Recurrences Review

Overview

This document is intended to review different aspects of recurrences as we covered them in lecture and section. When we discuss Recurrences in this course, we tend to focus on three different topics: The **Recurrence** itself, the **Closed Form** of a recurrence, and **Asymptotic Analysis** of recurrences. We will spend some time on each of these topics in this document, with the intention of providing clearer differentiation between them.

The Recurrence itself

In general terms, a Recurrence is a piecewise function representing a certain aspect of a function; we have thus far focused on runtime (the number of "steps" an algorithm takes), but we could model recurrences on other aspects, including output (what the algorithm returns). This document will continue to focus on recurrences for runtime.

When we are tasked with writing a recurrence for a function, our first step is to figure out which parts of the function represent base cases and which represent recursive cases; then, we figure out the runtime of the base cases and add conditions to our piecewise function for each base case. Our next step is to find out how much "recursive work" each of our recursive cases do (where "recursive work" is just the recursive function calls), and log that in our recurrence. Finally, we analyze how much nonrecursive work our recursive cases do and record that in our recurrence. Putting everything together, a recurrence will look something like this:

$$T(n) = \begin{cases} \text{[base case]} & \text{(condition)} \\ \quad\vdots & \\ \text{[base case]} & \text{(condition)} \\ \text{[recursive case]} & \text{(condition)} \\ \quad\vdots & \\ \text{[recursive case]} & \text{(condition)} \end{cases}$$

Note that the lecture slides presented recurrence functions in a way that "looked different"; specifically, like this:
T(n) = [base case] when (condition)
⋮
T(n) = [base case] when (condition)
T(n) = [recursive case] when (condition)
⋮
T(n) = [recursive case] when (condition)
But these are actually equivalent ways to express T(n), and you may choose to use either.

Closed Forms

Once we've found a recurrence, we may be asked to find a closed form. A closed form for a recurrence for runtime is an **exact** representation of the number of steps the function takes. A recurrence itself is **not** a closed form, nor is an expression with a summation. To find a closed form, we have two strategies to choose from (note we will assume our recurrence only has one recursive case in these situations, as having more than one makes finding a closed form more complicated):

(a) Unrolling a recurrence is a strategy we typically use when the recursive case only has one recursive call. That is, if the recursive case looks like $T(n) = T(n/2) + $[nonrecursive work], for example. When we unroll, we write out the recursive case of our recurrence, substitute the recursive call to T with it's recursive case, and keep going until we can detect a pattern. Consider the following example:

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 0 \\ T(n-1) + c_1 n + c_2 & \text{otherwise} \end{cases}$$

We would start unrolling like so:

$$\begin{aligned} T(n) &= T(n-1) + c_1 n + c_2 \\ &= T(n-2) + c_1(n-1) + c_2 + c_1 n + c_2 \\ &= T(n-3) + c_1(n-2) + c_2 + c_1(n-1) + c_2 + c_1 n + c_2 \\ &= \ldots \end{aligned}$$

From here, we could start to detect a pattern: we know we will have n recursive terms, plus c_0 (for the non-recursive work once we reach the end), so we know there will be n c_2 terms, as well as $\sum_{i=1}^{n} c_1 * i$ work from all the c_1 terms; so our recurrence is thus transformed into:

$$T(n) = c_0 + \sum_{i=1}^{n} c_1 * i + c_2 n$$

Once we apply the first closed form in your summation sheets, we get our final closed-form representation of $T(n)$:

$$T(n) = c_0 + c_1\left(\frac{n(n+1)}{2}\right) + c_2 n$$

It's important to note that, you may be tempted to try to simplify an expression (especially if it's particularly ugly). **Don't**. The moment you've eliminated any summations from your expression, you've found a closed form; simplification increases the risk of making typos or other minor mistakes.

(b) The other method we could use to find a closed form is by building up a recurrence tree. This is the strategy we commonly use when our recurrence has more than one recursive call. If our recurrence makes k recursive calls, this method involves envisioning a tree where each node represents the nonrecursive work done at that step in the recurrence and has k child nodes which each represent the next step in the recurrence. The leaf nodes of the tree represent the base case of the recurrence. Let's consider an example:

$$T(n) = \begin{cases} c_0 & \text{if } n \leq 1 \\ 2T(n/2) + c_1 n + c_2 & \text{otherwise} \end{cases}$$

The root node of the recurrence tree represents $T(n)$; it has $c_1 n + c_2$ work and 2 children, both representing $T(n/2)$. The children both have $c_1(n/2) + c_2$ and two children of their own representing $T(n/4)$, and so on. All the way at the bottom level are the base-case nodes, each with c_0 work and 0 children. You may want to consider drawing this tree out as a personal guide while we walk through the task of counting the work.

