Lecture 5: Functions, Recursion, Benchmarking, and Memoization

STAT 385 - James Balamuta

June 21, 2016

On the Agenda

- Functions
  - How to write a function
  - When should we write them?
- Recursion
  - Defining something in terms of itself
  - Factorial Calculation ($n!$)
- Benchmarking
  - How fast are you going?
- Memoization
  - Caching function calls

Talking about a Function

Previously, we just wrote statements like:

```r
cat("Hello World!")
```

within a main document.
If we wanted to repeat that phrase elsewhere in the program, we would have to either:

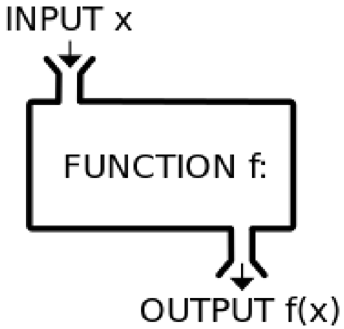
1. Retype
2. Copy and paste

to the new location.

- This is **not** an ideal situation.

What is a Function?

Definition: A piece of code that performs a specified task that may or may not depend on parameters and it may or may not return one or more values.



Why use a function?

Functions are great because:

1. The logic flow is chunked instead of a series of long statements
   - Also enables self-documenting depending on the function name.
2. Decrease the probability of an error
   - Update the code in one central place instead of tracking multiple locations
     - No need copy code from one section and paste it in another
   - Easily reuse code across multiple analysis
   - No need to understand the internal computation.
3. Easier to share code with people that you collaborate.
   - Knowledge of what the function **does**
   - Parameters are established
   - Return information is known

When should I use a function?

- The moment that you copy and paste more than a line of code within your analysis
    - This code should be within a function.
- If there is a possibility that you will need to reuse or call the same block of code at any point in time, it should be made into a function.

Characteristics of a Good Function

When writing functions, its important not to fall into various traps. Thus, you should aim for having functions that have:

1. Intuitive Naming Scheme
   - Function name and parameters are clear as to their origins.
2. Solving a **single** problem.
   - Try to outsource logic so that each step of the problem can be analyzed.
3. Concise
   - Taking many statements and wrapping it into a function should be *avoided*.

Function Declaration and Calling

To declare the most simplistic function in R, you must use:

```r
function_name = function ()  {
  # body
}
```

To call a function, use:

```r
function_name()
```

Function Declaration and Call Example

Here we opt to create a function that will by default say “Hello World!”

```
hello_world = function() { # Function Declaration
  cat("Hello World!\n")    # Body Statement
}

hello_world()              # Call Function
```

```
## Hello World!
```

Notes:

- The function is unable to receive user input.
- Each time the function is called the same result will appear.
- Thus, the function is **static**.

Dynamic Function Declaration

Often times, we will need something more flexible to the ever changing needs of the world. For a function to be flexible, we must add **arguments** to the function.
There are two kinds of arguments:

- **Positional:** Order matters in the entry
- **Default Parameter :** Not necessary to specify a value for the parameter.

```
func_dynamic = function (parameter1, parameter2 = NULL)  {
  # body
}
```

In this case, `parameter1` is *positional* and `parameter2` is *default parameter*.

Dynamic Function Declaration Call

We can then call the function with either:

```
# Specify both values
func_dynamic(parameter1, paramter2)

# Parameter2 will use NULL as the value
func_dynamic(parameter1)
```

Dynamic Function Declaration and Call - Formatting Output

Case Example: Results need to look presentable and consistent throughout a document.

```r
# Function Declaration
format_percent = function(x, digits = 3){   # Define Func
  percent = round(x * 100, digits = digits) # Round digits
  result = paste0(percent, "%")             # Add % sign

  return(result)                            # Return
}
```

Dynamic Function Declaration and Call - Formatting Output

To call the function, we can simply do:

```r
x = runif(5)

format_percent(x)
```

```
## [1] "44.596%" "43.618%" "91.896%" "74.335%" "5.625%"
```

```r
format_percent(x, digits = 0)
```

```
## [1] "45%" "44%" "92%" "74%" "6%"
```

The Mysterious of `return()`

One of the nice aspects of *R*, is you can avoid returning an object via `return()` and simply just leave the name of the last variable. e.g.

```
format_percent = function(x, digits = 4){
 percent = round(x * 100, digits = digits)
 result = paste0(percent, "%")

 result  # Last Result
}
```

This will save you *micro*seconds but increase your heartache.

