post01-anbo-cao

Anbo Cao

10/31/2017

Functions in R

Anbo Cao

10/31/2017

Introduction

Functions are one of the key ideas in many programing languages. Every programming languages use functions, such as C, Python, and others. Of course, R is one of them. The purpose of a function is to reduce many repeat works and formulate a well organized coding styles. I was surprised that we learned functions in r after so many weeks working on R. In the post, I will revisited the basics of the R functions. Futhermore, I will introduce some advanced R function(Recursion) that is even more powerful than many people can think.

Function revisit

The basics of functions is very simple, the user specify the inputing arguments. Then the function do some process with it and return a result. Here is a basic examples similar to the previous labs and hws:

```r
# Reading examples
nba = read.csv("./nba2017-players.csv")

# Body of the functions
year_started = function(nba) {
  return(2017 - nba$experience)
}

# Calculate the year that the player started
nba$year_started = year_started(nba)

# Showing the first 10 results
head(nba$year_started, 10)
```

```
##  [1] 2008 2006 2011 2017 2008 2012 2013 2015 2017 2011
```

In the above example, the function calculate the starting year of the nba players. Similarly, if we have other datas we want to process, we can repeatly use this function to calculate the years that the player started.

The R language itself provide many useful statistic functions like ‘mean’, ‘sum’, ‘sd’, and others. Furthermore, many packages and usefull tools are also implemented in functions like ggplot2. ggplot use ggplot() and other functions (ex. geom_point(), abline() …) to create beautiful graphics. The reason for the ggplot is so easy to use is because the functions.

Recursive functions

In R, many advanced idea of functions are also supported. One of the example is recursion. The definition of the recursion is that “A function that calls itself is called a recursive function and this technique is known as recursion.” The general idea of recursive functions is to breaking a large, diffcult problem into many small problem. Then it is much easier to be solved. However, because of espically property that the function is calling itself, usually the recusive function will have a worse performance than iterative(normal) functions. Even worse, if a recursive function is not implement properly(Every recursive call does not result to a smaller problem), the program will run forever.

One of the classic example is calculating finding the factorials.

\(n! = n * (n - 1) * (n - 2) * ..... * 2 * 1\)

We can implement it as:

```r
factorial_iter = function(n) {
  # trvial case for n is 0
  if (n == 0) {
    return(1)
  }
  result = 1
  for (i in 1:n) {
    result = result * i
  }
  return(result)
}
factorial_iter(5)
```

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

However, the recursive implementation is much simpler by simply stating its definitions again:

```
factorial_recursive = function(n) {
  # base case for n is 0 or 1
  if (n == 1 | n == 0) {
    return(1)
  }
  # we simple calculate n * by (n - 1)! by asking what is (n - 1)!
  return(n * factorial_recursiven - 1)
}
```

Another example is finding the fibonacci number:

The fibonacci number of the n-th term is defined as:

\(f(n) = f(n - 1) + f(n - 2)\)

Also, we define \(f(0) = 0, f(1) = 1\) We can implement the function normally with some calculations and thoughts:

```
fib_iter = function(n) {
  if(n == 0) { # avoid the case f(0) = 1 because of the for loop
    return(0)
  }
  prev = 0
  curr = 1
  next_ = 1 # avoid overwriting buildin methods
  for(i in 1:n) {
    prev = curr
    curr = next_
    next_ = curr + prev
  }
  return(prev)
}
for(i in 0:20) {
  print(fib_iter(i))
}
```

```
## [1] 0
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 8
## [1] 13
## [1] 21
## [1] 34
## [1] 55
## [1] 89
## [1] 144
## [1] 233
## [1] 377
## [1] 610
## [1] 987
## [1] 1597
## [1] 2584
## [1] 4181
## [1] 6765
```

