- Sometimes, a function can refer to itself at a different value, to show its relation to smaller cases.
  **Example:** The function $f(n) = 2^n$ can be referred to as doubling the previous value, $f(n - 1)$.

  $$f(n) = 2^n = 2 \cdot 2^{n-1} = 2 \cdot f(n - 1).$$

- Taking the self-referential formula as a definition, we call this *recursion*. 
**Computing Recursion**

- To evaluate recursion, apply the formula again on the smaller index:

  \[ f(n) = 2f(n-1) = 2(2f(n-2)) = 2(2(2f(n-3))) = \cdots \]

- This has to stop somewhere, so we create a set of base cases.

  \[ f(0) = 1. \]

- This allows us to compute until the base case arrives:

  \[ f(4) = 2f(3) = 4f(2) = 8f(1) = 16f(0) = 16. \]
We can use case notation to define a recursive function:

\[
f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
2 \cdot f(n - 1) & \text{otherwise.}
\end{cases}
\]

This outlines how we can translate from a function into a program:
1. Identify inputs
2. Test for base cases – Return if found
3. Otherwise – Return evaluation of recursive formula
int f(int n)
{
    if ( n == 0 )
    {
        return 1;
    }

    return 2*f(n-1);
}
When you recurse, the new parameters are used, creating a “call stack” (see markerboard demo).

Recursion formulas can have multiple instances of smaller cases:

\[
\text{choose}(n, k) = \text{choose}(n - 1, k - 1) + \text{choose}(n - 1, k).
\]

Don’t forget your base cases!