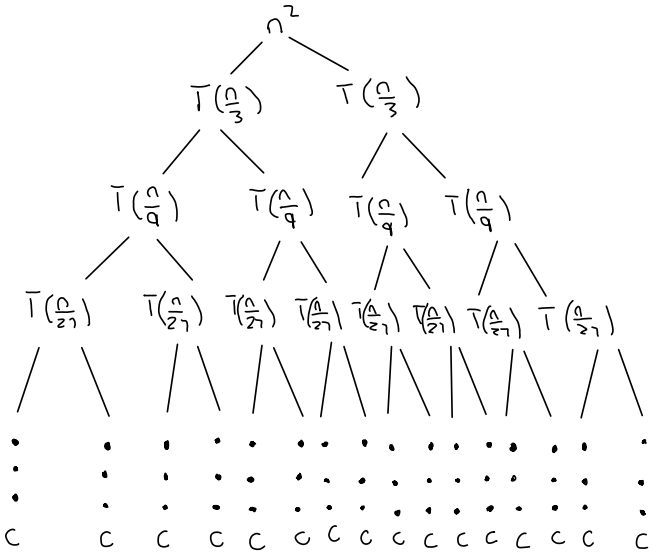
1. Consider the recursion $T(n) = 2T\left(\frac{n}{3}\right) + n^2$

(a) Draw the recursion tree



(b) How much work is there at the root of the tree? (complexity at the root)

The complexity at the root is n^2. This is the work done before any recursion happens, so it is just a one-time cost.

(c) What is the height of the tree? Explain your answer.

The height of the tree is $\log_3(n)$ since the recursion is of the form T(n) = aT(n/b) + f(n). When we have a recurrence of this form the height is given by $\log_b(n)$, and in this case b = 3 so we know our height is $\log_3(n)$.

(d) How much work is there at the leaves of the tree (complexity at the leaves). Explain.

We can see that each time we go down a level, we are incrementing our power of 3 in the denominator (in the first row our denominator is 3^1, in the second row it is 3^2, in the third row it is 3^3, etc.). We can also see that each time we go down a level we are doubling the amount of leaf nodes we have (the first row has 2, second has 4, third has 8, etc.). In order to determine the total work at any given row, we multiply the number of nodes by the work at each node. At row k, we know that we must have 2^k nodes (this is clear as $2^1 = 2$, $2^2 = 4$, $2^3 = 8$; we are simply doubling the amount of nodes we had in the previous level each time). The work at each node for level k is given by $n/3^k$, which we can clearly see in our tree ($n/3^1 = n/3$, $n/3^2 = n/9$, $n/3^3 = n/27$, etc.). Therefore, we can say that the total work at any generic level k is given by the equation $(2^k)(n/3^k)$.

(e) Solve the recurrence using the Master Theorem. Explain the connection to what you computed in (b) and (d) regarding more work at the root or leaves.

Using the Master Theorem, we will compare f(n) and $n^{(\log_b a)}$ with regard to our generic form of T(n) = aT(n/b) + f(n). Our equation in this example is T(n) = 2T(n/3) + n^2, which means that f(n) = n^2 and $n^{(\log_b a)} = n^{(\log_3 2)} = n^{(0.63092975357)}$. We can see that in this case f(n) = n^2 is polynomially larger than $n^{(\log_b a)} = n^{(0.63092975357)}$, so f(n) dominates $n^{(\log_b a)}$. According to the Master Theorem, this means that T(n) = Θ(f(n)) = n^2, and so this is the solution to this recursion example. This relates to the previous questions in that it is a shortcut to our answer: rather than drawing out the recursion tree and using it to calculate the work at the root and leaves, we can just use the form T(n) = aT(n/b) + f(n) and the Master Theorem to solve our recurrence. Without the Master Theorem, we would have had to do a summation of the work at each level ($(2^k)(n/3^k)$) from 0 to the height of our tree ($\log_3(n)$). Drawing the tree, determining the height and work at each level, and calculating this summation is a lot more work than just using the Master Theorem.

---

2. Strassen in his matrix multiplication problem found a way to compute matrix multiplication with only 7 recursions. We calculated the complexity using the master theorem in class.
(a) What would be the complexity if someone clever found a way to compute matrix multiplication with only 5 recursions? Write out the recursive equation and solve with the Master Theorem.

With 7 recursions in Strassen's method, we have the equation T(n) = 7T(n/2) + n^2, but if we somehow found a way to do it in 5 recursions instead then our equation would be T(n) = 5T(n/2) + n^2. Applying the Master Theorem to this equation, we will compare f(n) = n^2 with $n^{(\log_b a)} = n^{(\log_2 5)} = n^{(2.3219280949)}$, and we can see that the leaves will still dominate since $n^{(\log_b a)}$ is polynomially larger than f(n) in this case. So our complexity would be $\Theta(n^{(\log_2 5)}) = \Theta(2.3219280949)$ which is a small improvement over Θ(~2.8), which was the complexity we had with 7 recursions, but this could make quite a big difference in runtime if we had really big matrices.

(b) What about with only 3 recursions? Write out the recursive equation and solve with the Master Theorem.

If we had just 3 recursions, our equation would be T(n) = 3T(n/2) + n^2. Applying the Master Theorem we compare f(n) = n^2 with $n^{(\log_b a)} = n^{(\log_2 3)} = n^{(1.5849625007)}$, and here we see a difference from before since now the root actually dominates since f(n) is polynomially larger than $n^{(\log_b a)}$. So our complexity is now Θ(f(n)) = Θ(n^2). We have reduced the number of recursions so much that now the root dominates, so if we wanted to make this algorithm even faster we would have to figure out some clever way to reduce the root cost.

---

3. Solve the following recurrences with the Master Theorem. If the problem cannot be solved with the Master Theorem state this and explain why. If the problem can be solved with the Master Theorem, also specify if there is more work at the root or the leaves (or if they are equal) in your answer. (Write your solutions in Θ.)

(a) $T(n) = 27T\left(\frac{n}{3}\right) + n^3$

f(n) = n^3
n^($\log_b a$) = n^($\log_3 27$) = 3
The work at the root and leaves is equal, so our complexity is Θ((n^3)log(n)).

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

This cannot be solved using the Master Theorem because the given recurrence equation is not of the form T(n) = aT(n/b) + f(n). Here we have two different recursions which will result in two different branches with different costs if we were to draw out a recursion tree, so the Master Method cannot be used here.

(c) $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

This also cannot be solved using the Master Theorem because f(n) is not a polynomial function.

(d) $T(n) = 5T\left(\frac{n}{2}\right) + n^2$

f(n) = n^2
n^($\log_b a$) = n^($\log_2 5$) = n^(2.3219280949)
The work at the leaves dominates since n^($\log_b a$) = n^(2.3219280949) is polynomially larger than f(n) = n^2, so our complexity is Θ(n^(2.3219280949)).

(e) $T(n) = 2T\left(\frac{n}{3}\right) + nlogn$

f(n) = nlog(n)
n^($\log_b a$) = n^($\log_3 2$) = n^(0.63092975357)
The work at the root dominates since f(n) = nlog(n) is polynomially larger than n^($\log_b a$) = n^(0.63092975357), so our complexity is Θ(nlog(n)).