Clearly, the function above is not very easy to understand and hard to implement. What if we simply use the definition of the fibonacci number to write the function?

```
fib_recursive = function(n) {
  if (n == 0 | n == 1) { # if n is 1 or 0, return it
    return(n)
  } else {
    # if n is not 0 or 1, simply ask: what is f(n - 1) and f(n -2)?, then calculate their sum
    return(fib_recursive(n - 1) + fib_recursive(n - 2))
  }
}
for(i in 0:20) {
  print(fib_iter(i)) # print all the results
}
```

```
## [1] 0
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 8
## [1] 13
## [1] 21
## [1] 34
## [1] 55
## [1] 89
## [1] 144
## [1] 233
## [1] 377
## [1] 610
## [1] 987
## [1] 1597
## [1] 2584
## [1] 4181
## [1] 6765
```

Obviously, the code above is much simpler and easier to observe. However, it has some serious performance issuses when a large n is entered such as 1000. The iteration will stop very soon but the recursive functions will never stop due to the tree-recursion issuse (1000 become 999 and 998 and so on, the run time will be something like \(a*2^n\))

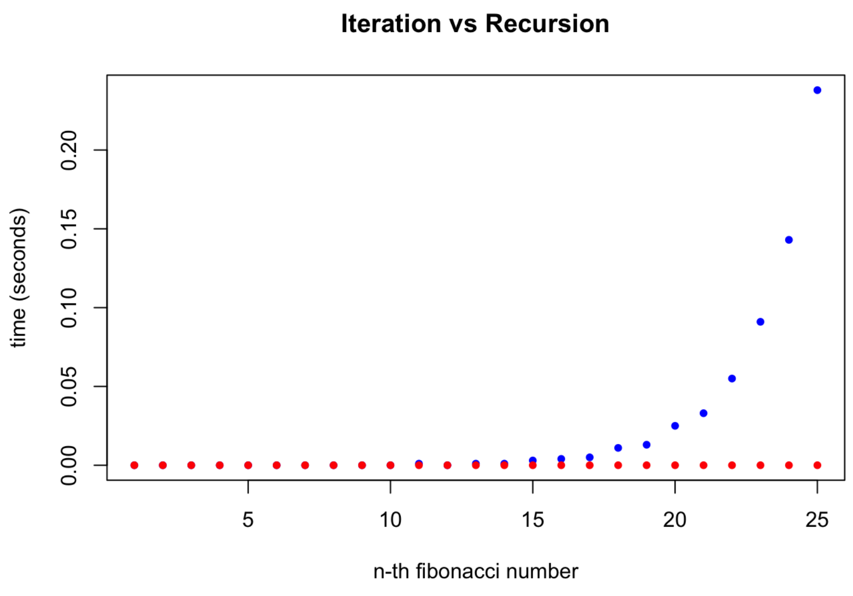
We can plot the graph to show the performance between the iterations and recursions:

```
recursive_time = c() # vector to store the time it takes to run the code
iteration_time = c() # vector to store the time it takes to run the code
for (i in 1:25) {

  time = proc.time() # record the time before running the code
  fib_recursive(i)
  recursive_time[i] = proc.time() - time # find the time it takes to run the code

  time = proc.time()
  fib_iter(i)
  iteration_time[i] = proc.time() - time
}

plot(recursive_time, xlab = "n-th fibonacci number", ylab = "time (seconds)", col = "blue", pch = 20, main = "Iter
ation vs Recursion")
points(iteration_time, col = "red", pch = 20)
```



We can clearly observe that the runtime of the recursion start increasing exponentially after n is larger than 17. Therefore, if someone is planning to use recursive functions to implement certain ideas, he/she has to be careful with the runtime issues.

