## **What is Recursion?**

Sometimes, a problem is difficult or too complicated to solve because it is too large. If the problem can be separated into smaller variations of itself, then it will be easier to figure out how to solve one of these smaller versions and then create a solution.

This is the concept behind recursion, and with regards to a programming function, recursion occurs whenever a function calls itself within its context. It is called continuously until the fundamental condition that splits the loop is fulfilled.

## **Why recursion method is a bad idea to find the Fibonacci of a number**

Recursion used in complex sequence buries the machine in a very long chain of “calls” with hardly any processing between them, which is perhaps the most expensive thing a computer can do.

In this case, if recursion is used to determine the Fibonacci sequence, every factor can be measured right from the start. It does not save the already estimated values, and in a long sequence, it can flood the push-down list and trigger standard memory to burn out, and then cause mass storage swapping. If the recursion goes too deep, it may cause a memory leak in production.

Reasons why recursion is not a good idea in solving Fibonacci, are stated below:

1. In languages that do not have excellent support for recursion, it is costly and brittle to solve all but the smallest and most straightforward problems with recursion.
2. Also, in languages that support recursion well, the simple, naive (and most widely cited) recursive approach is grossly inefficient.
3. Often, when the input gets higher, and it does not even have to be that high, maybe just 50 or 100 is high enough to the point where all those additional “overlapping” simulations will stack up, and the program will not end.
4. If the appropriate recursive model is chosen, then it is not a flawed idea to use recursion.

The widely cited recursive approach for identifying the Fibonacci sequence looks like this (pseudocode):

# This function - and all the versions below - starts from the

# 0th value in the sequence, not the 1st

function fib(n)

case n is 0

return 1

case n is 1

return 1

case n > 1

return fib (n - 1) + fib (n - 2)

Although this method is a clear example of the reasoning of a recursive model, it has two key issues as a functional solution:

1. It performs the same formulas a variety of times. Considering the last row of the program: when attempting to discover fib(9), fib(7) will be determined twice (once to enable estimation of fib(8) on the left and the other to evaluate fib(7) on the right). This repetition of action increases exponentially as the sequence continues to back down.
2. Every call to fib(n) introduces another stack frame (i.e., sets the arguments and the position of the code to return to after the function is invoked to the stack). While the stack is finite and the above approach generates an in-depth collection of nested function calls for all but the smallest numbers, this is a method that has a high chance of overflowing the stack and forcing the software to fail.

This does not mean that recursion is a wrong choice to this situation, only that the above approach is not feasible.

There are additional ways to solve this problem recursively that do not have these challenges.

1. When using a language that supports tail call optimisation, it is ideal to use a tail-recursive solution.
2. With a language that supports lazy evaluation, corecursion can be used.

Tail recursion

In the TCO supported language, in which the same stack frame is reused for consecutive recursive calls, this would be effective.

function fib(n)

function next (count, first, second)

if count = 0

return first + second

else

return next(count -1, second, first + second)

if n < 2

return 1

else

return next(n - 2, 1, 1)

Lazy evaluation and corecursion

In a language that has some support for lazy evaluation, it is possible to create recursive functions that produce sequences of values in such a way that, as long as the output is processed one item at a time, the code runs in steady space without creating additional stack frames.