CS 161 (Stanford, Winter 2025) Lecture 3

Pre-lecture exercises will not be collected for credit. However, you will get more out of each lecture if you do them, and they will be referenced during the lecture. We recommend writing out your answers to pre-lecture exercises before class. Pre-lecture exercises usually should not take you more than 30 minutes.

Pre-Lecture Exercises

In this pre-lecture exercise, you'll explore *recurrence relations.* A recurrence relation defines a function $T(n)$ recursively. For example, for $n = 2^i$ which is a power of 2, we might define:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + n & n > 1 \\ T(n) = 1 & n = 1 \end{cases}.$$

Why is a function like this relevant to us? It turns out that it is a good way to write down the running time of divide-and-conquer algorithms. For example, we saw with MergeSort that we broke up one problem of size n into two problems of size $n/2$; and then it took us an extra $O(n)$ operations to merge the solutions. Let's say for concreteness that it takes $11n$ operations, where the 11 is arbitrary. So if $\tilde{T}(n)$ is the number of operations it takes to run MergeSort on a list of size n, we could write something like

$$\tilde{T}(n) = 2\tilde{T}(n/2) + 11 \cdot n,$$

which is similar to the function $T(n)$ above. The way to interpret the base case $T(1) = 1$ is that it takes one operation to sort a list of length 1 since we just return it; it's already sorted. We saw in class that $\tilde{T}(n) = O(n \log(n))$. That is, the running time of MergeSort is $O(n \log(n))$.

Exercise 1

The first problem in the pre-lecture exercise is to understand the above text and make sure you understand the connection between the function $\tilde{T}(n)$ defined above to the running time of MergeSort.

For the rest of the pre-lecture exercise, you'll see if you can generalize the argument that we saw in class to different recurrence relations. For reference, **on the next page we've shown two different ways of showing that $T(n) = O(n \log(n))$.** (We went with $T(n)$ instead of $\tilde{T}(n)$ because it's a little bit cleaner to write down without carrying that factor of "11" around everywhere, and the point still gets across.) If you found the tree method confusing, you

might like the second method which just uses algebra.

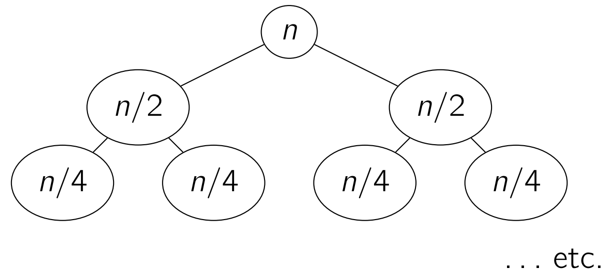
Exercise 2

Now try to generalize the argument for $T(n)$ to the following two recurrence relations. That is, for each of $T_1(n)$ and $T_2(n)$ below, figure out an expression of the form $T_j(n) = O(\cdot)$. **Assume that n is a power of 2 if it helps.**

- $T_1(n) = \begin{cases} T_1(n/2) + n & n > 1 \\ T_1(n) = 1 & n = 1 \end{cases}.$
- $T_2(n) = \begin{cases} 4 \cdot T_2(n/2) + n & n > 1 \\ T_2(n) = 1 & n = 1 \end{cases}.$

Here are two different ways to understand the running time of $T(n)$, when n is a power of 2:

SOLUTION 1. Just like we did in class, imagine a tree with $\log(n)+1$ levels. The top node is labeled "n", its two children are labeled "$n/2$", and so on.



Consider $T(n) = T(n/2) + T(n/2) + n$. In the context of the tree above, that means that $T(n) = n +$ (stuff contributed by things in the tree lower than the root). That is,

$$T(n) = (\text{label on the root}) + (\text{stuff contributed by things lower than the root}).$$

We can repeat this logic recursively to figure out what that second term is, all the way down to the bottom of the tree, where we have $T(1) = 1$. We conclude that each node in the tree that's labeled k contributes k to the sum.[1] Now we add everything up:

- The zeroth layer contributes n, since there is one problem of size n, which contributes n.
- The first layer also contributes n, since there are two problems of size $n/2$, each of which contributes $n/2$.
- $\cdots$
- The t'th layer also contributes n, since there are 2^t problems of size $n/2^t$, each of which contributes $n/2^t$.
- $\cdots$
- The $\log(n)$th layer (which is the bottom one) also contributes n, since there are n problems of size 1, each of which (by the base case $T(1) = 1$) contributes 1.

Altogether there are $\log(n) + 1$ layers, each contributing n, so we conclude that, when n is a power of 2,

$$T(n) = n(\log(n) + 1).$$

Notice that this is an exact answer when n is a power of 2, we don't even need a $O(\cdot)$.

[1] Notice that this is a special consequence of the fact that the term we are adding in the definition of $T(n)$ is exactly n; if it were, say $11 \cdot n$, the contribution of a node labeled k would be $11k$.

SOLUTION 2. We can do the same calculation without the tree, by repeatedly applying our formula.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 4(2T(n/8) + n/4) + 2n \\ &= 8T(n/8) + 3n \end{aligned}$$

and at this point we can spot the pattern: for all $j \leq \log(n)$,

$$T(n) = 2^j T(n/2^j) + jn.$$

To formally prove that this is true, we should use a proof by induction; that's called the *substitution method* and we'll talk about it soon. But for now, you can convince yourself that this is true.

Once we have this, we can just plug in $j = \log(n)$, and get

$$T(n) = 2^{\log(n)} T(n/2^{\log(n)}) + n\log(n) = n \cdot T(1) + n\log(n) = n(\log(n) + 1),$$

just as before.