Recursion and Structural Induction

Benjamin Cosman, Patrick Lin and Mahesh Viswanathan

Fall 2020

TAKE-AWAYS

- Functions and sets can be defined *recursively*
- Recursive function definitions are especially convenient when the domain is a recursively-defined set
- *Structural induction* is a variant of induction used for recursively-defined sets

Recursively-defined functions and sequences

Some functions and sequences are most naturally defined recursively, i.e. in terms of themselves. For example, the Fibonacci sequence is $0, 1, 1, 2, 3, 5, 8, 13, \cdots$, where the first two items are specified explicitly as 0 and 1, and every subsequent item is the sum of the previous two. More formally, we can write this definition as follows:

$$f_n = \begin{cases} 0 & \text{when } n = 0 \\ 1 & \text{when } n = 1 \\ f_{n-1} + f_{n-2} & \text{when } n \geq 2 \end{cases}$$

Similarly, the factorial function "!", which is defined on positive integers as the product of the given number with all the smaller positive integers, can be written as follows: [1]

$$n! = \begin{cases} 1 & \text{when } n = 1 \\ n \cdot (n-1)! & \text{when } n \geq 2 \end{cases}$$

This should look like a familiar "piecewise-defined" function, except that some of the pieces will make "recursive calls" to the function itself. It is also very similar to how recursive functions work in whatever programming language you are familiar with.

[1] Factorial is also frequently defined on 0 using the base case that $0! = 1$; we don't need that yet for our purposes.

How NOT to write recursive functions

Like any function, a recursively-defined function $f : A \to B$ must assign exactly one element of B to each element of A. So if we are trying to define an $f : \mathbb{N} \to \mathbb{N}$, the following is NOT a valid definition because $f(0)$ has no value:

$$f(n) = \begin{cases} 3 & \text{when } n = 1 \\ n + f(n-1) & \text{when } n \geq 2 \end{cases}$$

and the following is NOT a valid definition because $f(2)$ has two different values:

$$f(n) = \begin{cases} 1 & \text{when } n \leq 2 \\ n \cdot f(n-1) & \text{when } n \geq 2 \end{cases}$$

A more subtle version of this issue can arise for recursively-defined functions when the recursive "calls" never actually reach a base case, e.g. by skipping over the base case(s) or by making recursive calls that don't progress towards a base case at all. For example, the following is NOT a valid function on any domain that includes 1, because $f(1)$ is defined in terms of $f(-1)$, which is undefined:

$$f(n) = \begin{cases} 1 & \text{when } n = 0 \\ n \cdot f(n-2) & \text{when } n \geq 1 \end{cases}$$

Similarly, the following is NOT valid because the recursive calls would go on upwards forever[2], so $f(1)$ does not have any actual value:

$$f(n) = \begin{cases} 1 & \text{when } n = 0 \\ f(n+1) & \text{when } n \geq 1 \end{cases}$$

[2] more formally, the definition isn't valid because it doesn't specify a *unique* function: *any* function which maps 0 to 1 and all positives to the same constant would fit the rule described

Recursively-defined sets

Recall that we have seen two ways of defining sets so far:

- "Roster notation", where elements are listed explicitly (e.g. $\{3, 4, 8\}$, or, somewhat informally because of the $\cdots$, $\{0, 2, 4, \cdots\}$ to represent the non-negative evens).
- Set-builder notation, where we specify a rule for which elements are included (e.g. $\{n \in \mathbb{N} \mid n \text{ is even}\}$)

Sets can also be defined using recursion. For example, here is a different way of specifying a set S containing all the non-negative even numbers:

"Define S recursively by

- $0 \in S$, and
- If $a \in S$, $a + 2 \in S$."

The first bullet point, specifying certain initial elements in S, is called the base case (or base step), and the second bullet point, specifying how to create more elements given ones that already exist, is called the recursive or constructor case (or step). A recursively-defined set contains each element specified in the base case, each element that can be constructed by any (finite) number of applications of the constructor case, and no other elements. In this example, $0 \in S$ because we said so explicitly, then because $0 \in S$, $0 + 2 = 2 \in S$, then because $2 \in S$, $4 \in S$, etc. Note that S does NOT contain any odd numbers (there's no way to construct an odd by adding 2 to a member of S)

Recursive set definitions are especially useful for many datatypes that appear in computer science, like strings and trees. Here's a recursive definition for the set of binary strings, which we denote $\{0,1\}^*$:

"Define $\{0,1\}^*$ recursively by

- $\lambda \in \{0,1\}^*$, and
- If $w \in \{0,1\}^*$, $0w \in \{0,1\}^*$ and $1w \in \{0,1\}^*$."

