As we have seen, many functions can naturally be defined in terms of other functions.

```haskell
factorial :: Int -> Int
factorial n = product [1..n]
```

factorial maps any integer $n$ to the product of the integers between 1 and $n$. 
Expressions are **evaluated** by a stepwise process of applying functions to their arguments.

For example:

\[
\text{factorial 4} \\
= \\
\text{product [1..4]} \\
= \\
\text{product [1,2,3,4]} \\
= \\
1*2*3*4 \\
= \\
24
\]
Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

factorial 0 = 1
factorial (n+1) = (n+1) * factorial n

factorial maps 0 to 1, and any other positive integer to the product of itself and the factorial of its predecessor.
For example:

\[
\text{factorial 3} = 3 \times \text{factorial 2} = 3 \times (2 \times \text{factorial 1}) = 3 \times (2 \times (1 \times \text{factorial 0})) = 3 \times (2 \times (1 \times 1)) = 3 \times (2 \times 1) = 3 \times 2 = 6
\]
Note:

- factorial 0 = 1 is appropriate because 1 is the identity for multiplication: 1\times x = x = x \times 1.

- The recursive definition diverges on integers \(< 0\) because the base case is never reached:

```
> factorial (-1)
Error: Control stack overflow
```
Why is Recursion Useful?

- Some functions, such as factorial, are **simpler** to define in terms of other functions.

- As we shall see, however, many functions can **naturally** be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of **induction**.
Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

\[
\text{product} :: [\text{Int}] \rightarrow \text{Int} \\
\text{product} \ [{}] = 1 \\
\text{product} \ (n:ns) = n \times \text{product} \ ns
\]

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.
For example:

\[
\begin{align*}
\text{product } [2,3,4] &= 2 \times \text{product } [3,4] \\
&= 2 \times (3 \times \text{product } [4]) \\
&= 2 \times (3 \times (4 \times \text{product } [])) \\
&= 2 \times (3 \times (4 \times 1)) \\
&= 24
\end{align*}
\]
Using the same pattern of recursion as in product we can define the `length` function on lists.

\[\text{length} :: [a] \rightarrow \text{Int}\]

\[\text{length} [] = 0\]

\[\text{length} (_:xs) = 1 + \text{length} \, xs\]

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.
For example:

\[
\begin{align*}
\text{length } [1, 2, 3] &= 1 + \text{length } [2, 3] \\
&= 1 + (1 + \text{length } [3]) \\
&= 1 + (1 + (1 + \text{length } [])) \\
&= 1 + (1 + (1 + 0)) \\
&= 3
\end{align*}
\]
Using a similar pattern of recursion we can define the **reverse** function on lists.

**reverse** :: [a] → [a]

reverse [] = []

reverse (x:xs) = reverse xs ++ [x]

**reverse** maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.
For example:

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- **Zipping the elements of two lists:**

  \[
  \text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)]
  \]

  \[
  \text{zip} \ [\] \ _ \ = \ [\]
  \]

  \[
  \text{zip} \ _ \ [\] \ = \ [\]
  \]

  \[
  \text{zip} \ (x:xs) \ (y:ys) \ = \ (x,y) : \text{zip} \ xs \ ys
  \]
Remove the first $n$ elements from a list:

\[
\text{drop} :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{drop 0 } xs = xs \\
\text{drop (n+1) } [] = [] \\
\text{drop (n+1) } (_:xs) = \text{drop n } xs
\]

Appending two lists:

\[
(++) :: [a] \rightarrow [a] \rightarrow [a] \\
[] ++ ys = ys \\
(x:xs) ++ ys = x : (xs ++ ys)
\]
The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;

- Non-empty lists can be sorted by sorting the tail values \( \leq \) the head, sorting the tail values \( > \) the head, and then appending the resulting lists on either side of the head value.
Using recursion, this specification can be translated directly into an implementation:

```haskell
qsort      :: [Int] → [Int]
qsort []    = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
where
    smaller = [a | a \leftarrow xs, a ≤ x]
    larger  = [b | b \leftarrow xs, b > x]
```

Note:

- This is probably the simplest implementation of quicksort in any programming language!
For example (abbreviating qsort as q):

\[ q \ [3, 2, 4, 1, 5] \]

\[ q \ [2, 1] \quad ++ \quad [3] \quad ++ \quad q \ [4, 5] \]

\[ q \ [1] \quad ++ \quad [2] \quad ++ \quad q \ [] \]

\[ q \ [] \quad ++ \quad [4] \quad ++ \quad q \ [5] \]

\[ [1] \]

\[ [] \]

\[ [] \]

\[ [] \]

\[ [5] \]
Exercises

(1) Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:

  \[ \text{and} :: [\text{Bool}] \rightarrow \text{Bool} \]

- Concatenate a list of lists:

  \[ \text{concat} :: [[\text{a}]] \rightarrow [\text{a}] \]
Produce a list with n identical elements:

\[
\text{replicate :: Int} \to a \to [a]
\]

Select the nth element of a list:

\[
(!!) :: [a] \to \text{Int} \to a
\]

Decide if a value is an element of a list:

\[
\text{elem :: Eq a \Rightarrow a} \to [a] \to \text{Bool}
\]
(2) Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

that merges two sorted lists of integers to give a single sorted list. For example:

```haskell
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```
Define a recursive function

```haskell
msort :: [Int] → [Int]
```

that implements **merge sort**, which can be specified by the following two rules:

- Lists of length \( \leq 1 \) are already sorted;
- Other lists can be sorted by sorting the two halves and merging the resulting lists.