Recursion in R

Warren
October 28, 2017

setup for ggplot

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.4.2
```

Introduction

In lecture, we learned about loops (iterations). The main idea of it is to repeatly do a task without typing out the same code over and over again. But there are more than one way to accomplish this task!
In this post, I would like to share with you another way of repeating in R: recursion.

A first encounter with recursion

Let's suppose we would like to calculate factorials of natural numbers. Of course, we have the built-in function factorial() at our hand. But, for educational purposes, let's suppose it does not exist for now.
If we are to do this with iterations, we would have this function:

```
iterFact = function(x){  ##Iterative Factorial
  prod = 1
  for(i in 1:x) prod = prod * i
  return (prod)
}
```

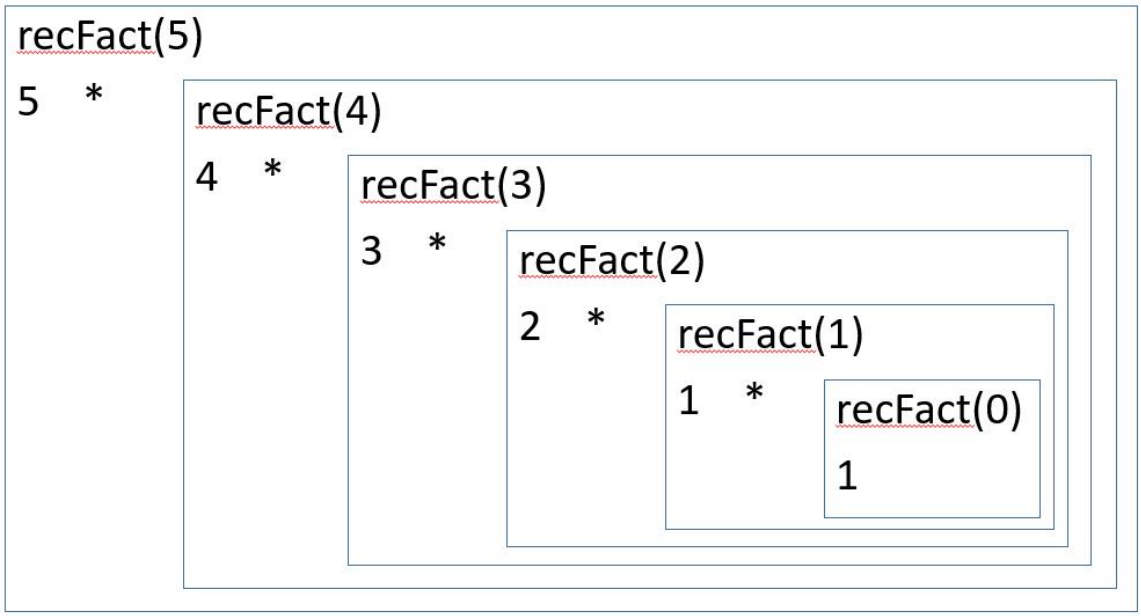
But let's inspect factorials more. Suppose we are calculatiing 5! by hand, then 5! = 5 x 4 x 3 x 2 x 1. Notice 4 x 3 x 2 x 1 = 4!, and thus 5! = 5 x 4!. This means that when we compute 5!, we are also secretly computing 4! (and similarly, 3!, 2! and 1!) as well. To generalize: when we compute n!, we compute n x (n - 1)!. Let's write a function base on this intuition:

```
recFact = function(x) { ##Recursive Factorial
  return (x * recFact(x-1))
}
```

Wait a sec! Do you see something wrong with this function? For example, what will happen if we call the function recFact(5)?
The problem is that this function does not know when to stop. Taking the recFact(5) example, the function will calculate 5 x 4 x 3 x 2 x 1 x 0 x (-1) x (-2) x ... Although this expression evaluates to 0, the computer will keep multiplying 0 with some number and never stop. Thus we need to give the function a brake!
Factorials stop a 0, where mathematicians define 0! = 1. Adding this condition to the original function, we get:

```
recFact = function(x) { ##Recursive Factorial
  if (x == 0) return (1)
  else return (x * recFact(x-1))
}
```

To better understand this program, let me include a visualization of recFact(5):



Lets call our friend factorial() back and see if we got the correct functions:

