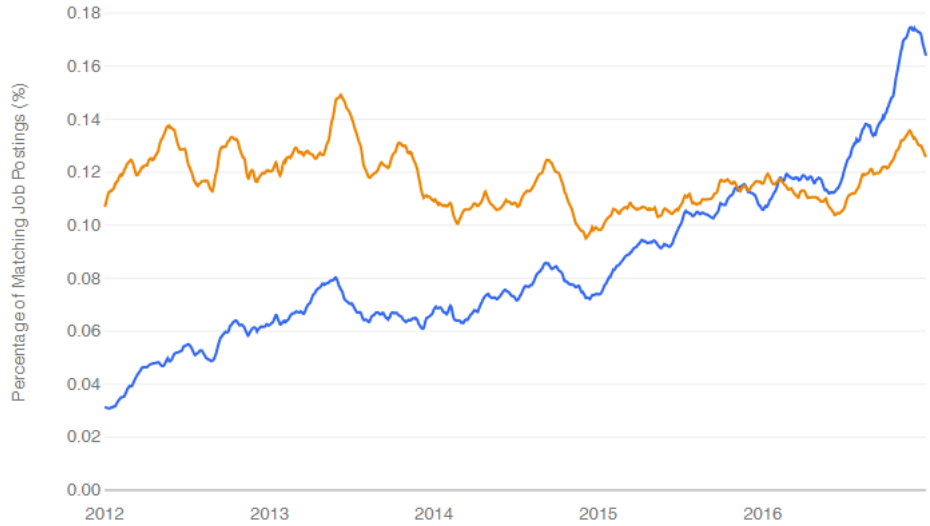
Delving into more complex functions: Recursive and Replacement functions.

Lakshay Badlani

10/28/2017

Introduction

Over the recent years R as a programming language has been gaining more popularity[1]. This is especially the case if we look at trends specifically for Data Science jobs (please refer to graph shown below). As a result, from this one can infer that as a language is very suited to data analysis. The reason for this is that R allows the user to create and customize complex functions suited to any task especially the manipulation of data sets. Moreover they improve the readability of written code, allow us to perform repeated work in a convenient manner and can be applied to any general set of parameters (with their respective values determined by the user)[2]. Thus, their importance cannot be understated and as Hadley Wickham puts it "functions are the building block of R."[3]



The above graph showcases the steady growth of R as a tool in data analysis possibly due to the multitude of functions you can create in it.[4]

In my opinion many people, especially those new to programming, seem to undervalue the importance of functions as most newcomers take simple processes such as reading in tables or plotting a graph for granted whilst in fact the ability to perform these tasks so easily is due to functions. Consequently, my motivation for dedicating this post to complex functions is three fold. Firstly, given the fact they are the backbone of any data analysis projects (a particular interest of mine) becoming more comfortable with using and creating functions, especially custom ones, is necessary in order to make more headway when it comes to extracting information from data. Connected to this having a greater mastery over functions not only improves my personal skill set but makes me a more well-rounded data scientist and statistician, Additionally in order to gain a real understanding of the power of R I believe one needs to be able well versed in many different types of functions which is why I have chosen to focus on two in this post, namely recursive and replacement functions. For each different type, I will try to explain their general format, situations where they can be used, give an example of how it is used and lastly why it might be advantageous to use them.

Recursive functions

Recursive functions are special due to the fact that they have the intrinsic property of calling themselves in the return statement of the function. Whilst this may catch many people off guard upon first glance since it appears to create an infinite loop, it actually allows us to utilize a special programming technique. Namely recursive functions allow us to break down a problem into its smallest subparts, solve for those smaller individual subsets, and then solve put all those solutions back together to solve our initial problem[5]. We avoid the aforementioned infinite loop by introducing what is known as a base case which represents the smallest block of a problem and tells us that we have to start to solve the original problem from this point. In other words it actually returns a value instead of calling on the function again.

The generality of solution that can be attained is what makes this so powerful. In addition, it gives us another method of approaching problems in which we might have to use a for loop (these are known as iterative solutions) as it instead of going through every value one after the next we look at all the individual values, solve for them and combine it to the end. In order to aid understanding of this type of function I looked up a concrete example to help me see the concept in practice. For example

