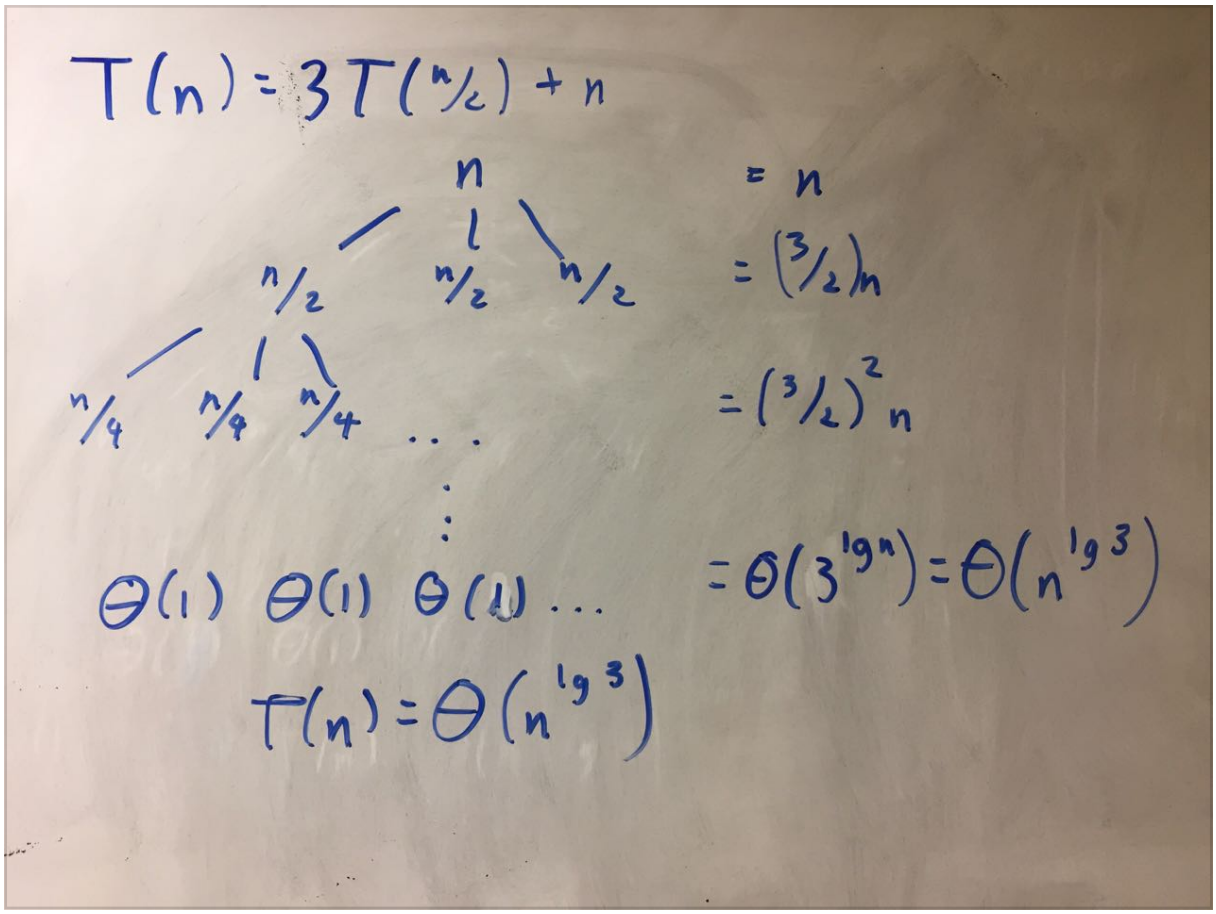
- Usually, you'll see recurrences of the form T(n) = a T(n/b) + f(n).



- For these, each node except for the leaves has *a* children. So there are a^i nodes at depth i.
- Nodes at depth i each contain the value f(n / b^i). The sum of the row at depth i is therefore a^i f(n / b^i).
- The depth of the tree itself is log_b n, and therefore T(n) = \sum_{i=0}^{log_b a} a^i f(n / b^i).
- So in merge sort for example, T(n) = 2T(n/2) + cn so T(n) = \sum_{i=0}^lg n 2^i cn / 2^i = \sum_{i=0}^lg n cn = c n lg n = \Theta(n log n).