```
factorial(5)
```

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

```
iterFact(5)
```

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

```
recFact(5)
```

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

```
factorial(50)
```

```
## [1] 3.041409e+64
```

```
iterFact(50)
```

```
## [1] 3.041409e+64
```

```
recFact(50)
```

```
## [1] 3.041409e+64
```

```
factorial(100)
```

```
## [1] 9.332622e+157
```

```
iterFact(100)
```

```
## [1] 9.332622e+157
```

```
recFact(100)
```

```
## [1] 9.332622e+157
```

Hmmm, seems good!

Analyzing recursion

Recursion, just as its names suggests, means re-occuring. In R, the functions are smart enough so that they can call themselves. For example, in our recFact function, it repeatly calls itself when the function's input x (or parameter, in computer science language) is not 0. We call those functions "recursive functions", and this method of repeating "recursion".

Recursive functions must have two parts. The first one is the recursive step, A.K.A. where the recursion happens. In this part the function calls itself with (a) new parameter(s) such that it repeats itself while evaluating. In our recFact example, "return (x * recFact(x-1))" is the recursive step. The other one is the base case, or where the function stops (the brake). Just like in while loops or in repeat loops, we need to specify the program a stopping point so that it will not go on forever. In our recFact example, "if (x == 0) return (1)" is the base case.

Another way of thinking about the base case is that, as its the name suggests, this case is the most trivial, the most fundamental case of our function. For example, 0! is part of the definition of factorials, and needs no calculation as it is defined to be 1. Thus, it becomes the base case of our recFact function. Therefore, when writing recursive functions, it is quite often easier to think about the base case first, and then build the recursive step on top of it.
If you know induction, recursion is very similar to it, but the other way around. In induction we prove that the base case (usually n=1 or n=0) first, and then assume case n is true. By proving that case n+1 is true, we prove the whole statement. In recursions, we start with case n. By gradually reducing n, we reach the trivial case (n=1 or n=0 usually).

Why recursion?

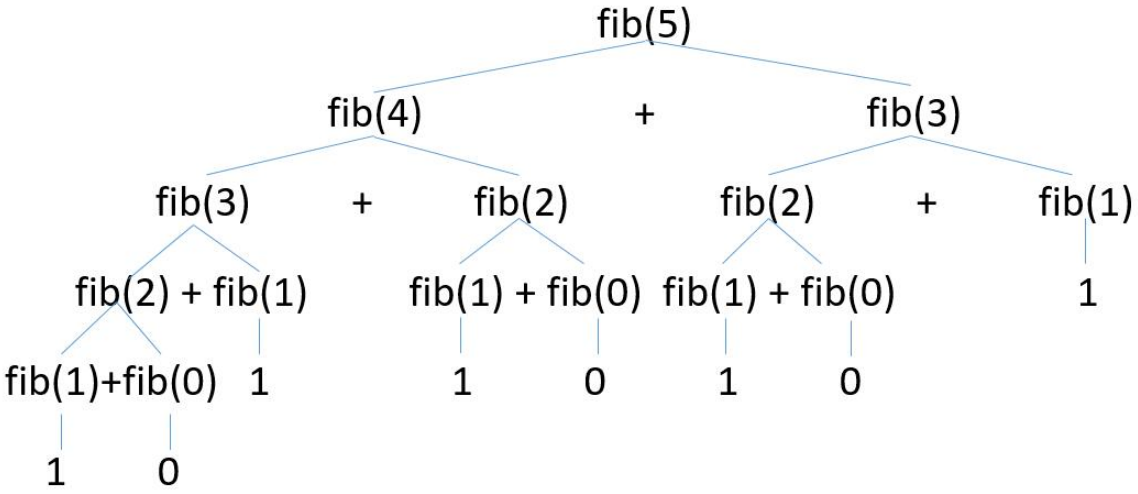
But, why do we need recursions? This concept seems hard to understand, and the code is not as easy to read as the iterative version of the same function!

Well, although for problems like factorials or summing things up, recursion seems to be unnecessary at all, there are problems where the problems themselves are built recursivly! For those problems, recursion is definitely the best tool to use.

Let's look at a famous example: Fibonacci numbers. If you don't know what they are, Fibonacci numbers are a sequence of numbers starting with 0 and 1, and every other number equals the sum of the previous two numbers. The first few terms of Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55... When counting Fibonacci numbers, we often use 0-indexing (counting from 0: 0, 1, 2, 3...) instead of 1-indexing (counting from 1: 1, 2, 3, 4...), so the 0th fibonacci number is 0, the 1st is 1, the 2nd is 1, the 3rd is 2...

Now let's write a program that returns the nth Fibonacci number, fib(n). Before implementing it, let's take a deep look at the definition of Fibonacci numbers. We should be able to realize that Fibonacci numbers are defined recursively: every number equals the sum of the previous two numbers. For example, if we want to calculate the 5th number (fib(5)), we will add the 4th number and 3th number; to find the 4th number and 3th number, we need to calculate the 3th number and the 2nd number, as well as the 2nd number and 1st number, etc. The base case for this recursive definition is that, the 1st number is 1 and the 0th number is 0. Here is a visualization of fib(5):

A visualization of fib(5)



Now that we fully understand what fibonacci numbers are, let us write the actual function.