Here λ refers to the empty string,[3] and writing two strings (or a string and a character) next to each other like $0w$ refers to concatenating those strings, e.g. if $w_1 = 001$ and $w_2 = 11$, then $w_1 w_2 = 00111$ and $w_1 \lambda = w_1 = 001$. So based on this definition, λ is a string, then because λ is a string, $0\lambda = 0$ and $1\lambda = 1$ are strings, then because 0 and 1 are strings, 00, 10, 01, and 11 are strings, etc. Note that *infinite* sequences of ones and zeros are *not* strings by this definition - every finite sequence, no matter how long, can be constructed by a finite number of applications of the constructor case, but an infinite sequence cannot.

[3] In math we usually write strings without quotation marks - 001 is what we write where most programming languages would write "001". This generally leads to a nice-looking, uncluttered notation, but it does have the disadvantage that the empty string "" would be invisible without the quotation marks, which is why we have to use a special notation λ. So λ isn't a character like 0 and 1; it's just the notation for a string that has no characters at all.

Recursively-defined functions are especially well suited for recursively defined sets. Since every element of the set is there either because it's mentioned in the base step or because it's constructed by the constructor step, the function just has to specify how it behaves on each of those cases. For example, here's the string length function $|w|$ based on the definition of binary strings above:

$$|s| = \begin{cases} 0 & \text{when } s = \lambda \\ 1 + |w| & \text{when } s = 0w \\ 1 + |w| & \text{when } s = 1w \end{cases}$$

We often shorten such definitions by writing them in a compact form like this:

$$
\begin{aligned}
|\lambda| &= 0 \\
|0w| &= 1 + |w| \\
|1w| &= 1 + |w|
\end{aligned}
$$

Structural Induction

We can prove things about recursively defined objects using a variant of induction called *structural induction*. To prove $\forall x \in S(P(x))$ by structural induction (where S is a recursively defined set), you show that P is true for each element enumerated in the base case, and then show that it's true for each element enumerated in the constructor case as long as it's true for the smaller elements used to construct x. For example:

Theorem 1. *The length of the concatenation of two binary strings is the sum of the lengths of the individual strings, i.e.* $\forall t \forall s(|st| = |s| + |t|)$.

Proof. Fix string t, and we proceed by structural induction on s.

Base case: $s = \lambda$. In this case, $|st| = |\lambda t| = |t| = 0 + |t| = |\lambda| + |t| = |s| + |t|$.

Inductive case 1: $s = 0w$. Assume as our inductive hypothesis that $|wt| = |w| + |t|$. Then

$$
\begin{aligned}
|st| &= |(0w)t| \\
&= |0(wt)| && \text{(concatenation is associative)} \\
&= 1 + |wt| && \text{((recursive) definition of length)} \\
&= 1 + |w| + |t| && \text{(IH)} \\
&= |0w| + |t| && \text{((recursive) definition of length)} \\
&= |s| + |t|
\end{aligned}
$$

Inductive case 2: $s = 1w$. The proof is almost identical to the previous case so it is omitted.

Thus by structural induction, $|st| = |s| + |t|$ holds for every s, i.e. $\forall s(|st| = |s| + |t|)$ is true. Then since t was arbitrary, we have proven $\forall t \forall s(|st| = |s| + |t|)$. □

Structural induction is actually equivalent to normal induction, where the induction variable n would be the number of applications of the constructor needed to construct s. For example, here's the same proof rewritten as a proof by normal induction:

Proof. Fix string t. We will prove $\forall n \in \mathbb{N}(P(n))$, where $P(n)$ is the predicate "for any s which can be constructed using n applications

of the constructor step, $|st| = |s| + |t|$". We proceed by induction on n: s Base case: We need to show $P(0)$. The only string that can be constructed using 0 constructor steps is λ, so we need to show that for $s = \lambda$, $|st| = |s| + |t|$ holds. This is true because then $|st| = |\lambda t| = |t| = 0 + |t| = |\lambda| + |t| = |s| + |t|$.

