Recursion

Recursion

In programming, **recursion** occurs whenever a function calls itself. For example, this is a recursive function:

```
1  def recursive_function(num)
2      puts "This is recursive call #{num}"
3      recursive_function(num + 1)
4  end
```

The function is recursive because it calls itself on line 3.

recursive_function

```ruby
def recursive_function(num)
    puts "This is recursive call #{num}"
    recursive_function(num + 1)
end
```

If we call recursive_function(1), it will produce this output:

This is recursive call 1
This is recursive call 2
This is recursive call 3
...

Unfortunately, we never tell the function when to stop calling itself, so it will keep going until it causes a stack error:

```
This is recursive call 11886
This is recursive call 11887
SystemStackError: stack level too deep
```

(The stack is what the computer uses to track function calls.)

Components of Recursive Functions

We keep recursive functions from falling into infinite loops by providing them with a base case.

A **base case** tells the function when to stop calling itself.
It typically represents the simplest case of a problem.

If the conditions for the base case are not met, then we execute the **recursive case**, which calls the function again.
The recursive case should bring us closer to the base case.
It typically represents a subset of the original problem.

When to use Recursion

Problems that lend themselves well to recursive solutions are ones whose solution can be found in part by solving the same problem for a smaller subset of the data.

Examples include:

- Mathematical sequences that calculate each new number based on applying a consistent rule to the previous number(s)
- Sorting algorithms that sort and then recombine subsets of the data
- Search algorithms on sorted data where each pass eliminates portions of the remaining data that must be searched

Example: Factorial

Suppose we want to write a function to calculate the factorial of a number. We calculate a number's factorial (expressed with an ! after the number: 5!) by multiplying all the numbers up to and including that number. So

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

We can also express 5! as

$$5! = 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!$$

We can abstract this into a general rule:

$$n! = n \times (n\text{-}1 \times n\text{-}2 \times \ldots \times 1) = n \times (n\text{-}1)!$$

This ability to express the desired result in terms of a smaller subset of the same problem (n-1!) suggests recursion might be useful.

Example: Factorial

To set up a recursive factorial function, we need to establish the base case and the recursive case. Starting with the recursive case, we noted that n! = n x (n-1)!. Our recursive case is therefore:

```ruby
def factorial(num)
    #recursive case
    num * factorial(num-1)
end
```

We also need a base case to tell the function when to stop. For factorial, the simplest case would be 1!. For 1!, we do not need to calculate the value. We can simply return 1.

```ruby
def factorial(num)
    #base case
    return 1 if (num == 1)

    #recursive case
    num * factorial(num-1)
end
```

N.B.: Always check for the base case **before** executing the recursive case!

Example: Factorial

When we call factorial(5), the code will execute as follows:

```
1    def factorial(num)
2        #base case
3        return 1 if (num == 1)
4
5        #recursive case
6        num * factorial(num-1)
7    end
```

Put factorial(2) on the stack

Check the base case, 2 != 1

Execute the recursive case (line 6)

Stack

factorial(2)	
factorial(3)	**Paused at line 6**
factorial(4)	**Paused at line 6**
factorial(5)	**Paused at line 6**