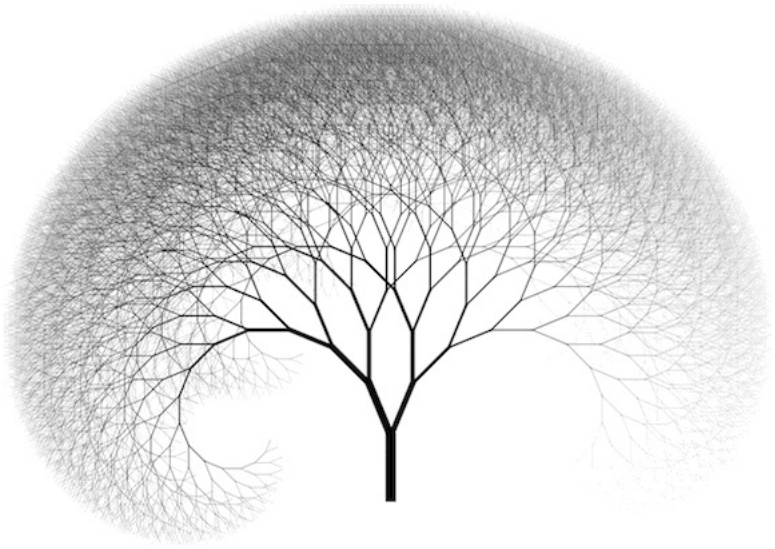
Exploring Recursion in R

Harley Velez



In this post we'll explore the place of recursion in the R programming ecosystem. We'll start from the basics so don't worry if you don't know what recursion is yet.

The Building Blocks

> "To understand recursion, you must first understand recursion" - Unknown

Lets first learn the basics of what recursion is. Geeks for Geeks lists recursion as the process in which a function calls itself directly or indirectly. A simple example is a function that decrements an integer to printing each value to 1. An iterative version would look like,

```
iterativeDecrementer <- function(x) {
  while (x > 0) {
    print(x)
    x = x - 1
  }

return()
}

iterativeDecrementer(5)
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
```

```
## NULL
```

If we want to do this recursively we can write it as,

```
recursiveDecrementer <- function(x) {
  if (x == 1) {
    print(x)
    return()
  }

  print(x)
  recursiveDecrementer(x - 1)
}

recursiveDecrementer(5)
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
```

```
## NULL
```

You can see that the function will continue to call itself over and over until the condition of `x == 1` is met and then it will finally break out. We can call this statement the base case. When our base case it met we finish our call.

If we want to create a function that will give us the factorial of a number how could we do this? First we need to decide on a base case. For a factorial function we want multiply a number `n` by `n - 1`, `n - 2`, to `n - (n - 1)`. So for 5 we would do `5 * 4 * 3 * 2 * 1`. So our base case should be when our number gets down to 1. The function will end up looking somewhat similar to the previous,

```
recursiveFactorial <- function(n) {
  if (n == 1) {
    return(n)
  }

  return(n * recursiveFactorial(n - 1))
}


recursiveFactorial(5)
```

```
## [1] 120
```

Notice that in this case we are returning a value to the previous function call. So when the first call is made it says `return(5 * recursiveFactorial(4))`, then the next happens as `return(4 * recursiveFactorial(3))` and so forth down to 1. When it hits 1 it sends the one to the call before it, do the multiplication, then send it to the call before it until we get back to the original call where `n=5`. The function finally does that call and then return the result. Also keep in mind the usage of the `return()` function here. In typical R functions the last line written in a function will be the returned value (citation) but since a recursive function doesn't follow this typical flow it is necessary to explicitly say what you are returning.

One way I like to think of recursion is an assembly line. Where the first person, will need to know what 4! is in order to multiply 5 by it. So he turns to the person next to him and asks them what factorial of 4 is. That person needs to know what 3! is in order to multiply it by 4 to get 4!. So he asks the person next to him for 3!. They need 2! so they ask the person next to them. That person then needs 1! from the person next to them. 1! is the last piece needed so the 1! person passes the information back to the 2! person and so on back up until the original person finally knows what 4! is so he can multiply it by 5 to get 5!.

Note: If this is your first time seeing recursion and the above example didn't immediately click, don't worry. Recursion is a difficult topic to initially wrap your head around. If you keep trying to learn it and playing with it in your own code it will at some point click and become easy.

Runtime Analysis

> "Time is the longest distance between two places." — Tennessee Williams, The Glass Menagerie

Now that we've established a method for writing recursive functions in R we can begin to weigh the pros and cons of using recursion in our code. As I said before, iteration and recursion are interchangeable; we write a function with either. If you're interested in why this is, check out this discussion with several proofs for iterative == recursive. Often times however, we will find a function or algorithm suits one method much nicer than the other. In these cases we may run into issues of cleanliness of code versus runtime of said code.