```
fib = function(n){ ## a function that finds the nth fibonacci number
  if (n == 0) return (0) ##base case 0: return 0
  if (n == 1) return (1) ##base case 1: return 1
  return (fib(n-1) + fib(n-2)) ##recursive step: return the sum of previous two fibonacci numbers
}
```

```
## some outputs to verify the correctness of our program
fib(0)
```

```
## [1] 0
```

```
fib(1)
```

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

```
fib(2)
```

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

```
fib(3)
```

```
## [1] 2
```

```
fib(4)
```

```
## [1] 3
```

```
fib(5)
```

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

```
fib(10)
```

```
## [1] 55
```

We can see that this function is very easy to implement once we understand recursion. Now think about iteration: how would you implement an iterative version of fib(n), iterFib(n)?

It is not easy, or at least not as easy as fib(n), right? We need to move up from 0 to n, get the current number, add it with the previous number to get a new number, and move the "current" number to "previous" number, and then move the "new" number to "current" number, and repeat...

```
iterFib = function(n){ ## a function that finds the nth fibonacci number, using iteration
  if (n == 0) return(0)
  if (n == 1) return(1) ##base cases are still needed,
                        ##because they are part of the definition of Fibonacci numbers
  x = 0 ##let x be the "previous" number
  y = 1 ##let y be the "current" number
  z = 1 ##let z be the "new" number
  for(i in 1:n) { ##We already have the 0th term, so we do n loops to get to the nth term
    x = y ##move the "current" number to "previous" number
    y = z ##move the "new" number to "current" number
    z = x + y ##calculate the next "new" number
  }
  return(x) ##return the desired value.
            ##Notice that at the end of the loop, the "current" term is the (n+1)th term.
            ##Thus the "previous" term, x, is the desired value
}
```

