Recursion: A Useful Tool for Composing Functions

Wenqu Wang

Introduction

We just learned about how to make a simple function in R. However, sometimes the problem for us to solve is more complex and we need more tools. A common programming strategy is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results. And recursion is one of the most powerful tools that apply such idea.

The purpose of this post is to introduce you the powerful tool of recursion in making R functions. In this post, I will discuss the definition and application of recursion with concrete examples and graphs.

A simple introductory example: factorial

Let's start with a simple example. Suppose you want to compute the factorial of an integer n using a R function.

The most common way to compute the factorial of an integer n, which is obtained by multiplying n by (n - 1), then (n - 2), all the way down to 1. A loop would be useful for this way of solving the problem.

```
#factorial of n using a for loop
factorial01 <- function(n) {
  result = 1
  for (i in 1 : n) {
    result = result * i
  }
  result
}
```

```
#factorial of 4
factorial01(4)
```

```
## [1] 24
```

Though this way of solving the problem is great, it looks a bit complex and hard to understand. Better way of solving the problem do exist, that is using recursion.

Let's look at the problem from a different respective. The factorial of n is actually equals to n times the factorial of n - 1; and the factorial of 1 is just itself. Take this approach, we could break the problem into small sub-problems and finally find the result by summing the result of all sub-problems.

```
#factorial of n using recursion
factorial02 <- function(n) {
  if (n == 1) {
    n
  } else {
    n * factorial02(n - 1)
  }
}
```

```
#factorial of 4
factorial02(4)
```

```
## [1] 24
```

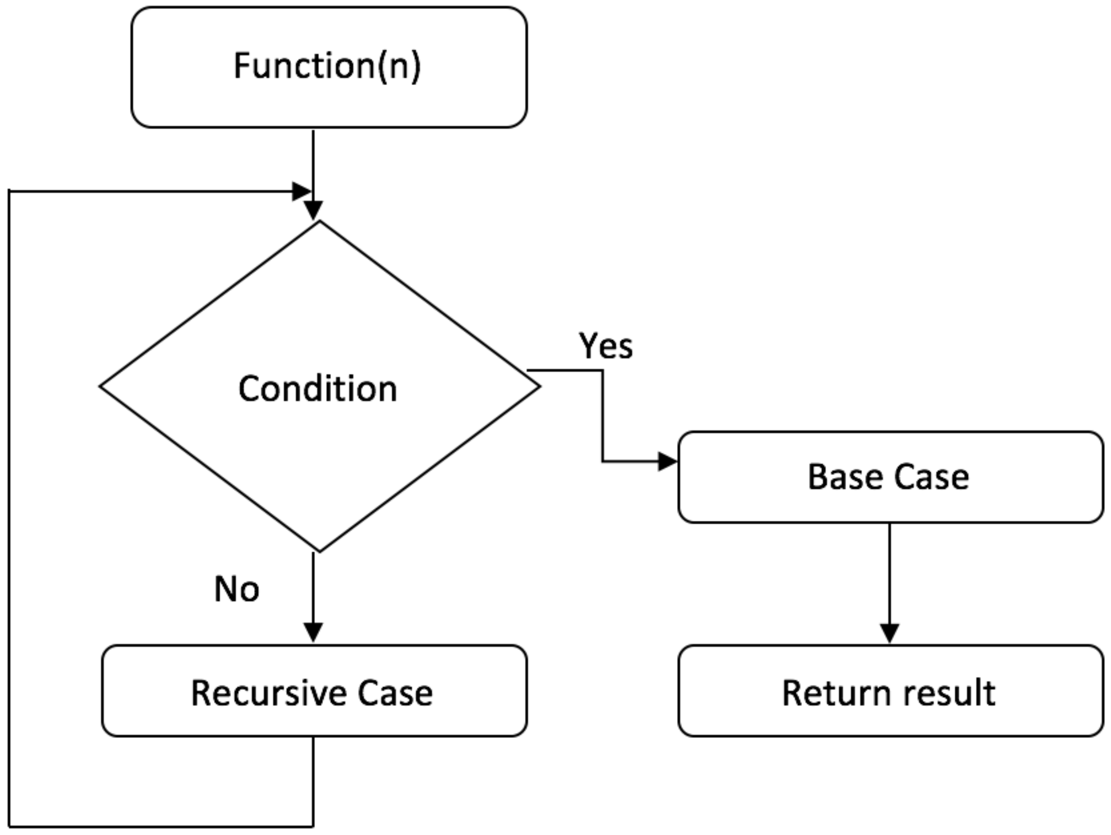
From the above example, we could see the power of recursion. It makes the function easier to understand and more concise.

Definition of recursion

A class of objects or methods exhibit recursive behavior when they can be defined by the following two properties:

- A simple **base case**(or cases) -- a terminating scenario that does not use recursion to produce an answer
- A **recursive case**(or cases) -- a set of rules that reduce all other cases toward the base case

To make the process of recursion easier to understand, I drew a flowchart for you.



