Blog Post01

Lucy Jingru Wang

October 20, 2017

Recursion vs Iteration in R

The purpose of using for loops is to eliminate repetitive code – so if we want to add 1 to the variable x 10 times, we can write a for loop that iterates 10 times instead of writing x+1 in 10 different lines. Recursion, is a even more simplified version of iterative approach to coding. After learning for loops in class, I grew curious about the recursive methods in R. In Python and Java, programmers sometimes use recursion for its elegant script. However, it definitely isn't the most efficient type of implementation since multiple frames are opened in the use of function calls but aren't closed until the base case is reached. (The concept of base cases and frames will be explained in the post) Does recursion work the same in R? We will explore the idea of recursion as well as various common examples of recursive implementation in this blog.

For loop vs. recursion 1:

For L00p:

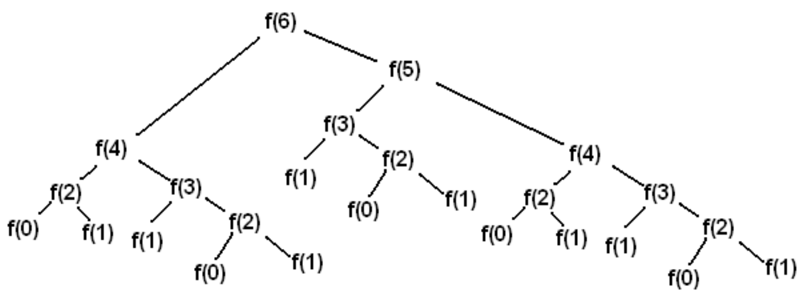
```r
fib <- function(n) {
    if(n==0){
    return (0)
  }
  else if(n < 3) {
    return (1)
    }

  else {
    prev = 1
    prev_n = 0
    for(x in 1:(n-1)){
      curr = prev+prev_n
      prev_n = prev
      prev = curr
    }
    return (curr)
  }
}
f <- c(0,1,2,3,4,10,100)
for(i in f){
 print(fib(i))
}
```

```
## [1] 0
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 55
## [1] 3.542248e+20
```

Compare the two functions: fib above and fib2 below. Each method does the job of calculating the nth number in the fib sequence, and yet the for loop method is much longer than the recursive below. In the first fib function, there are four declared variables and one passed in. $x_n = x_{n-1} + x_{n-2}$ 0,1,1,2,3,5,8,13… prev is the (n-1)th term prev_n is the (n-2)th term 'prev' stands for the n-1 term while 'prev_n' stands for the n-2 term. What is happening in the for loop (which could also be written with the repeat{}and while loops) is that for all n's above 1, each fibonacci value below the nth will be recalculated to obtain the final result. For instance, if our n were 4, then we would start with prev_n and prev both equal to 1, add them to get the fib(2) value, store the fib(1) as prev_n and fib(2) as prev, add fib(1)+fib(2) to obtain fib(3), store fib(2) as prev_ n and fib(3) as prev, and then add those two to obtain fib(4). The loop will exit since our iteration has finished repeatingly executing the code a total of four times.

The loop method, in my opinion at least, can be easily understood just by listing out several values. The recursive method, on the other hand, requires more thinking but less code. In the recursive method, our base case, or simplest case is when 0<=n<=2. If n is smaller than or equal to 2 and greater than 0, we know for sure that our fib value will be 1, and there isn't anything to add. In a simple world, people will only ask for fib(1) and fib(2). But since rarely are we given simple cases, our n value will most likely be greater than 1 and thus with the recusion, we would call itself. This fib function can be drawn as a tree. As you can see in the image below, each n will call the fib function 2 times. If we use the recursion method for fib2(6) say, then it will call f(5) and f(4), and each of those "nodes" will then call the function two times. As a result, this takes forever to execute since the function is called over and over verses in iteration, it only runs through a loop as the values are added up to obtain the final fib(n).

Interesting fact: it took about less than a second to run fib with iteration, yet it's still running the code 6 minutes later for the recursive method of fib(100).