```
## some outputs for verification
iterFib(0)
```

```
## [1] 0
```

```
iterFib(1)
```

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

```
iterFib(2)
```

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

```
iterFib(3)
```

```
## [1] 2
```

```
iterFib(4)
```

```
## [1] 3
```

```
iterFib(5)
```

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

```
iterFib(10)
```

```
## [1] 55
```

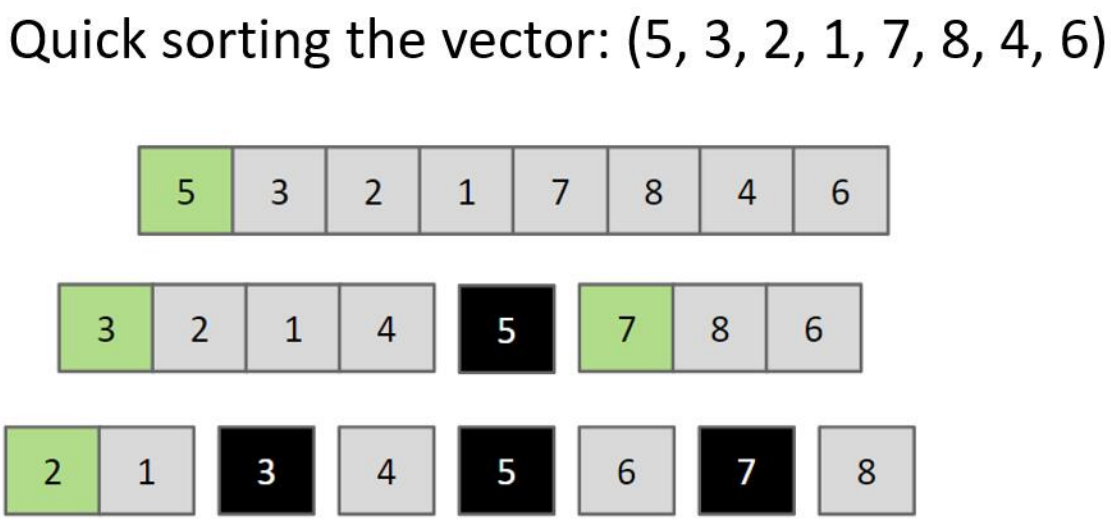
That took us some time, and a lot of thinking! You can see that iteration, although viable, might not be the best solution to this problem (in the sense of easiness to write or read. We will discuss some other way to judge a function later).

There are a lot of other problems where the problem itself is recursive and thus recursive functions are easy and intuitive solutions to them. Some examples include:
Using the Euclidean Algorithm to find the greatest common divisor of two positive integers (If you take Math 55 or CS 70);
Traversing a graph. For example, a tree data structure. (If you take CS 61B. But hey, if you do take CS 61B, you probably do not need to read this post!)

There are also stat 133 related examples! Let's inspect two more:

1. Finding the determinant of a matrix. Remember when we were doing PCA, we calculated eigenvalues and eigenvectors of a matrix? One part of it is to find the determinant of a matrix. As you have taken (or is taking) Math 54, you probably know the recursive definition of calculating the determinant: the base case is an 2x2 matrix, where we calculate entry[1,1]xentry[2,2] - entry[1,2]xentry[2,1]. For square nxn matrices A larger than 2x2, we pick a row r, for all the columns, do the following: lets say the column is c. Remove row r and column c from the matrix A to form a new matrix B. Calculate the determinant of B using this same method (A.K.A. recursion), and call the result Det(B). Calculate (-1) ^ (r+C) x A[r,c] x Det(B). Let's call the result k. Once we have do this to all the columns and get all the k's, the determinant of A is equal to the sum of all the k's.
   Obviously, this algorithm will be a mixture of iteration (going through all the columns for a row) and recursion (calculate the determinant of sub-matrices). Fully implementing this algorithm can get pretty lengthy and hard, but showing how the idea of recursion can be used in here is sufficient for the purpose of this post. (And, btw, the function det() from R base package gives you the determinant of a matrix, so you do not need to implement it at all!)

2. Sorting. When we use functions order() (or sort()), we are sorting things. There are a lot of ways to sort, and depending on what the inputs are, each way of sorting can vary in speed. But generally speaking, the fastest comparison sort (that is, sort based on comparing elements) uses recursion. Because of its speediness, this sorting algorithm is called "quick sort".
   Let's suppose we have a list (for example, a vector) of elements. The base case is when the list contains only one or no element, which means we do not need to sort anything. Now suppose the list has more than one element. We randomly choose an element from this list, and call it the pivot. Next, we form 3 new vectors: left, middle and right. We now look at the elements in the original vector one by one. If an element is less then the pivot, it goes into the "left" vector. If the element is equal to the pivot, it goes into the "middle" vector. If the element is larger than the pivot, it goes into the "right" vector. For the "left" and "right" vectors, quick sort them again (we don't need to sort the middle one because all its elements are equal to the pivot, A.K.A. it is sorted). At the end, concatenate (i.e., forming a vector by connecting) "left", "middle", and "right". Because we know "left", "middle", and "right" are all sorted, and by how we put elements into "left", "middle", and "right", we know that the concatenated vector must be sorted.
   Let's look at an example. Suppose we want to run quick sort on this vector of integers: c(5, 3, 2, 1, 7, 8, 4, 6). For simplicity, let us suppose that the randomly chosen pivots somehow are always the first elements. The following image shows how the algorithm runs:



We can see that at the end, the whole vector is broken down into vectors with one or 0 element (base case). Now if we concatenate every list, we will have our desired output: (1, 2, 3, 4, 5, 6, 7, 8)
Let us implement a quick sort algorithm for vectors, with a randomized pivot.