Let's take a look at the previous example. The base case for our factorial02 function is returning n when n equals to 1; and the recursive case is returning the product of n and factorial02(n - 1) when n does not equal to 1.

```
#factorial of n using recursion
factorial02 <- function(n) {
  if (n == 1) { #condition
    n #base case
  } else {
    n * factorial02(n - 1) #recursive case
  }
}
```

More advanced recursion: tree recursion

The process of solving a recursive problem involving breaking down the problem and find a base case and a recursive case. However, for some problem, one single recursive case is not enough for solving the problem. In such cases, we need tree recursion to help us figure out the solution.

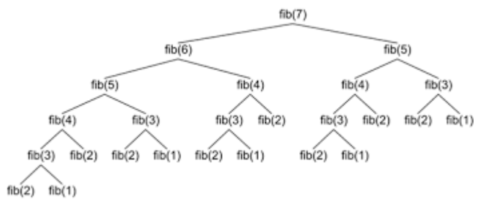
As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two.

When we are trying to break the problem down, we can find that the nth fibonacci numbers equals the sum of the (n-1)th and (n-2)th fibonacci numbers. So we need two recursive cases to complete our recursion. We also need two base cases that is the first and second fibonacci number equal to 0 and 1 respectively.

```r
#fibonacci sequence using tree recursion
fib <- function(n) {
  if (n == 1) {
    0 #base case 1: the first fibonacci number
  } else if (n == 2) {
    1 #base case 2: the second fibonacci number
  } else {
    fib(n - 2) + fib(n - 1) #recursive cases
  }
}
```

```r
# the sixth Fibonacci number
fib(6)
```

```
## [1] 5
```



As shown in the picture above, each fibonacci number could be broken into two smaller fibonacci numbers. The largest number is like the root of a tree and the smaller ones are like branches of a tree, and that's why we call such structure tree recursion.

Pros and Cons of recursion

At this point, we have known that recursion is a good way of solving a problem. However, for each recursive solution, we could come up with a iterative solution by using loops. So, why would we prefer recursion over iteration? Let's look at an example.

We've already came up with a recursive solution to the Fibonacci sequence, and now we want an interative one.

```r
#fibonacci sequence using a for loop
fib2 <- function(n) {
  if (n <= 2) {
    n - 1
  } else {
  curr = 1
  prev = 1
  for (i in 3:(n-1)) {
    temp = curr
    curr = curr + prev
    prev = temp
  }
  curr
  }
}
```

```r
# the sixth Fibonacci number
fib2(6)
```

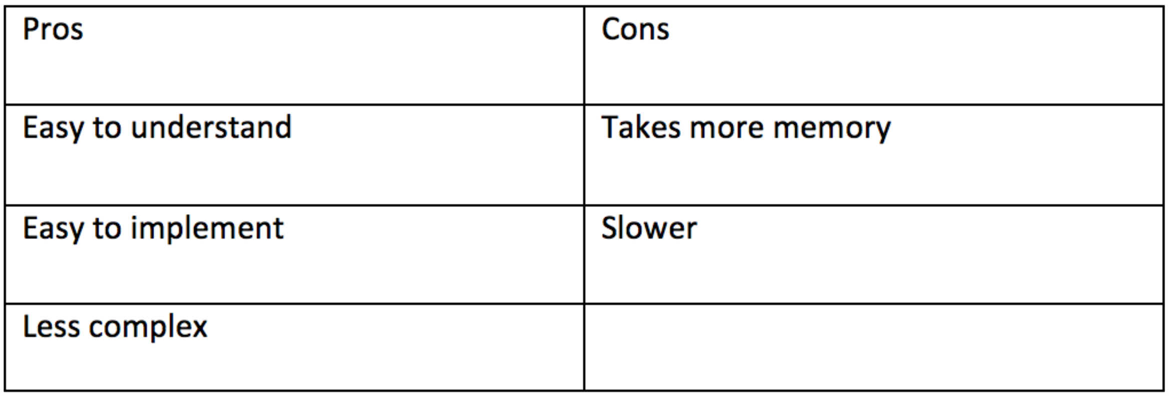
```
## [1] 5
```

When first looking this chunk of code, you are definitely confused about what is the meaning of each line. However, when you look at the recursive one, you can understand what each line of code means immediately. So, the first advantage of recursion is increased clarity and easier to implement.

Also, this solution takes 14 lines while our previous one takes 9 lines. Comparing to iteration, recursion makes our solution less complex.

However, recursion usually takes more time than iteration as it will repeat a lot of calls to itself. For example, when we are trying to compute "fib(7)", we need to compute fib(5) twice, which waste a significant amount of run time.

To sum up, I made a chart of the advantages and disadvantages of recursion.



Pros	Cons
Easy to understand	Takes more memory
Easy to implement	Slower
Less complex	

Conclusion

After reading this post, I hope you have some thought about how recursion works in programming R functions. Recursion offers a unique approach of solving a problem, that is dividing a problem into smaller sub-problems. By this idea, recursive code is usually easy to understand and concise. Although it is slower and requires more memories, recursion is good enough to be a powerful tool for programmers. In the future, I hope recursion will help you biuld more usful functions in R.

References

1.Recursion Algorithms
2.How Recursion Works
3.R Recursive Functions
4.Definition of Recursion
5.Fibonacci Tree
6.Pros and Cons of Recursion
7.Pros and Cons of Recursion