Recursive Definition

Recursive function definitions in mathematics are basically similar to recursive procedures in programming languages

A recursive definition always has two parts:

- Base case or cases
- 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 $n \geq 2$

Both the base case and the recursive formula must be present to have a complete definition

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 induction examples.

For example, the famous Fibonacci numbers are defined:

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

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

It isn't at all obvious how to express this pattern non-recursively.

Finding closed forms (1)

The simplest technique for finding closed forms is called "unrolling".

For example, suppose we have a function T : N → Z defined by

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1)+3, \quad \forall n \geq 2 \end{aligned}$$

The values of this function are T(1) = 1, T(2) = 5, T(3) = 13, T(4) = 29, T(5) = 61.

It isn't so obvious what the pattern is.

- The idea behind unrolling is to substitute a recursive definition into itself,
- so as to re-express T(n) in terms of T(n–2) rather than T(n–1).
- We keep doing this, expressing T(n) in terms of the value of T for smaller and smaller inputs,
- until we can see the pattern required to express T(n) in terms of n and T(0).
- So, for our example function, we would compute:

$$
\begin{aligned}
T(n) &= 2T(n-1)+3 \\
&= 2(2T(n-2)+3)+3 \\
&= 2(2(2T(n-3)+3)+3)+3 \\
&= 2^3T(n-3)+2^2\cdot 3+2\cdot 3+3 \\
&= 2^4T(n-4)+2^3\cdot 3+2^2\cdot 3+2\cdot 3+3 \\
\ldots \\
&= 2^kT(n-k)+2^{k-1}\cdot 3+\ldots+2^2\cdot 3+2\cdot 3+3
\end{aligned}
$$

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

Now, we need to determine when the input to T will hit the base case.

In our example, the input value is $n - k$ and the base case is for an input of 1.

So we hit the base case when $n - k = 1$. i.e. when $k = n - 1$

Substituting this value for k back into our equation, and using the fact that T(1) = 1, we get

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

Finding closed forms (2)

The second technique for finding closed forms is using "induction" starting with a guess or claim for the solution.

Claims involving recursive definitions often require proofs using a strong inductive hypothesis.

For example, suppose that the function f : N → Z is defined by

$$f(0) = 2$$

$$f(1) = 3$$

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

Claim $\forall n \in \mathbb{N}, f(n) = 2^n + 1$

Claim $\forall n \in \mathbb{N}, f(n) = 2^n + 1$

Proof: by induction on n.

Base: $f(0)$ is defined to be 2. $2^0+1 = 1+1 = 2$. So $f(n) = 2^n+1$ when $n = 0$.

$f(1)$ is defined to be 3. $2^1 + 1 = 2 + 1 = 3$. So $f(n) = 2^n + 1$ when $n = 1$.

Induction: Suppose that $f(n) = 2^n + 1$ for $n = 0, 1, \ldots, k$.

$f(k+1) = 3f(k) - 2f(k-1)$

By the induction hypothesis, $f(k) = 2^k+1$ and $f(k-1) = 2^{k-1}+1$. Substituting these formulas into the previous equation, we get:

$f(k+1) = 3(2^k+1) - 2(2^{k-1}+1) = 3 \cdot 2^k + 3 - 2^k - 2 = 2 \cdot 2^k + 1 = 2^{k+1} + 1$

So $f(k+1) = 2^{k+1} + 1$, which is what we needed to show.

We need to use a strong induction hypothesis, as well as two base cases, because the inductive step uses the fact that the formula holds for two previous values of n (k and $k - 1$).