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

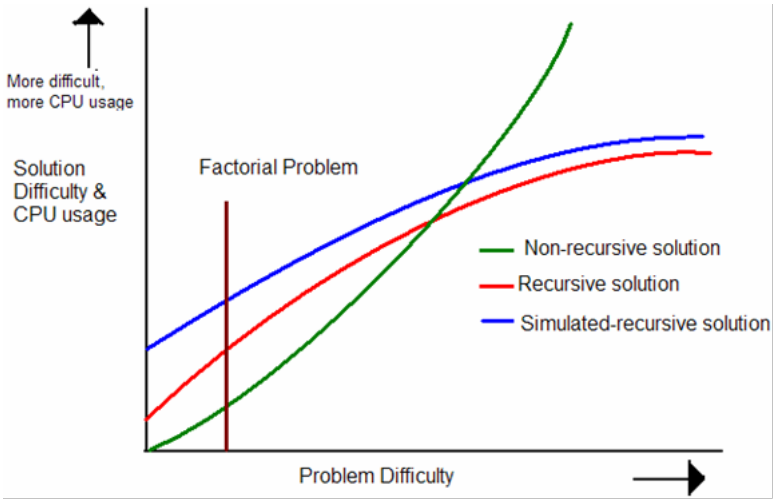
or more succinctly put

$$4! = 4 \times 3!$$

This can be broken down even further since we know

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

As a result one can see how we broke down the factorial of 4 into the factorial of 2 multiplied by 2 numbers. Since we are using such small numbers this might seem quite useless however if we were computing much larger factorials having to calculate a factorial of even a slightly smaller number saves a lot of computational time.



This graph showcases the differences between iterative/recursive solutions and showcases how there are certain situations where recursive solutions are better and vice versa.[6]

Coded example

Sticking to the factorial example showcased above and the described general structure of recursive functions I attempted to code a function to solve the factorial of any number in a recursive manner.

```
recursive_factorial = function(x) { #Defining a recursive function follows the normal method.

  #The following is our base case since it is the smallest number with an actual factorial the factorial of 1 is j
ust 1.
  if (x == 1) {
    return (1)

  #The following is the recursive case which calls the function on itself and breaks the proble down further.
  } else {
    return (x*recursive_factorial(x-1))
  }
}
```

To test that this function performs as it is supposed to I compared the output to the input built factorial function on an arbitrary number.

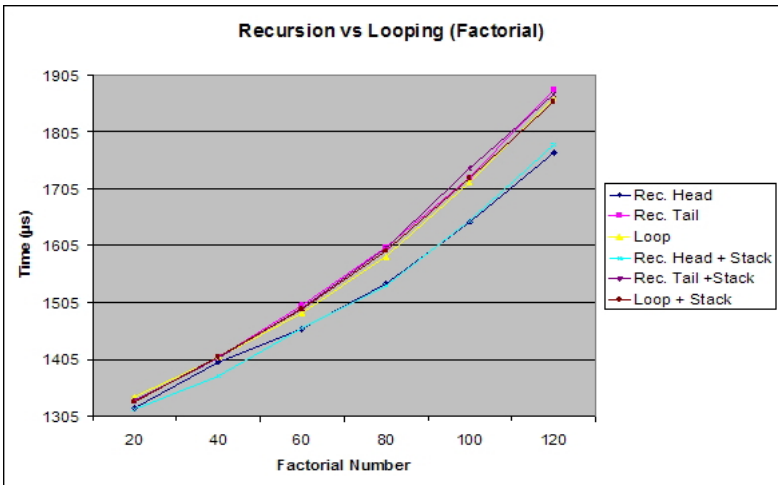
```
recursive_factorial(10) == factorial(10)
```

```
## [1] TRUE
```

Advantages and Disadvantages of Recursive functions

Personally, whilst exploring these functions for myself I have found several advantages and disadvantages of utilizing this type of function. First of all one of the things that immediately stood out was that the readability of the function and my code in general definitely improved. In addition, there were certain problems in which thinking in a recursive manner was much easier. An example of this was the fibonacci numbers which lends itself very easily to the school of thought. However, some disadvantages I experienced was that trying to apply recursion to some situations just was not as intuitive as the equivalent iterative method. Moreover, debugging the function when it did not perform as intended was much harder.

Using online references, I found additional advantages and disadvantages of recursive functions that might not have been apparent. One particularly interesting disadvantage I discovered was the fact that these sort of functions use more memory because of the number of function calls that are made[7]. However it seems like this is counterbalanced by the fact that there can be an improvement in runtime for these function in comparison to iterative functions.[8]



The graph above shows how for the recursive solution for factorial problem as we demonstrated previously is in fact faster than the iterative solution as the number we try to compute the factorial of increases.[9]

Replacement Functions

A replacement function is a special type of function in r that will allow you to set or modify the value of its first function in place[10]. What makes this behavior different to normal is that in most cases with functions if we pass in a parameter and then set it equal to something else within the function its global value does not change [11]. However, in the case of a replacement function its value will change hence the name. The key characteristic of this type of function is that their names take the form “xyz<-” where xyz can be any valid combination of characters and letters. The body of the function follows the same rules as the functions we have seen in class however when calling the function, we only pass in the variable we want to change (this will be made clearer with the example below).