A tool we can use to roughly estimate the runtime for code in R is the `Sys.time` function. Credit to r-bloggers.com for this creative method. As you'll soon see, we'll take our computers clock time before and after the code finishes and then check the difference.

Let's start by seeing how long the factorial function takes iteratively and recursively. First we'll use the factorial function we wrote above using a larger input,

```
start <- Sys.time()
# Depending on your machines capabilities you can adjust the input up or down
recursiveFactorial(500)
```

```
## [1] Inf
```

```
end <- Sys.time()
end - start
```

```
## Time difference of 0.002571106 secs
```

NOTE: You may notice the factorial functions hit the infinity cap in R. This shouldn't actually affect our runtime analysis since both functions do the same amount of multiplication and more importantly the runtime isn't dominated by the time to multiply it's dominated by the time for R to open a new function call.

And now an iterative approach,

```
iterativeFactorial <- function(n) {
  for (i in 1:(n-1)) {
    n = n * i
  }

  return(n)
}
```

```
start <- Sys.time()
iterativeFactorial(500)
```

```
## [1] Inf
```

```
end <- Sys.time()
end - start
```

```
## Time difference of 0.006170988 secs
```

Every time you run these speed tests you will get a different answer. This is because your computer may compute the function faster or slower depending on what your processor is currently doing. We should still expect runtimes that are pretty consistent with a few outliers if we keep running the tests.

With this in mind lets take the average of running both of them several times. Starting with the recursive,

```
sum = 0
amountOfLoops = 250

for (i in 1:amountOfLoops) {
  start <- Sys.time()
  # Depending on your machines capabilities you can adjust the input up or down
  recursiveFactorial(500)
  end <- Sys.time()

  sum = sum + (end - start)
}

averageForRecursive = sum / amountOfLoops

print(averageForRecursive)
```

```
## Time difference of 0.0002950363 secs
```

Now for the iterative,

```
sum = 0
amountOfLoops = 250

for (i in 1:amountOfLoops) {
  start <- Sys.time()
  # Depending on your machines capabilities you can adjust the input up or down
  iterativeFactorial(500)
  end <- Sys.time()

  sum = sum + (end - start)
}

averageForIterative = sum / amountOfLoops

print(averageForIterative)
```

```
## Time difference of 2.412987e-05 secs
```

We can now see that iteration has out performed recursion in this case. But will it always outperform recursion? Unfortunately in most cases the answer is yes. While it is dependent on the language, all modern languages are written in a way such that iteration will be faster. One reason is that every time the computer makes a call to a function is has to allocate memory in order to store the variables for this new function (Learn more here). The astute reader may be thinking to themselves that this would not only be slower but also take up more memory, and they'd be right. After the function is done the computer then has to deallocate this memory and go back to where it was before the function was called. So you can imagine with our factorial function the iterative version has to allocate and deallocate memory once while the factorial version has to do it 500 times. For a more advanced discussion on this topic check out this discussion.

So why recursion?

> "You'll miss the best things if you keep your eyes shut." — Dr. Seuss

Learning that recursion in R is slower than iteration and that iteration can do anything recursion can seems to leave us asking what the point of using recursion is. As I mentioned before you may find yourself in situation while writing code where you have to choose between runtime and readability (here is another great stackoverflow discussion on the topic). There is no golden rule and the decision is made on a case by case basis. Sometimes you'll find the runtime tradeoff to be negligible compared to the elegance of a recursive solution. You should always keep in mind that your future self and often other people will need to read and understand your code. On top of this, when you write your solutions iteratively you are often saving and updating variables in a way that you wouldn't need to if the solution was recursive since the computer would be handling it behind the scenes. This manual manipulation gives more chance for error. In some more barebones languages like C an "off by one error" is a situation where a loop terminates one loop farther than intended. This vulnerability can sometimes be used to break into the system running the program by malicious hackers. You can learn more about this error type of error here. This particular vulnerability isn't an issue in R but it's important to realize that iteration is more of a manually defined method than a recursive method is.

In the end it seems that recursion does have its place even in the R ecosystem. It may not always be needed but there are times where it may actually lead to a simpler solution. So next time you're designing an algorithm or are stuck writing a clean function in R don't forget about recursion as it may just be the elegance you were looking for.

References used throughout post:

https://stackoverflow.com/questions/2651112/is-recursion-ever-faster-than-looping

http://www.geeksforgeeks.org/recursion/

https://stackoverflow.com/questions/2093618/can-all-iterative-algorithms-be-expressed-recursively

https://www.htbridge.com/vulnerability/off-by-one-error.html#impact

https://www.youtube.com/watch?v=75gBFiFtAb8&t

https://www.programiz.com/r-programming/return-function

https://stackoverflow.com/questions/183201/should-a-developer-aim-for-readability-or-performance-first

Citations:

http://matthewjamestaylor.com/blog/create-fractals-with-recursive-drawing

https://xkcd.com/