FIB TREE

```
fib2 <- function(n) {
  if(n==0) {
    return (0)
  }
  else if(n<=2) {
    return (1)
  }
  else {
    return (fib2(n-1)+fib2(n-2))
  }
}
f <- c(0,1,2,3,4,10)

for(i in f){
 print(fib2(i))
}
```

```
## [1] 0
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 55
```

Fibonacci and the Golden Ratio

It turns out that the fibonacci sequence isn't just a random set of patterned numbers that the Italian mathematician came up with in the 13th century. Rather the numbers give a golden ratio that nature follows. After the first four numbers in the sequence, the ratio of a_n and a_{n+1} starts to approach 0.618 while alternative numbers approach a ratio of 0.382. Let's see this in action:

```
fib(5)/fib(6)
```

```
## [1] 0.625
```

```
fib(100)/fib(101)
```

```
## [1] 0.618034
```

```
fib(1000)/fib(1001)
```

```
## [1] 0.618034
```

```
fib(5)/fib(7)
```

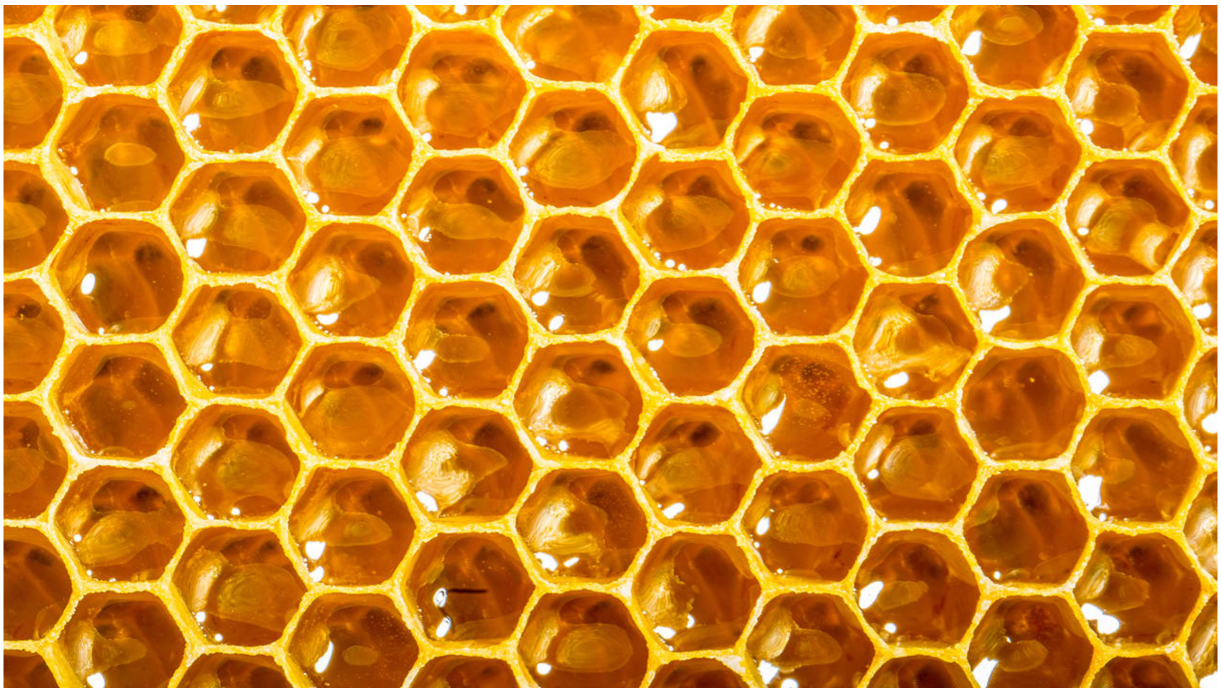
```
## [1] 0.3846154
```

```
fib(1200)/fib(1202)
```

```
## [1] 0.381966
```

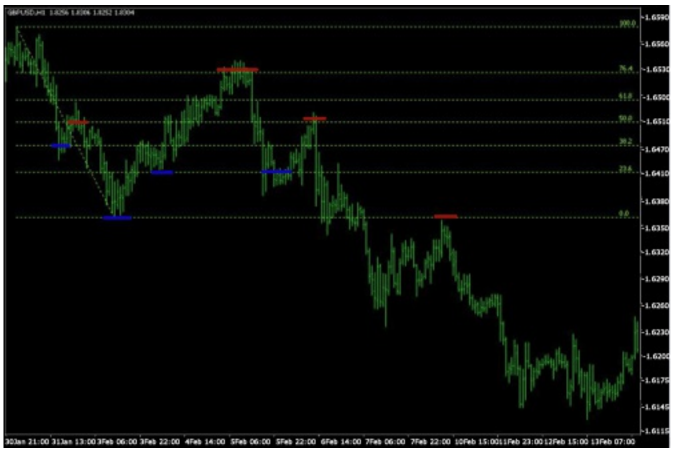
As seen from the outputs above, the ratio of the nth/(n+1)th term in the sequence approaches 0.618 while the alternative ratio approaches 0.382.