The reasons these functions can be useful is that it enables us to avoid having to constantly reassign variables and also assign temporary variables to store values that we might only need in the intermediary stage of data processing. As a result replacement functions are heavily used when we need to alter one variable, usually a list given its versatility as a storage type, multiple times. Below is a simple example of how to define and utilize a replacement function.

Coded Example

```
#Notice how the naming convention for replacement functions is different.

'multiply_first_last<-' = function(x, value) { # They always take in two different parameters, the first is the va
riable to modify and the second is how to manipulate it.*

  # Below we find the first and last elements of our variable and multiply it appropriately.
  x[1] = x[1] * value
  x[length(x)] = x[length(x)] * value
  x

  # These assignments should now have changed the values in x permanently.
}
```

*This formatting for defining a function is unique in R and makes sure that the parser knows it will be creating a replacement function. [12]

In order to test if this function actually worked I passed in an arbitrary vector and checked the output.

```
y = c(1, 2, 3, 4, 5) # Create an arbitrary vector named y

multiply_first_last(y) <- 3 # We have to pass in arguments in this format due to how R parses this type of functio
n.

y
```

```
## [1]  3  2  3  4 15
```

Advantages and Disadvantages of Replacement Functions

Since replacement functions were mostly created for convenience sake and there function can be easily replicated by other lines of code there is very little to talk about in terms of advantages and disadvantages. However I will attempt to give as many as possible. One clear advantage is that it definitely makes your code look much more readable and clear especially if you are repeatedly performing the same manipulation. Moreover it results in less temporary variables having to be created and thus can be less confusing for the user themselves when they are writing code. Lastly since it is in the form of a function it enables many modifications to be performed on a variable using just one function call make your code more succinct and cleaner.

On the other hand whilst we might expect to save some memory usage utilizing them as we have less variables created, replacement functions only “act” like they modify arguments in place as Hadley Wickham puts it. This is because they actually create a copy of the variable in another part of memory, modify it and then reassign the variable behind the scenes[13]. Thus we still use as much memory as we would have if just utilized another variable.

Conclusion

Overall, whilst making this post I learnt a lot about functions but one thing in particular stood out: functions have a multitude of applications and almost always make data processing a less complicated process. As a result I believe one should use functions as often as possible and wherever possible given the convenience they bring and the fact they make your code more readable. In addition being knowledgeable and comfortable about different types of functions including in built ones will make your life much easier! I say this because having various ways to approach a problem often yields a solution in a smaller timeframe. A cool website to help you become more comfortable with functions and keep you up to date with interesting packages is: http://rfunction.com/.

References

1. Muenchen, Robert A. “The Popularity of Data Science Software.” R4stats, www.r4stats.com/articles/popularity/.↩
2. Fitzle LLC. “Lesson 3.6: Advantages of Functions.” Youtube.com. Vanderbilt University. June 13th 2015. https://www.youtube.com/watch?v=NjWPheOtdws↩
3. Wickham, Hadley. “Functions.” Advanced R, www.adv-r.had.co.nz/Functions.html.↩
4. Muenchen, Robert A. “The Popularity of Data Science Software.” R4stats, www.r4stats.com/articles/popularity/.↩
5. French, Jason A. “Recursion in R.” Jason French Blog, Northwestern University, July 26th 2014, www.jason-french.com/blog/2014/07/26/recursion-in-r/.↩

6. Unknown. “Recursion made simple.” Code Project, 27 July 2012, www.codeproject.com/Articles/32873/Recursion-made-simple.↩
7. Unknown. Recursion. Computer Science at UWISC, University of Wisconsin Madison, pages.cs.wisc.edu/~vernon/cs367/notes/6.RECURSION.html.↩
8. Paskin, Brian. “Looping versus recursion for improved application performance.” IBM Developer Works, IBM, 17 July 2013, www.ibm.com/developerworks/websphere/techjournal/1307_col_paskin/1307_col_paskin.html.↩
9. Paskin, Brian. “Looping versus recursion for improved application performance.” IBM Developer Works, IBM, 17 July 2013, www.ibm.com/developerworks/websphere/techjournal/1307_col_paskin/1307_col_paskin.html.↩
10. Wickham, Hadley. “Functions.” Advanced R, www.adv-r.had.co.nz/Functions.html.↩
11. Matloff, Norman. “The Art of R Programming: A Tour of Statistical Software Design”. 1st Edition. No Starch Press, 2011. Page 75.↩
12. Unknown. “Subset 3.4.4: Subset Assignment.” R Language Definition, CRAN Project, cran.r-project.org/doc/manuals/R-lang.html#Subset-assignment.↩

unctions.” Advanced R, www.adv-r.had.co.nz/Functions.html.↩