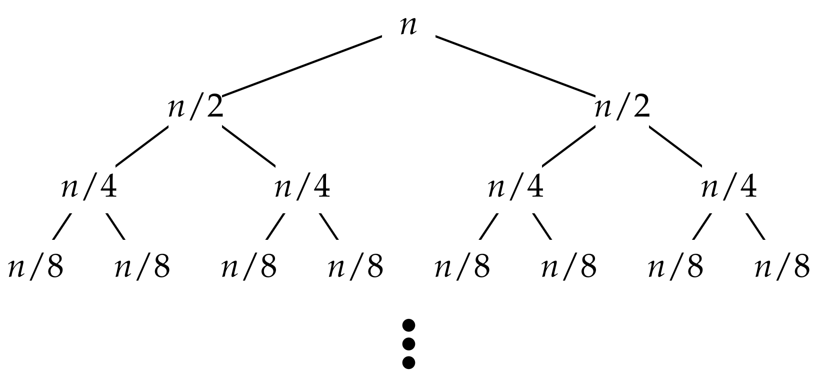
Algorithms: Introduction to Divide & Conquer

Model 1: Merge sort

```
mergesort($A$) =
    if $len(A) \leq 1$ then
        return $A$
    split $A$ into halves $(A_L, A_R)$
    $A'_L \leftarrow$ mergesort($A_L$)
    $A'_R \leftarrow$ mergesort($A_R$)
    $A' \leftarrow$ merge($A'_L$, $A'_R$) // merge is a non-recursive function that takes two sorted lists as input and yields
                          // a new sorted list that includes the contents of the original lists
    return $A'$
```

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n/2) + \Theta(n)$$



Recall the *merge sort* algorithm, which works by splitting the input list into halves, recursively sorting the two halves, and then merging the two sorted halves back together. Merge sort pseudocode is shown in Model 1.

Learning objective: Students will use recurrence relations and recursion trees to describe and analyze divide and conquer algorithms.

1 How long does mergesort take on a list of length 1?

2 Just by looking at the code, how many recursive calls does merge-sort make at each step?

Hint: don't overthink this one; yes, it's really that easy.

3 If A has size n, what are the sizes of the inputs to the recursive calls to mergesort (you can assume n is even)?

4 (Review) If A has size n, how long does it take (in big-Θ terms) to merge A_L and A_R after they are sorted? In other words, what is the run-time of `merge(`A_L`,`A_R`)`?

5 Let $T(n)$ denote the total amount of time taken by mergesort on an input list of length n. Use your answers to the previous questions to explain the equations for $T(n)$ given in the model. This is called a *recurrence relation* because it defines $T(n)$ via recursion.

6 Suppose some algorithm X takes an input of size n, splits the input into three equal-sized pieces, and makes a recursive call on each piece. Deciding how to split up the input into pieces takes $\Theta(n^2)$ time; combining the results of the recursive calls takes additional $\Theta(n)$ time. In the base case, algorithm X takes constant time on an input of size 1. Write a recurrence relation $X(n)$ describing the time taken by algorithm X, similar to the one given in the model.

7 Now suppose algorithm X makes only two recursive calls instead of three, but each recursive call is still on an input one-third the size of the original input. How does your recurrence relation for X change?

8 Write a recurrence relation for binary search.

Analyzing Mergesort's Recursion Tree

Now let's return to considering merge sort. The tree shown in the model represents the call tree of merge sort on an input of size n, that is, each node in the tree represents one recursive call to merge sort. The expression at each node shows how much work happens at that node (all of the non-recursive work, which for mergesort is the work of the `merge` routine). Note that this work in expressed in terms of the problem size *in that particular recursive call*.

9 Notice that the entire tree is not shown; the dots indicate that the tree continues further with the same pattern. What is the depth (number of levels) of the tree, in terms of n?

10 How much total work happens on each individual level of the tree?

11 How much total work happens in the entire tree?

12 Draw a similar tree for the second version of algorithm X.

Note. The strategy of expressing a recurrence relation in tree form and analyzing the recursion tree's dimensions (e.g., nodes per level, number of levels, work per node) is not the only way to justify the total cost of a recursive algorithm. In fact, if we were to continue that style of analysis for the algorithm above, we'd see that the algebra is quite messy. Nonetheless, the recursion tree method is very effective.

We will consider additional techniques later in this class, and you may always use any justifiable approach that produces a correct result.

A Representative Problem

Please consider the following problem as representative of what you may find on the exam. You may work on it as part of the activity period, collaborate with peers *without restriction*, ask questions in office hours, or talk to the TAs. This question is not graded, so you may discuss it as openly as is helpful for your learning.

13 Consider an algorithm that solves problems of size n by dividing them nine subproblems of size $n/3$, recursively solving each subproblem, and combining the solutions in (n^2) time.

Please provide a recurrence relationship for this algorithm, and solve the recurrence giving as tight a Big Oh bound as possible. You must justify your answer using a method we have discussed in class— e.g., if using the recursion-tree method, draw the first few levels of the tree and explain why that leads to the time bound. You *do not* need to verify by induction (unless you are using the guess and check method, in which case you do need a proof).