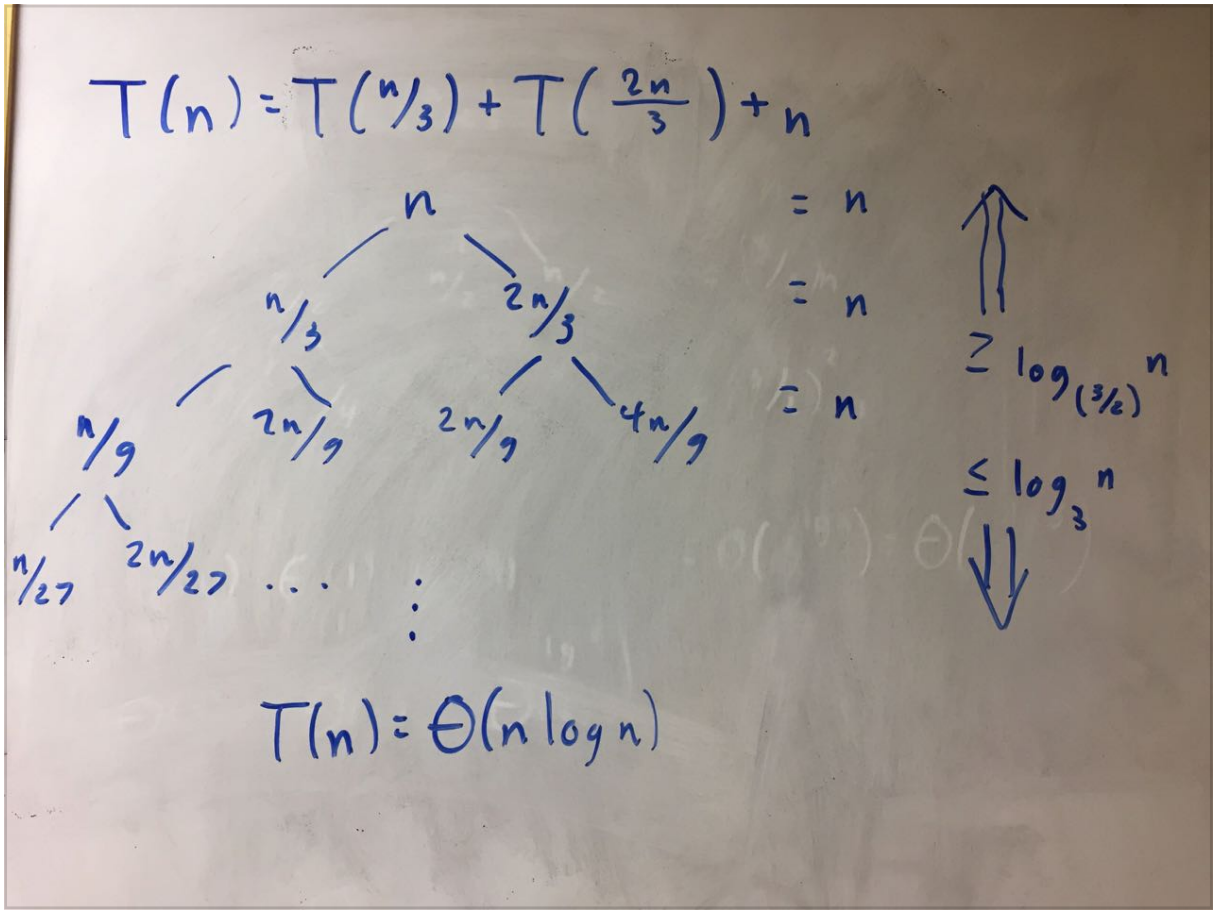


- Or consider T(n) = 3T(n/2) + n with T(1) = 1.



- Now, T(n) = \sum_{i=0}^lg n 3^i n / 2^i. This is an *increasing* geometric series, so it is dominated by its last term. It's easiest to see the last term by noting the leaf row dominates the sum of row costs in the recursion tree. There are 3^lg n = n^lg 3 leaves, each contributing 1 to the row cost, so T(n) = Theta(n^lg 3).