To calculate the amount of work done in this tree, we reason like so: we split n in half every time we go down a level in the tree, so the height of the tree is $log_2(n)$. If we consider the root node level 0, its children level 1, their children level 2, and so on, then we can express the work in an arbitrary node at level i as $c_1(\frac{n}{2^i}) + c_2$, and we can say that each level in our tree has 2^i nodes. Note that the base-case work cannot be represented in terms of i, so we have to handle that differently. There are 2^{height} base-case nodes, each with c_0 work, so we can express the work done in the tree as the sum over the height minus 1 of the multiple of the number of nodes at level i and the amount of work on of those nodes does, plus the amount of base case work. In this case:

$$T(n) = \left(\sum_{i=0}^{log_2(n)-1} 2^i\left(c_1 \frac{n}{2^i} + c_2\right)\right) + 2^{log_2(n)} c_0$$

We're almost there; all we have to do now is remove the summations. To do that, we need to use algebraic manipulation to be able to use one of the closed-forms on the summations sheet. Doing a bit of arithmetic, we can transform the above expression into the following:

$$T(n) = (c_1 n \sum_{i=0}^{log_2(n)-1} 1) + (c_2 \sum_{i=0}^{log_2(n)-1} 2^i) + 2^{log_2(n)} c_0$$

All the first summation is doing is adding 1 $log_2(n)$ times, and the second summation we can express using the third closed-form from our summation sheet to get the following:

$$T(n) = c_1 n log_2(n) + c_2 \frac{1 - 2^{log_2(n)}}{1-2} + 2^{log_2(n)} c_0$$

We do not need to simplify this any further. This is a sufficient closed-form for our recurrence!

Asymptotic Analysis

The third common question we may be asked regarding recurrences is to derive some asymptotic bound for the recurrence. There is one main way we do this, as well as one special method we can apply given a few certain restrictions. The main way we do this is by finding a closed form, like we did in the previous part, and then analytically determining the asymptotic bound. The special way, only applicable if we're looking for Big-O from a recurrence of a certain form, is the Master Theorem.

(a) Let's consider the unrolling recurrence from the previous part: when we look at the closed form for this recurrence, we see that the dominant term is n^2. Thus, we can say that $T(n) \in \mathcal{O}(n^2)$. It is also in $\Omega(n^2)$.

(b) The **Master Theorem** is a special way to analyze the asymptotic runtime of a recurrence. It is **only** useful for Big-O analysis, and it **only** works on recurrences resembling the following:

$$T(n) = \begin{cases} C_0 & \text{if n} \leq n_0 \\ aT(\frac{n}{b}) + n^c + n^{c-1} + \dots & \text{otherwise} \end{cases}$$

That is, it only works when the base case is a constant and the recursive case involves one or more recursive calls where n is divided by some constant > 1.
Master theorem analyzes the runtime by using the constants a, b, and c like so:

- if $log_b(a) < c$, then $T(n) \in \mathcal{O}(n^c)$
- if $log_b(a) = c$, then $T(n) \in \mathcal{O}(n^c lg(n))$
- if $log_b(a) > c$, then $T(n) \in \mathcal{O}(n^{log(a)})$

Let's consider for an example the second recurrence for which we found a closed form: That recurrence had constant base-case work and a recursive case looking like $2T(n/2) + c_1 n + c_2$. For Master Theorem, the constants we need to analyze are $a = 2$, $b = 2$, and $c = 1$. $log_2(2) = 1$, which is what c is, so by Master Theorem we know $T(n) \in \mathcal{O}(nlg(n))$. We can confirm this looking at our closed form: Of the three summed terms, the dominant term is $c_1 n log_2(n)$ (note that $2^{log_2(n)} = n$ by logarithmic properties), so the closed form must be in $\mathcal{O}(nlog_2(n))$, which is exactly what Master Theorem told us it would be.