# Recursion

Raniconduh

## Introduction

Recursion is a mathematical process that allows a function to be defined in terms of itself. Recursive algorithms are often much easier to come up with than iterative ones, and enough analysis will allow a recursive algorithm to be very efficient. Recursion is so powerful that many programming languages don't define a loop construct, instead requiring recursion. In the most general case, recursion is used to split up a large problem into smaller sub-problems that can be easily calculated.

## Mathematical Description

A recursive function takes the form of one or more base cases as well as a function definition in terms of itself. Recursive functions will typically only operate on discrete sets such as the naturals or the integers. The base case is the lowest term in the recursion and any recursive function call will eventually make its way down to the base case. A recursive definition may be in the following form:

```
f(1) = 1
f(n) = n * f(n - 1)
```

A recursive function may use anything as a base case and may use any other functions in the recursive definition, but this is a simple case. Here, the base case is shown as `f(1) = 1`. This means that `f(n = 1)` is necessarily `1`. The recursive definition then follows as `f(n) = n * f(n - 1)`. This means that for any `n`, the function calls itself with `n - 1` as the argument and then multiplies the result by `n`. To determine what this function does, we can unroll it. That is, we can look at what happens when the function is called with a given value.

### Unrolling

Let's consider the case where `n = 3`. Then, we want to find `f(3)`. Since `n =/= 1`, we must look at the recursive definition.

```
f(3) = 3 * f(3 - 1)
     = 3 * f(2)
```

```
f(2) = 2 * f(2 - 1)
     = 2 * f(1)
f(1) = 1
```

Unrolling the function allows each term to be defined using the next term down. By finding the value of every term down, we can simply back-substitute into the previous equations to determine the final output. Note that `f(1) = 1` was set without looking at the recursive definition, since it is the base case. Then, by substitution:

```
f(3) = 3 * f(2)
     = 3 * 2 * f(1)
     = 3 * 2 * 1
     = 6
```

In this case we find that `f(3) = 6`. However, looking at the definition allows us to see that the function `f(n)` is actually calculating `n!`, so it is the factorial function. Applying this same analysis to `f(n)` for larger values of `n` will allow us to confirm this and we find its definition: `n! = (n)(n - 1)(...)(1)`. This function can also be translated into code to find the factorial of a number during program execution:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

The base case is now specified by the `if` statement, and the recursive definition is found in the `return` statement.

### Time Complexity

Often it is necessary to determine how many recursions are necessary for the function to compute its result. On a computer, for instance, large recursive functions can lead to stack overflows, which are undesirable. In the case of the factorial function, it is clear that `f(n)` takes `n - 1` recursive cycles to compute the end result. For more complex recursive algorithms, calculating this time complexity becomes increasingly more difficult, and it is covered in another article.

## Backtracking Recursive Algorithms

Recursive algorithms are often used to solve problems computationally, rather than just to compute an output based on an input (like in the case of the factorial function). One such example is a search on a binary tree. In this case, it is useful to implement backtracking. This means that the algorithm will descend down the tree, and if it can't find the item being searched for, it will regress to the previous level of recursion and try again. The following is one such example of this:

```
def search(t: tree, s):
    if t.value == s:
        return true

    if search(t.left, s) is true:
        return true
    else if search(t.right, s) is true:
        return true
    return false
```

The above algorithm will search a (unordered) binary tree for an element `s` and will return `true` if the element is in the tree. Otherwise, it will return false. By examining the code, we see that the algorithm will first descend as far down on the left of the tree as possible. If the farthest left node is not the search element, it will backtrack to the node above it, and then check the right node. The algorithm will continue this process until the element has been found. Now, let's trace the algorithm's execution for a simple tree input:

```
   7
  / \
 3   12
```

The above binary tree is composed of random values. Now, let's search for `12` in this tree. First the algorithm will consider the root node `7`. Since it is not the correct value, we will look at the left node, `3`. Since `3` is also not the correct value, we will try to look at the next left node. Since `3` is a leaf, it has no left node, so we will return to `3` to look at its right node. Once again, `3` has no right node, so we return to the root. Now, we consider the right node, `12`, of the tree. Since `12` is the value we are searching for, we can return true. The path that the algorithm takes through the tree can be drawn out as follows:

```
   7

   7
  /
 3

   7
  / \
 3   12

7 -> 3 -> 7 -> 12
```

Now, let's consider a more complicated tree:

```
    1
```

```
   / \
  2   5
 / \
3   4
```

When searching for `5`, the algorithm will first check the root `1`, then its left child `2`, then its left child `3`. Since `3` is a leaf, the algorithm will go back to node `2` and check its right children. Since `4` is also a leaf, and there are no more children of `2` to check, the algorithm goes back to the root `1` and check its right children. In this case, `5` is the right child of the root, so the algorithm will finally return `true`. The following traces out the path the algorithm takes through the tree, with the caret marking the node the algorithm is currently checking.

```
 1 ->  1  ->   1  ->  1  -->   1
 ^    /       /      /        / \
     2       2      2        2   5
     ^      /      / \      / \  ^
           3      3   4    3   4
           ^          ^
```

## Proofs

Recursion is often used in mathematical proofs, where it takes the form of induction. In this case, a proof is wanted to be able to determine if a fact holds for every case. First a base case is specified where the fact is obviously true. Then, recursion is applied to larger cases to show that every case will eventually decay into the base case, and therefore the given fact must be true.

## Conclusion

Recursive functions allow for a relatively simple solution to problems that can appear quite large – instead of thinking of a direct solution to a complicated problem, it is often easier to break the problem up into smaller, more manageable pieces. It is important to note, however, that recursive algorithms on computers often require extra computational power as well as memory when compared to their direct solution counterparts. In this sense, choosing between a recursive solution and a direct solution requires weighing out the programmer time it takes to develop a direct solution versus the computational power and memory being wasted by a recursive solution.

Recursion is a necessary tool for any programmer and being able to understand its inner workings is essential to being a good programmer and computer scientist. That is not to say, however, that it doesn't have its downsides. To remedy these, a concept called dynamic programming is employed. It is worthwhile to study this idea as it leads to many efficient and simple solutions to seemingly complicated problems.