So.. Why does that matter?


BZZ

- The number of females divided by the number of males in any hive approaches the ratio of 1.618.
- The ratio between a person's height and distance from their belly button to feet also approaches that number.

But Fibonacci isn't only found in nature, it is also used in Finance and stocks… what? #Fibonacci Retracements Retracement generally refers to a change in the direction of a stock's price pattern.
On a retracement graph like the one below, lines are drawn at the 100,61.8,50, 38.2,and 0% to predict resistance and support lines. While resistance refers to the upper price level for stocks that is often reached but not exceeded, support lines mark the price level at which buyers tend to enter stock. Is this fibonacci model really effective? According to various business sources, it is a useful tool for prediction. However, exactly what makes this model so accurate is still in question and cannot be clearly explained. The Fibonacci ratio is often referred to as a "magic ratio" in mathematics and nature.


Retracement

Cosine Series RECURSION

$$cosx = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

The cosine series is another interesting iterative/recursive method to try out.

In the below implementation: - let x be the angle in radians that we want to predict cos(x) for. - let n be the number of values we want to subtract from 1 in the series

If we set the initial sum as 1, and subtract from it $\frac{x^{2t}}{(2t)!}$ for each t from 1 to n (input from user), we get an approximate value of cos(x) After testing out the cosine of π and $\pi/3$, you can see that the function accurately gives the cosine values.

L0OP

```
cosinef <- function(x,n) {
  sign = -1
  sum = 1
  for(t in 1:n) {
    sum = sum + sign*(x^(2*t))/(factorial(2*t))
    sign=-sign
  }
  return(sum)
}

cosinef(pi,11)
```

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

```
cos(pi)
```

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

```
cosinef(pi/3,11)
```

```
## [1] 0.5
```

```
cos(pi/3)
```

```
## [1] 0.5
```

```
cosinef(pi/4,15)
```

```
## [1] 0.7071068
```

```
cos(pi/4)
```

```
## [1] 0.7071068
```

Recursive

```
cosinef2 <- function(x,n,sum=1) {
  if(n==0) {
    return (sum)
  }
  else {
    return (cosinef2(x,n-1,sum+(-1)^n*(x^(2*n))/(factorial(2*n))))
  }
}
cosinef2(pi,11,1)
```

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

```
cosinef2(pi/3,11)
```

```
## [1] 0.5
```

```
cosinef2(pi/4,15)
```

```
## [1] 0.7071068
```

```
cos(pi/4)
```

```
## [1] 0.7071068
```

```
cos(3*pi/4)
```

```
## [1] -0.7071068
```

```
cosinef2(3*pi/4,10)
```

```
## [1] -0.7071068
```

How the recursive implementation works: This implementation is a bit different from the fibonacci one previously implemented in that it

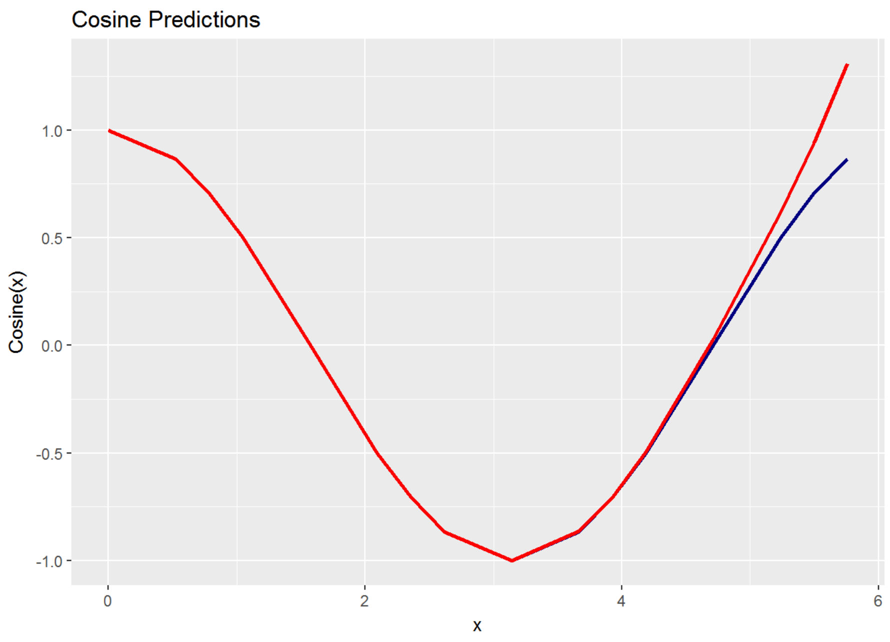
adds/subtracts values starting from the nth element.
$sum + (-1)^n * x^{2n}/(2n)!)$ x = value that we want to predict n = number of elements we want to use for prediction