Inductive case: Fix s and fix $k > 0$ and assume as our inductive hypothesis that P holds for all values smaller than k. Now we need to show that if s can be constructed using k applications of the constructor, then $|st| = |s| + |t|$. Since $k > 0$, we know that s is constructed using at least 1 application of the constructor step, so without loss of generality we know $s = 0w$. [4] Then it must be possible to construct w using only $k - 1$ applications of the constructor, so the inductive hypothesis applies to w, and we get:

[4] That is, it could also be $1w$, but the proof will be so similar that we omit it.

$$\begin{aligned}
|st| &= |(0w)t| \\
&= |0(wt)| && \text{(concatenation is associative)} \\
&= 1 + |wt| && \text{((recursive) definition of length)} \\
&= 1 + |w| + |t| && \text{(IH)} \\
&= |0w| + |t| && \text{((recursive) definition of length)} \\
&= |s| + |t|
\end{aligned}$$

Thus by induction, $|st| = |s| + |t|$ holds for every s regardless of the number of constructor applications required to construct s, i.e. $\forall s(|st| = |s| + |t|)$ is true. Then since t was arbitrary, we have proven $\forall t \forall s(|st| = |s| + |t|)$. □

Note that this proof is significantly more cumbersome than the structural induction version.

Trees

Recall that a (directed) binary tree is a tree where each vertex has at most two children. A binary tree can be built by creating a new root vertex and attaching it to the old roots of up to two old binary trees. In other words, the set of (directed) binary trees can be defined recursively as follows:

- A solitary root vertex is a binary tree.
- If T_1 is a binary tree, then so is T_1 plus a new root vertex which is a parent of T_1's former root.
- If T_1 and T_2 are binary trees, then so is T_1 and T_2 together along with one new root vertex which is a parent of the former roots of T_1 and T_2.

We will denote the three cases above as $\bullet$, $\bullet - T_1$, and $T_1 - \bullet - T_2$.[5] Let's define a couple functions on trees recursively. The number of vertices in a tree, $v(T)$, is just the number of nodes in any subtrees, plus one for the root:

$$\begin{aligned} v(\bullet) &= 1 \\ v(\bullet - T_1) &= 1 + v(T_1) \\ v(T_1 - \bullet - T_2) &= 1 + v(T_1) + v(T_2) \end{aligned}$$

The height of a tree $h(t)$ is the maximum distance from the root to a leaf, which increases by one each time a new root is added on top:

$$\begin{aligned} h(\bullet) &= 0 \\ h(\bullet - T_1) &= 1 + h(T_1) \\ h(T_1 - \bullet - T_2) &= 1 + max(h(T_1), h(T_2)) \end{aligned}$$

(Note that in the $T_1 - \bullet - T_2$ case we took the maximum height of the two subtrees, because only the *largest* distance from the root to a leaf matters.)

Now as one more example of structural induction, we prove a (rather trivial) lower bound on the number of vertices in a tree:

Theorem 2. *For any tree T, $v(T) \geq h(T) + 1$.*

Proof. We proceed by structural induction on T.

Base case: $T = \bullet$. Then $v(T) = v(\bullet) = 1 \geq 0 + 1 = h(\bullet) + 1 = h(T) + 1$.

Inductive case 1: $T = \bullet - T_1$. Assume as our inductive hypothesis that $v(T_1) \geq h(T_1) + 1$. Then $v(T) = v(\bullet - T_1) \overset{def}{=} 1 + v(T_1) \overset{IH}{\geq} 1 + (h(T_1) + 1) \overset{def}{=} h(\bullet - T_1) + 1 = h(T) + 1$. [6]

Inductive case 2: $T = T_1 - \bullet - T_2$. Assume as our inductive hypothesis that $v(T_1) \geq h(T_1) + 1$ and $v(T_2) \geq h(T_2) + 1$. Then

$$\begin{aligned} v(T) &= v(T_1 - \bullet - T_2) \\ &= v(T_1) + v(T_2) + 1 && \text{(definition of } v\text{)} \\ &\geq (h(T_1) + 1) + (h(T_2) + 1) + 1 && \text{(IH)} \\ &= h(T_1) + h(T_2) + 3 && \text{(cleanup)} \\ &\geq h(T_1) + h(T_2) + 2 && (3 \geq 2) \\ &\geq max(h(T_1), h(T_2)) + 2 && \text{(max of two nats is} \leq \text{their sum)} \\ &= h(T_1 - \bullet - T_2) + 1 && \text{(definition of } h\text{)} \\ &= h(T) + 1 \end{aligned}$$

[5] If you are using LaTeX, you can typeset this using "T_1-\bullet-T_2"

[6] This is a more compact way of writing a chain of (in)equalities similar to the one in the next case. The "def" and "IH" over the (in)equality signs are like writing "(definition)" and "(inductive hypothesis)" next to the corresponding steps.

Thus $v(T) \geq h(T) + 1$ in every case, induction complete.

□