- We can even solve slightly weirder looking recurrences using recursion trees.
- For example, T(n) = T(n/3) + T(2n/3) + n.



- Each node now has two children, one for each recurrent use of T, but the children have different values.
- But each complete level of the tree still sums to n.
- The lowest leaf is at depth log_(3/2) n, so T(n) ≤ n log_(3/2) n = O(n log n).
- And the highest leaf is at depth log_3 n, so T(n) ≥ n log_3 n = \Omega(n log n).
- In other words, T(n) = Theta(n log n)

Master Theorem

- So you'll sometimes have these weird looking recurrences, but most have the form T(n) = a T(n/b) + f(n); i.e., you're analyzing an algorithm with a recursive subproblems, each of size n / b, with a combine step that takes f(n) time. This form is so common, that there is even a

theorem to explain all the cases.

- There's a few versions of this theorem. I'll give a version similar to the one Erickson describes since it's easier to remember, covers most of the useful cases, and I can actually prove it during lecture. The textbook describes a more powerful version if you're interested.
- Master theorem: Let a ≥ 1, b > 1, and d ≥_ 0 be constants, and let f(n) = Theta(n^d). Let T(n) = a T(n / b) + f(n).

1. If a (n / b)^d = k n^d for some constant k < 1, then T(n) = Theta(f(n)).
2. If a (n / b)^d = K n^d for some constant K > 1, then T(n) = Theta(n^{log_b a}).
3. If a (n / b)^d = n^d, then T(n) = Theta(f(n) log n)

- Most anything you can do with the master theorem, you can do more directly with recursion trees. So memorize it if you really want to, but don't worry if you can't. I'll usually just stick to recursion trees, myself.
- In fact, looking at recursion trees is enough to prove the theorem.

1. In this case, a^i f(n / b^i) = k^i f(n). But since k < 1, the level sums in the recursion tree form a decreasing geometric series. The sum of a geometric series with constant ratio is a constant times its largest member. The root level dominates, so T(n) = Theta(f(n)).
2. In this case a^i f(n / b^i) = K^i f(n). K > 1, so now we have an increasing geometric series. The *leaf* level dominates. We have a^log_b n = n^{log_b a} leaves, each of constant value. So T(n) = Theta(n^{log_b a}).
3. Now each row has the same sum of f(n). There are Theta(log_b n) rows, so T(n) = Theta(f(n) log n).

- Again, you don't need to memorize the theorem since the proof is just straightforward use of recursion trees, but it might makes things quicker and easy if you do. Examples:
- T(n) = 9T(n/3) + n. Here, 9 * (n / 3) = 3 * n, so the second case applies. T(n) = Theta(n^{log_3 9}) = Theta(n^2)
- T(n) = T(2n/3) + 1. Here, 1 * (1) = 1 * 1, so the third case applies. T(n) = Theta(1 * log n) = Theta(log n).
- T(n) = 2T(n/4) + n. Here, 2 * (n / 4) = (1 / 2) * n, so the first case applies. T(n) = Theta(n).
- One last one for today. T(n) = T(n / 4) + lg^2 n. Well, we can't use the theorem directly since n lg^2 n does not equal Theta(n^d) for any d. But for an upper bound, T(n) ≤_ lg^2 n S(n) where S(n) = 3S(n/4) + 1. 1 * 1= 1 * 1, so the third case applies and S(n) = O(log n) and T(n) = O(log^3 n).

CLRS Master Theorem

- More general master theorem: Let a ≥ 1 and b > 1 be constants, let f(n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence
  T(n) = aT(n / b) + f(n),

where we interpret n/b to mean either floor{n / b} or ceil{n / b}. Then T(n) has the following asymptotic bounds:

1. If f(n) = O(n^{log_b a - eps}) for some constant eps > 0, then T(n) = Theta(n^{log_b a}).
2. If f(n) = Theta(n^{log_b a}), then T(n) = Theta(n^{log_b a} log n)
3. If f(n) = Omega(n^{log_b a + eps}) for some constant eps > 0, and if a f(n / b) ≤_ c f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Theta(f(n)).