How this function works: 1. Function will compute 1 + nth term and return cosinef2(x,n-1,1+nth) 2. In the second runthrough of the cosinef2 function through the recursive, it will treat (1+nth term) as the current sum, and then calculate a new sum = (1+ nth term + (n-1) term) 3. The new sum will then be run through the recusion where cosinef2(x,n-2, (1+ nth term + (n-1) term)) 4. This will continue on until the function finally reaches base case where n=0. That is when the sum that has been accumulating is finally returned.

Graphical visualization of results from our cosine function with n=6 compared to the RStudio Cosine function

The red line graphed is our predicted cosine value graph with 6 terms from the summation. As you can see, it differs very little from the actual value of cosine(x), and if we change the n to a larger number, then the graphs using cos(x) and cosinef2(x) will be nearly identical.

```
library(ggplot2)
X_values <- c(0,pi/6,pi/4,pi/3,pi/2,2*pi/3,3*pi/4,5*pi/6,pi,pi,7*pi/6,5*pi/4,4*pi/3,3*pi/2,5*pi/3,7*pi/4,11*pi/6)
Original <- cos(X_values)
Recursive <- cosinef2(X_values,6)
dat <- data.frame(X_values,Original,Recursive)

ggplot(data = dat, aes(x="X Value",y = "Cosine Value"))+
  geom_line(aes(x=dat$X_values, y = dat$Original),col = "navy",size = 1) +
  geom_line(aes(x=dat$X_values, y = dat$Recursive),col = "Red",size = 1) +
  ylab("Cosine(x)") + xlab("x")+
  ggtitle("Cosine Predictions")
```



Factorial Recursion

A Walk Through: Factorial Function in Recursion: ##### How do you come up with the recursive version of factorial function?

Recursion can get extremely complicated in logic, and thus it is important for us to master the general approach to writing a recursive method before trying to implement them. Below, we will do a step-by-step walk through for how to write the recursive method for a factorial function.

- The most crucial step to writing a recursion is to determine its base case
- Base case is the simplest case that exists for a particular function
- What is the simplest case of factorial? Is it 10!,9!,8!,7!….,1!,or 0!? Well, since $10! = 10 * 9 * 8 * 7 * \ldots$ and $9! = 9 * 8 * 7 * \ldots$ and so on, 0! must be the simplest because $0! = 1$ right? Right.

And thus now we can write the base case as an if statement at the top of our recursive function like this: (n is the number that we want to compute for factorial) if(n == 0) return 1 If the user asks for the factorial of 0, the function will just return 1, no computation necessary.

but if the n isn't 0, what happens? We would have to assume that our recursive method works and call on itself, meaning: we assume that the "factorial" function works and thus we would return n*factorial(n-1) for factorial(n) In mathematical expression, that is simply n! = n*(n-1)!, a fact we are sure of.

Thus the second part of our factorial function is just: else { return n*factorial(n-1) } Put together, our function will be: (So that I don't override the original factorial function in RStudio, I will title the function factorial1 instead)

```
factorial1 <- function(n) {
if(n == 0)
  return (1)
else {
  return (n*factorial1(n-1))
}
}
factorial1(0)
```

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

```
factorial1(1)
```

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

```
factorial1(2)
```

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

```
factorial1(6)
```

```
## [1] 720
```

```
factorial(6)
```

```
## [1] 720
```

Vector Recursion

Relating Java arrays to factors in R

Like the vectors in R, Java ArrayLists can store primitive types such as integer, booleans (equivalent to logical), characters, and strings (character equivalent). Below I will go over the JAVA and R equivalent methods: contains(Object o), indexOf(Object o), and get(int index).

