

1 Why Recursion?

Recursion is a useful computation technique borrowed from mathematics. A recursive function attempts to solve a problem by doing a bit of computation and then breaking the input into smaller problems that can be solved using the same strategy.

Recursive code is generally shorter and easier to write than iterative code. In fact, loops are often turned into recursive functions when they are compiled or interpreted.

2 Iterative code

For instance, an iterative version of the factorial function:

```
// calculates factorial of a positive integer
int fact(int n) {
    int answer = 1; // factorial of 0 or 1 is 1

    // loop thru each int n...2 and mult them together
    while (n > 1) {
        answer = answer * n;
        n = n - 1;
    }

    return answer;
}
```

A math definition of iterative factorial looks like

$$n! = \prod_{i=0}^n i$$

If you didn't already have the code for iterative factorial above, it would be easy to make mistakes with the loop. You have to understand how to write a function that handles the \prod in our definition.

3 Recursive code

A recursive solution realizes that for each integer [0..n] we are multiplying number into a previous answer.

A math definition of recursive factorial looks like

$$\begin{aligned} n! &= n * (n-1)! \\ 1! &= 1 \\ 0! &= 1 \end{aligned}$$

This version has no dots, \prod symbols, or other unknown bits and is a simple recipe for a base case and a general (recursive) case. In particular, you should notice that the recursive case calls itself on a smaller problem.

¹Liberally borrowed from Professor Wittie

4 Format of a recursive function

You can write almost all recursive functions using this format:

```
// base case
if (test for the base case)
    return some base case value

// another base case
else if (test for another base case)
    return some other base case value

// the recursive case
else
    return (some work) and then (a recursive call)
```

In the case of the factorial function, we get:

```
// calculates factorial of a positive integer
int fact(int n) {

    // base cases: fact of 0 or 1 is 1
    if (n == 1)
        return 1;
    else if (n == 0)
        return 1;

    // recursive case: multiply n by (n-1) factorial
    else
        return n*fact(n-1);
}
```

5 Tail Recursion

For many programming languages, calling a method like `fact` defined in the previous section with a large number will cause the program stack to overflow. There are just too many calls to `fact` stacked on top of each other and the program does not know it will be so big. In these cases, you want to use a variant of recursion called *tail recursion*.

With tail recursion, you pass parts of the solution along with the recursive call so that you do not have to return to the function that made the recursive call. This way, the language can do some tricks to reduce the stack size without compromising the computation.

Typically, tail recursion is implemented by having the result be passed as an argument to function. In our `fact` example, we could implement tail recursion as follows:

```
// calculates factorial of a positive integer
int fact(int n) {

    // call an auxilliary function
    return fact_aux(n, 1);

}
```

```
// an auxilliary function for factorial using tail recursion
private int fact_aux(int n, int sol) {

    //deal with the base cases
    if (n == 1 || n == 0)
        return sol;

    //handle the general case
    return fact_aux(n-1,sol*n);
}
```