```
quickSort = function(vec) { ##a quick sort function for vectors
  if(length(vec) <= 1) return(vec) ##base case: length is 1 or 0, nothing to sort
  left = c()
  middle = c()
  right = c() ## initialize 3 vectors for left, middle and right
  pivot = sample(vec, 1) ##randomly choose a pivot
  for(i in vec){ ##inspect all the elements in the vector
    if(i == pivot) middle = c(middle, i) ##if it is equal to the pivot, it goes to "middle"
    if(i < pivot) left = c(left, i) ##if it is less then the pivot, it goes to "left"
    if(i > pivot) right = c(right, i) ##if it is greater then the pivot, it goes to "right"
  }
  left = quickSort(left) ##recursive step: quick sort the "left" vector
  right = quickSort(right) ##recursive step: quick sort the "right" vector
  return(c(left, middle, right)) ##return the concatenated vector
}
```

```
##tests for verifications
vec1 = c(5, 3, 2, 1, 7, 8, 4, 6)
quickSort(vec1)
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
vec2 = c(1.6, 1.2, 1.5, 1.3, 1.4, 1.1)
quickSort(vec2)
```

```
## [1] 1.1 1.2 1.3 1.4 1.5 1.6
```

```
vec3 = c("s", "t", "a", "t", "1", "3", "3", "f", "a", "l", "l", "2", "0", "1", "7")
quickSort(vec3)
```

```
##  [1] "0" "1" "1" "2" "3" "3" "7" "a" "a" "f" "l" "l" "s" "t" "t"
```

Notice here that elements of the vector are comparable (that is, it makes sense and exists a way to compare two elements in the vector) is essential for our quickSort function to work. Even if something is comparable, there may be various ways to compare. For example, vectors, by default, are not comparable in R. But we can define ways to compare vectors, including but are not limited to: comparing the length of the vectors; comparing the first element of the vectors; comparing the largest elements of the vectors, etc. There are implementations of more generalized quickSort functions which also take into account uncomparable elements, and comparator functions that defines how to compare things. Here is an example of one of them from package rje v1.9: https://www.rdocumentation.org/packages/rje/versions/1.9/topics/quickSort

Advanced: Recursion VS Iteration, V2.0

We've been boasting about recursion for quite a while. We've said that recursion solve some problems way easier than iteration does; that the code looks elegant; that it gave us the fastest comparation sort, etc. Recursion seems to be good problem solvers, but at what cost?

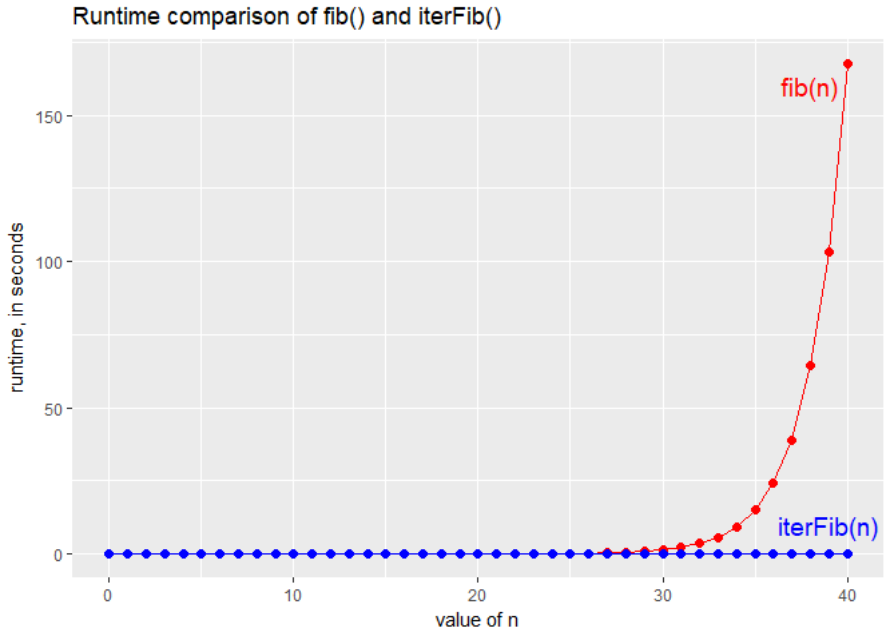
It turns out that recursion are inefficient in terms of space (memory) and time. Let's look at them one by one.

Recursion uses a lot of space. Everytime you call function, imagine opening a tiny application on your computer. Those applications are not closed until you reach the return statement. As recursive functions call themselves, wait for the results from recursive steps, and then return, they open a lot of applications at the same time. Opening apps takes more memory then doing stuff inside the app (declaring variables, performing calculations, etc). Thus, recursions takes a lot more memories. Here is an analogy. Let's say we are using the Google Chrome internet browser (which is notorious for its memory consumption). We open a youtube video page. It is usually OK for us to play the video, like the video, write a comment, fast forward, etc. However, imagine opening 100 tabs of youtube videos and play all of them at the same time. Your computer is likely to give you a warning message that you run out of memory soon.

Recursions are also usually slower than iterations. The reasoning behind involves runtime analysis, which is definitely beyond the scope of this course. So, to persuade you, let's just look at an example. Remember our old friend, fib() and iterFib(), who calculate fibonacci numbers? In the "code" folder, I wrote an R script to measure the runtime for fib(n) and iterFib(n), where n is from 0 to 40. I recorded the results in two csv files inside the "data" folder. You can look at them for

your reference. Let's draw a graph to analyze the results.

