Thursday 7/10: Recursion

Defining recursion

A *recursive* definition defines an object in terms of smaller objects of the same type. Because this process has to end at some point, we need to include explicit definitions for the smallest objects. So a recursive definition always has two parts:

- Base case(s)
- Recursive formula

For example, the summation $\sum_{i=1}^{n} i$ can be defined as:

- $g(1) = 1$
- $g(n) = g(n-1) + n$, for all $n \geq 2$

Sometimes, there might be more than one base case, and the result for n may depend on more than one smaller value. For example, here is the definition for Fibonacci numbers:

- $F_0 = 0$
- $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$, for all $i \geq 2$

Unrolling

Unrolling is the process to substitute a recursive definition into itself. For example, the recursive formula of the Fibonacci numbers in the above example can be unrolled like this:

$F_i = F_{i-1} + F_{i-2} = (F_{i-2} + F_{i-3}) + (F_{i-3} + F_{i-4}) = ...$ until you reach the base cases or find a clear pattern. Using this pattern, many recursive numerical formulas can be simplified into a *closed form*, which is an equivalent expression that does not involve recursion.

Let's see the following example. Suppose we have a function $T : \mathbb{N} \to \mathbb{Z}$ such that:

- $T(1) = 1$
- $T(n) = 2T(n-1) + 3, \forall n \geq 2$

How to find the closed form of a recursion

Step 1: Unroll for a few iterations and find the pattern

$$\begin{aligned}
T(n) &= 2T(n-1) + 3 \\
&= 2(2T(n-2) + 3) + 3 \\
&= 2(2(2T(n-3) + 3) + 3) + 3 = 2^3 T(n-3) + 2^2 * 3 + 2 * 3 + 3 \\
&= ... \\
&= 2^k T(n-k) + 2^{k-1} * 3 + ... + 2^1 * 3 + 2^0 * 3
\end{aligned}$$

The first few lines of this are mechanical substitution. To get to the last line, you have to imagine what the pattern looks like after k substitutions.

Step 2: Further simplify the formula for the k-th level

We can use summation notation to compactly represent the result of the k-th unrolling step:

$$\begin{aligned} T(n) &= 2^k T(n-k) + 2^{k-1} * 3 + 2^{k-2} * 3 + ...2 * 3 + 3 \\ &= 2^k T(n-k) + 3 * (2^{k-1} + 2^{k-2} + ...2^2 + 2^1 + 2^0) \\ &= 2^k T(n-k) + 3 \sum_{i=0}^{k-1} (2^i) \end{aligned}$$

Step 3: Plug in base case(s)

The formula we have in step 2 uses $T(n-k)$, which is still an unknown value until it hits the base case. Therefore, we want $T(n-k)$ to leverage our base case, which is $T(1)$. In other words, let's set $n-k=1$, so that $k=n-1$. Then, we can plug in this value to the formula we calculated in step 2:

$$\begin{aligned} T(n) &= 2^k T(n-k) + 3 \sum_{i=0}^{k-1} (2^i) \\ &= 2^{n-1} T(1) + 3 \sum_{i=0}^{n-2} (2^i) \text{ because } k = n-1 \\ &= 2^{n-1} + 3 \sum_{i=0}^{n-2} (2^i) \text{ because } T(1) = 1 \\ &= 2^{n-1} + 3(2^{n-1} - 1) = 4(2^{n-1}) - 3 = 2^{n+1} - 3 \end{aligned}$$

So the closed form for this function is $T(n) = 2n + 1 - 3$. Note that the unrolling process isn't a formal proof that our closed form is correct.

Let's try to unroll and analyze another recursive definition:

$$\begin{aligned} &S(1) = c \\ &S(n) = 2S(\frac{n}{2}) + n, \forall n \geq 2 \end{aligned}$$

Then we have

$$\begin{aligned} S(n) &= 2S(\frac{n}{2}) + n \\ &= 2(2S(\frac{n}{4}) + \frac{n}{2}) + n = 4 * S(\frac{n}{4}) + 2 * \frac{n}{2} + n \\ &= 8 * S(n/8) + n + n + n \\ &... \\ &= 2^i S(\frac{n}{2^i}) + i * n \end{aligned}$$

Note that in this case, we are reducing the size of subproblems in half each time. Therefore, we hit the base case when $\frac{n}{2^i} = 1$, i.e. when $i = \log n$. Now plugging in the value of i, we get

$$S(n) = 2^i S(\frac{n}{2^i}) + i * n = 2^{\text{log}n} * c + n * \text{log}n = cn + n\text{log}n$$

Note how this closed form is different from the Fibonacci one due to how the subproblems are reducing differently.