Memoization

One of the reason that the recusive function is taking so much time is because it is repeatly computing certain values. The general idea of memoization is to let the program remember the result it compute before. If the same function call happens again, simple return the result instead of calculating it again. For example, when calculating f(9), f(9) will call f(8) and f(7). The problem is that the f(8) will also call f(7), and the problem is f(8) and f(9) are both calculating f(7). Therefore, in order to solve the runtime problem for the recursive function of fibonacci number, we can implement memoization in the function:

```r
# the vector that remember the result that is computed by the functions
# the index is argument(n), the values is the n-th fibonacci number
# initialize it as numeric mode to avoid the vector is actually NULL, capacity is 1000
memo = vector(mode = "numeric", length = 1000)
```

```r
fib_recursive_memoized = function(n) {
  # base case, similar to the previous implementation
  if (n == 0 | n == 1) {
    return(n)
  }

  # if n-th term is calulated, simply return
  if (memo[n] != 0) {
    return(memo[n])
  }

  # recursive call to find its value if n-th term is never calculated
  value = fib_recursive_memoized(n - 1) + fib_recursive_memoized(n - 2)

  # save the result <<- let the value assign to the global enviroment
  memo[n] <<- value

  return(value)
}

for (i in 1:50) {
  print(fib_recursive_memoized(i))
}
```

```
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 8
## [1] 13
## [1] 21
## [1] 34
## [1] 55
## [1] 89
## [1] 144
## [1] 233
## [1] 377
## [1] 610
## [1] 987
## [1] 1597
## [1] 2584
## [1] 4181
## [1] 6765
## [1] 10946
## [1] 17711
## [1] 28657
## [1] 46368
## [1] 75025
## [1] 121393
## [1] 196418
## [1] 317811
## [1] 514229
## [1] 832040
## [1] 1346269
## [1] 2178309
## [1] 3524578
## [1] 5702887
## [1] 9227465
## [1] 14930352
## [1] 24157817
## [1] 39088169
## [1] 63245986
## [1] 102334155
## [1] 165580141
## [1] 267914296
## [1] 433494437
## [1] 701408733
## [1] 1134903170
## [1] 1836311903
## [1] 2971215073
## [1] 4807526976
## [1] 7778742049
## [1] 12586269025
```

Without implementing the memoization, fib_recursive(50) is going to take almost infinite amount of time to compute. However, with memoization, fib_recursive_memoized(50) can compute the value as fast as the iterative solution.

Lets do another performance comparasion:

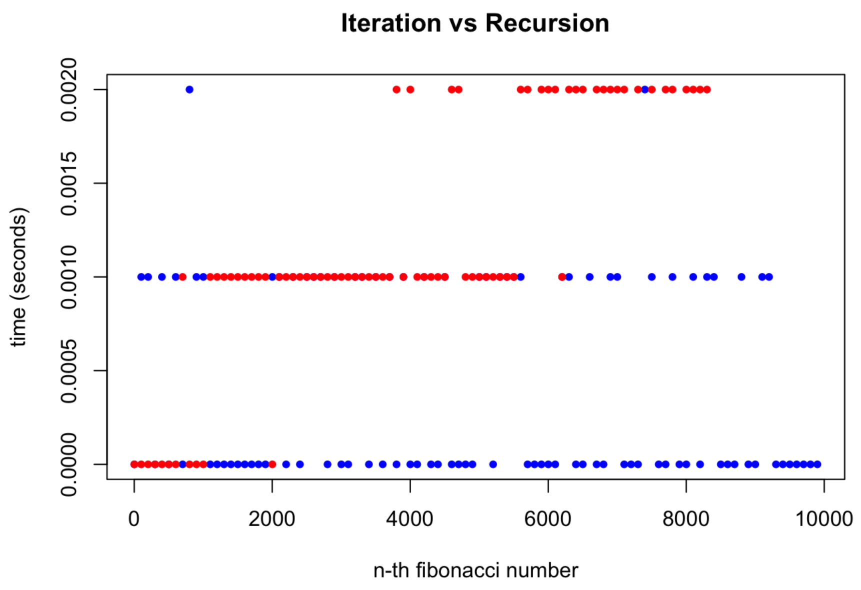
```
memo = vector(mode = "numeric", length = 10000)

recursive_time = c() # vector to store the time it takes to run the code
iteration_time = c() # vector to store the time it takes to run the code
for (i in seq(1, 10000, 100)) {

  time = proc.time() # record the time before running the code
  fib_recursive_memoized(i)
  recursive_time[i] = proc.time() - time # find the time it takes to run the code

  time = proc.time()
  fib_iter(i)
  iteration_time[i] = proc.time() - time
}

plot(recursive_time, xlab = "n-th fibonacci number", ylab = "time (seconds)", col = "blue", pch = 20, main = "Iter
ation vs Recursion")
points(iteration_time, col = "red", pch = 20)
```



The graph simply tells us that both of the iterative and the recursive functions take less than 0.002 seconds to compute the results and there is no distinctions in speed.