```
fibTime = read.csv("../data/fibTime.csv")
fibTime = fibTime[, 1]
iterFibTime = read.csv("../data/iterFibTime.csv")
iterFibTime = iterFibTime[, 1]
ggplot(x = 0:(length(fibTime) - 1)) +
  ggtitle("Runtime comparison of fib() and iterFib()") +
  xlab("value of n") + ylab("runtime, in seconds") +
  geom_line(aes(x = 0:40, y = fibTime), color = "red") +
  geom_line(aes(x = 0:40, y = iterFibTime), color = "blue") +
  geom_point(aes(x = 0:40, y = fibTime), color = "red", size = 2) +
  geom_point(aes(x = 0:40, y = iterFibTime), color = "blue", size = 2) +
  geom_text(aes(x = 38, y = 160), label = "fib(n)", size = 5, color = "red") +
  geom_text(aes(x = 39, y = 10), label = "iterFib(n)", size = 5, color = "blue")
```



From this graph, we can see that as n grows larger, fib(n) increases way faster than iterFib(n) does. If you still do not believe me, you can run the r script with larger n's to verify this, but be warned: it will take a VERY LONG time.

Summarization and conclusion

The short version of the take home message is that: recursion is a method of repetition in R other than iteration (loops). A recursive function is a function that calls itself and has two parts: base case and recursive steps. Recursion is a great tool for problem solving.
However, if you are into programming, you should also remember: recursion has its own limits and disadvantages. Comparing to iterations, it is very inefficient in space and time. Always think about which method is better before you write your function!

Reference:

1. https://cs61a.org/assets/slides/06-Recursion_1pp.pdf
2. https://cs61a.org/assets/slides/07-Tree_Recursion_1pp.pdf
3. https://www.programiz.com/r-programming/recursion
4. https://stackoverflow.com/questions/21710756/recursion-vs-iteration-fibonacci-sequence
5. https://www.rdocumentation.org/packages/base/versions/3.4.1/topics/sort
6. https://www.quora.com/What-is-the-difference-between-an-iterative-algorithm-and-recursive-algorithm
7. https://gist.github.com/primaryobjects/c9020cf98168930eb019
8. https://docs.google.com/presentation/d/1V_xvL2h-SQhHojqCIrQ-E3xqcjZKUmMt8otZcsUx3E8/edit#slide=id.g4661758db_1627
9. https://www.rdocumentation.org/packages/rje/versions/1.9/topics/quickSort