JAVA contains:

contains(Object o): searches within the arraylist to see if the Object o exists in the list; contains() returns a boolean (logical) TRUE or FALSE. Object o can be any type of object such as Integer or String. If the object is found in the ArrayList, then it will return TRUE, otherwise FALSE

R EQUIVALENT: %in%

search whether or not an element is inside a vector. Example: since the character 'b' is in the vector, the method returns true.

```
v <- c('a','b','c','e')
'b' %in% v
```

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

JAVA indexOf:

indexOf(Object o): This function will return the index of the object that is passed through the method. The difference between Java indexes and R indexes,however, is that Java index starts from 0 while R vector indexes start from 1

R EQUIVALENT match()

Like indexOf, match() returns the index of the element that is passed through the function within the vector. In this case, because the character 'b' is the second element in the vector, 2 is returned.

```
match('b',v)
```

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

JAVA get():

get(int index): This function will return the object that is stored in the arraylist at the integer value index.

R EQUIVALENT []:

For Java's ArrayList, the function get() must be used in order to retrieve an item from the list. But for vectors in R, all we need to do is use the brackets with the index number inside and the value at that position will be returned!

```
v[1]
```

```
## [1] "a"
```

Because recursive functions take up a lot of memory and are inefficient generally, it is actually not a good idea to implement a method like contains recursively. But just for the sake of recursion fun and testing out recursion in R, I will construct a recursive contains for vectors.

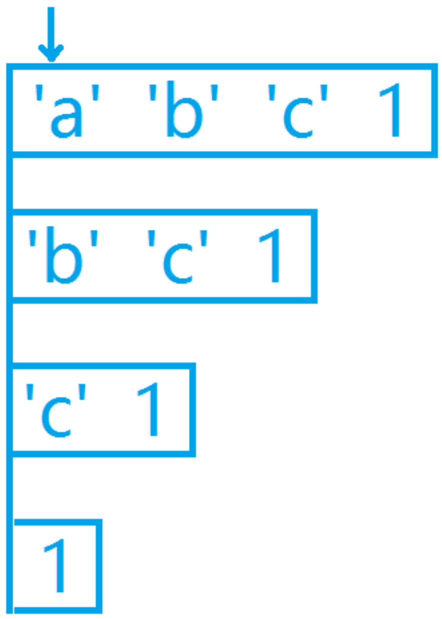
```
contains <- function(vectorv,item) {
  if(vectorv[1]==item) {
    TRUE
  }
  else if(length(vectorv)==1) {
    FALSE
  }
  else {
    contains(vectorv[2:length(vectorv)],item)
  }
}
```

How this recursive method works:

In my implementation of this method, like the factorial method, I again thought of the simplest case possible, and that is if the first element of the vector being passed in is the item that we were searching for. If the items are same, then we return TRUE because we know the item is in this vector. However, if the item isn't the first one in the vector and the length of our vector is 1, then that means the item just isn't in the list and thus we would return FALSE. In the case where the list has length more than 1, I can use $recursion:)!!$ Assuming that our recursive method works, I will pass through vectorv[2:length(vectorv)] as the new vector that we are searching for the "item" in. This makes sense because since we already know that the first index of the original vector is not the "item" we were looking for, we can just ignore it and then search through the rest of the list starting from index 2 until the last element in the vector at index "length(vectorv). Here is an visualization of what happens for the below snippet of code in which contains is called.

```
v <- c('a','b','c',1)
contains(v,5)
```

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



Frames of each call

In the first call, our vector looks like the first rectangular box, but once the contains function has verified that the first element isn't the same as what we are looking for (5) and that the length isn't 1, it calls itself recursively with the remaining elements in the vector until it reaches the last vector with just the element 1. In this last step, contains compares to see if 1 == 5, which is not true, and then it verifies that the length of the final vector is 1, which makes it return FALSE.

Conclusion:

Thank you for reading and hopefully you enjoyed learning about recursion through this blog!

References:

https://www.stocktrader.com/2009/05/26/fibonacci-numbers-investors-sequence-elliot-wave-theory/ http://www.investopedia.com/articles/technical/04/033104.asp?lgl=rira-baseline http://www.investopedia.com/terms/r/resistance.asp https://www.programiz.com/r-programming/recursion https://introcs.cs.princeton.edu/java/23recursion/ ww.tutorialgateway.org/recursive-functions-in-r/ http://r4ds.had.co.nz/iteration.html

Processing math: 100%