Recursive definition

Margaret M. Fleck

9 October 2009

In this lecture, we see how to define functions and sets recursively, a mathematical technique that closely parallels recursive function calls in computer programs.

1 Announcements

Exams were returned in lecture today. If you missed lecture, we'll also bring them to discussion sections next week and afterwards you can get them at office hours.

2 Recursive definitions

You've all seen recursive procedures in programming languages. Recursive function definitions in mathematics are basically similar.

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 or cases
- Recursive formula/step

Recursive definitions are sometimes called inductive definitions or (especially for numerical functions) recurrence relations.

For example, recall that the factorial function $n!$ is defined by $n! = n \times (n-1) \times \ldots \times 2 \times 1$. We can define the factorial function $n!$ recursively:

- $0! = 1$
- $n! = n \cdot (n-1)!$

The recursive definition gets rid of the annoyingly informal ... and, therefore, tends to be easier to manipulate in formal proofs or in computer programs.

Notice that the base and inductive parts of these definitions aren't explicitly labelled. This is very common for recursive definitions. You're just expected to figure out those labels for yourself.

Here's another recursive definition of a familiar function:

- $g(1) = 1$
- $g(n) = g(n-1) + n$

This is just another way of defining the summation $\sum_{i=1}^{n} n$. This particular recursive function happens to have a nice closed form: $\frac{n(n+1)}{2}$. Some recursively-defined functions have a nice closed form and some don't, and it's hard to tell which by casual inspection of the recursive definition.

Notice that both the base case and the inductive equation must be present to have a complete definition. For example, if we leave off the base case in the definition of g, there are quite a lot of different functions that would satisfy the inductive condition.

3 More interesting sorts of recursion

The true power of recursive definition is revealed when the result for n depends on the results for more than one smaller value, as in the strong in-

duction examples we saw on Wednesday. For example, the famous Fibonacci numbers are defined by:

- $F_0 = 0$, $F_1 = 1$
- $\forall i \in \mathbb{N}, i \geq 2, F_i = F_{i-1} + F_{i-2}$

So $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$.

Recursion is also often the best approach when the value for n depends on the value for $n-1$ in a way that's more complicated than simple sums or products. For example, we could define a function f by:

- $f(0) = 3$
- $f(n) = 2f(n-1) + 3$

That is, $f(0) = 3$, $f(1) = 9$, $f(2) = 21$, $f(3) = 45$, etc. It's not instantly obvious how to rewrite this definition using a sum with ....

These more complicated recursive definitions frequently do not have a closed form. Nor is it easy to write them (even informally) using ....

4 Proofs with recursive definitions

Recursive definitions are ideally suited to inductive proofs. The main outline of the proof often mirrors the structure of the definition.

For example, let's prove the following claim about the Fibonacci numbers:

Claim 1 *For any $n \geq 0$, F_{3n} is even.*

Let's check some concrete values: $F_0 = 0$, $F_3 = 2$, $F_6 = 8$, $F_9 = 34$. All are even. Claim looks good. So, let's build an inductive proof:

Proof: by induction on n.

Base: $F_0 = 0$, which is even.

Induction: Suppose that F_{3k} is even. We need to show that that $F_{3(k+1)}$ is even.

$F_{3(k+1)} = F_{3k+3} = F_{3k+2} + F_{3k+1}$

But $F_{3k+2} = F_{3k+1} + F_{3k}$. So, substituting into the above equation, we get:

$F_{3(k+1)} = (F_{3k+1} + F_{3k}) + F_{3k+1} = 2F_{3k+1} + F_{3k}$

By the inductive hypothesis F_{3k} is even. $2F_{3k+1}$ is even because it's 2 times an integer. So their sum must be even. So $F_{3(k+1)}$ is even, which is what we needed to show.

Some people feel a bit uncertain if the base case is a special case like zero. It's ok to also include a second base case. For this proof, you would check the case for $n = 1$ i.e. verify that F_3 is even. The extra base case isn't necessary for a complete proof, but it doesn't cause any harm and may help the reader.

Another example, again with the Fibonacci numbers:

Claim 2 *For any $n \geq 1$, $\Sigma_{i=1}^{n} (F_i)^2 = (F_n)(F_{n+1})$.*

For example $\Sigma_{i=1}^{4} (F_i)^2 = 1 + 1 + 4 + 9 = 15 = 3 \cdot 5 = F_4 F_5$.

I have no intuitions about such equations. So let's hope induction will work in a simple way and give it a try:

Proof: by induction on n.