Misc

There are many things you can do with recursive functions. Beside this post, there are many other helpful implemented method that is online waiting for you to be explore. For example, one of the most common practice is sorting. For example, QuickSort and MergeSort:

Quick Sort

```
# Author:  Jason A. French
quickSort <- function(vect) {
  # Args:
  #  vect: Numeric Vector

  # Stop if vector has length of 1
  if (length(vect) <= 1) {
      return(vect)
  }
  # Pick an element from the vector
  element <- vect[1]
  partition <- vect[-1]
  # Reorder vector so that integers less than element
  # come before, and all integers greater come after.
  v1 <- partition[partition < element]
  v2 <- partition[partition >= element]
  # Recursively apply steps to smaller vectors.
  v1 <- quickSort(v1)
  v2 <- quickSort(v2)
  return(c(v1, element, v2))
}
```

Merge Sort

```r
# http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#R

mergesort <- function(m)

{
   merge_ <- function(left, right)
   # Recursive function to compare and append values in order
   {
      # Create a list to hold the results
      result <- c()
      # This is our stop condition. While left and right contain
      # a value, compare them
      while(length(left) > 0 && length(right) > 0)
      {
          # If left is less than or equal to right,
          # add it to the result list
         if(left[1] <= right[1])
         {
            result <- c(result, left[1])
            # Remove the value from the list
            left <- left[-1]
         } else
         {
            # When right is less than or equal to left,
            # add it to the result.
            result <- c(result, right[1])
            # Remove the appended integer from the list.
            right <- right[-1]
         }
      }
      # Keep appending the values to the result while left and right
      # exist.
      if(length(left) > 0) result <- c(result, left)
      if(length(right) > 0) result <- c(result, right)
      result
   }

   # Below is our stop condition for the mergesort function.
   # When the length of the vector is 1, just return the integer.
   len <- length(m)
   if(len <= 1) m else
   {
      # Otherwise keep dividing the vector into two halves.
      middle <- length(m) / 2
      # Add every integer from 1 to the middle to the left
      left <- m[1:floor(middle)]
      right <- m[floor(middle+1):len]
      # Recursively call mergesort() on the left and right halves.
      left <- mergesort(left)
      right <- mergesort(right)
      # Order and combine the results.
      if(left[length(left)] <= right[1])
      {
         c(left, right)
      } else
      {
         merge_(left, right)
      }
   }
}
```

Take-Home Message

Some people might think learning programming technics are not related to study statistics. However, without the help of the computer, it is impossible to analyze any large amount of data in an acceptable amount of time. Therefore, learning how to effectively writting the r code(or other codes that share similar properties) is very important. The technics I mentioned above might not be a superies to a student who is major in Computer Science. However, for any people that is learning R in the first time, I believe this will be a very helpful post to read.

By effectively understand how to use recursive functions, it will enable the audience the ability to actually practice and achieve their thoughts.

References

https://www.programiz.com/r-programming/recursion https://www.tutorialgateway.org/recursive-functions-in-r/ https://stats.idre.ucla.edu/r/faq/how-can-i-time-my-code/ https://www.r-bloggers.com/fibonacci-sequence-in-r-with-memoization/ http://adv-r.had.co.nz/Functions.html http://www.jason-french.com/blog/2014/07/26/recursion-in-r/ https://www.r-bloggers.com/programming-outside-the-box-a-recursive-function-in-r/