Base: If $n = 1$, then we have $\Sigma_{i=1}^{1} (F_i)^2 = (F_1)^2 = 1 = 1 \cdot 1 = F_1 F_2$. So this checks out.

Induction: Suppose that $\Sigma_{i=1}^{k} (F_i)^2 = (F_k)(F_{k+1})$.

$\Sigma_{i=1}^{k+1} (F_i)^2 = (\Sigma_{i=1}^{k} (F_i)^2) + (F_{k+1})^2$.

By the inductive hypothesis, this is $F_k F_{k+1} + (F_{k+1})^2$.

But $F_k F_{k+1} + (F_{k+1})^2 = F_k F_{k+1} + F_{k+1} F_{k+1} = F_{k+1}(F_k + F_{k+1}) = F_{k+1} F_{k+2}$. (The last step is using the definition of the Fibonacci numbers.)

So $\Sigma_{i=1}^{k+1} (F_i)^2 = F_{k+1} F_{k+2}$, which is what we needed to show.

5 Recursive definitions of sets

A set can also be defined recursively. For example, let's define a set of numbers S by

Base: $3 \in S$

Recursion: If $x \in S$ and $y \in S$, then $x + y \in S$.

One way to understand such a definition is that you put all the elements from the base case into your set (in this case, just 3). Then you apply the recursive step or steps repeatedly, adding more items to the set. S is the set containing all numbers you could make using any number of steps. So, in our case, since 3 is in S, you can apply the recursive step with $x = y = 3$. So then 6 must be in the set. But then $3 + 6 = 9$ is in the set. If you continue this process, any positive multiple of 3 will eventually end up in the set. So S is the set of all positive multiples of 3.

When interpreting such definitions, you pick the smallest set consistent with the definition. Sometimes this restriction is stated explicitly, but frequently it is left understood. For example, the set of all multiples of 3, positive and negative, satisfies both of the conditions given above. But this isn't the smallest set that satisfies them.

If we do want to define all the mutiples of 3, we could define a set S as follows:

(1) $3 \in S$

(2) If $x \in S$ and $y \in S$, then $x + y \in S$.

(3) If $x \in S$ and $y \in S$, then $x - y \in S$.

To generate the elements of S from the base set $\{3\}$, you apply combinations of the rules (2) and (3) in any order. For example, since 3 is in S, $3 + 3 = 6$ is in S by rule 2 But then $3 - 6 = -3$ is in S. And so is $3 - 3 = 0$. And so forth.

6 Another recursively-defined set

Now, let's try defining a set of 2D integer points recursively. Such sets aren't especially interesting from the point of view of applications, but they make nice teaching examples when you are getting used to recursive definitions. They'll prepare you to understand recursive definitions of sets of trees, strings, and similar objects in later CS classes.

So, let's define a set $S \subseteq \mathbb{Z}^2$ by

(1) $(3, 5) \in S$

(2) If $(x, y) \in S$ then $(x + 2, y) \in S$

(3) If $(x, y) \in S$ then $(-x, y) \in S$

(4) If $(x, y) \in S$ then $(y, x) \in S$

To work out what S contains, start with the element that's explicitly given: $(3, 5)$. Then start working out what other values the three recursive rules force you to add.

If you feed $(3, 5)$ into rule (2), you get $(5, 5) \in S$. Feeding the output of rule (2) back into rule (2) gives you $(7, 5)$, and then $(9, 5)$, and eventually all numbers of the form $(n, 5)$ where n is odd and ≥ 3.

Now if we apply rule (3) to this (large) set of pairs in S, we find that pairs such as $(-3, 5)$, $(-5, 5)$, $(-7, 5)$, etc must also belong to S. We've now found that S contains most of the pairs that contain an odd integer followed by 5. We can fill in the gap by applying rule (2) again to $(-3, 5)$, to produce $(-1, 5)$ and $(1, 5)$.

So S contains all pairs $(n, 5)$ where n is odd. If we apply rule (4) to these pairs, S contains all pairs $(5, n)$, where n is odd. But, then, if we use rules (2) and (3) again, we can generate all pairs of the form (m, n) where both m and n are odd. Subsequent applications of the three recursive rules doesn't get us any new elements, so this is the full set S.

We could use induction to prove that this closed-form description matches the recursive definition of S, but we won't do that right now. For the moment,

concentrate on making sure you can figure out the correct closed form and explain informally why it's correct.