The Return Statement

```r
example_return = function(value = TRUE) {
 if(value) {
   return(TRUE)  # Clear
 } else {
   return(FALSE) # Clear
 }
}
```

```r
example_return = function(value = TRUE) {
 output = NULL
 if(value) {
   output = TRUE
 } else {
   output = FALSE
 }
 output                # Not Clear
}
```

The Return Statement

As was shown, *R* is very lax when it comes to `return()`. By default, *R* will return the last operation performed.
Whether you choose to use `return()` or not when designing a function is ultimately a personal preference.
Remarks:

- Harder to read without the `return` statement
- Requires you **not** to add any statements after `output`
- Might not be ideal if the function is a one liner.

Dynamic Function Declaration and Call - Formatting Output

Later, you may wish to add additional features to the function.
To do so, it is recommended that:

1. You use **default parameter** containing the unmodified value.
2. Make sure the function returns similar values.

```r
format_percent = function(x, digits = 4, psign = TRUE){
  percent = round(x * 100, digits = digits)

  if(psign){  # Check to see if % should be added
    result = paste0(percent, "%")
  } else {
    # Coerce to character to match % output.
    result = as.character(percent)
  }
  result
}
```

Dynamic Function Declaration and Call - Linear Regression

Often, when working with linear regression, we want to compute three different summations to compute other useful diagnostic information:

- The Total Sum of Squares (TSS)

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y}_i)^2$$

- Fitted Sum of Squares (FSS)

$$FSS = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

- Residual Sum of Squares (RSS)

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y})^2$$

Dynamic Function Declaration Example

Converting these statements to a function yields:

```
compute_tss = function (y, y_bar)  {
  return(sum( (y - y_bar)^2 ))
}

compute_fss = function (y_hat, y_bar)  {
  return(sum( (y_hat - y_bar)^2 ))
}

compute_rss = function (y, y_hat)  {
  return(sum( (y - y_hat)^2 ))
}
```

Dynamic Function Declaration and Call - Linear Regression

These summations have the following relationship:

$$TSS = RSS + FSS$$

$$\sum_{i=1}^{n} (y_i - \bar{y})^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

Dynamic Function Declaration - Linear Regression

Exploiting the previously made functions with the aforementioned relationship, we get:

```r
tss_relationship = function (y, y_hat, y_bar)  {
  rss = compute_rss(y, y_hat)
  fss = compute_fss(y_hat, y_bar)
  tss = rss + fss
  return( tss )
}
```

Dynamic Function Call - Linear Regression

Setup Function Calls:

```r
# Number of Observations
n = 10
# Generate x
x = seq(0, 1, length.out = n)

# Generate random y
# Set seed for reproducibility
set.seed(114)
y = runif(n)

# Calculate mean
y_bar = mean(y)

# Obtain y_hat
y_hat = lm(y~x)$fitted.values
```

Dynamic Function Call - Linear Regression

Call the functions:

```
# Compute
rss = compute_rss(y, y_bar)
fss = compute_fss(y_hat, y_bar)
tss = compute_tss(y, y_bar)
tss_v2 = tss_relationship(y, y_hat, y_bar)

# Verify equality
all.equal(tss, tss_v2)
```

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

Variable Scope in a Function

Within a function's {}, *R* uses the defined variables within that region.

```r
# Note `value` has not been defined.
multiple_constant = function(x) {
  return(value * x)
}

# Only on call is an error detected.
multiple_constant(5)
## Error in multiple_constant(5) :
## object 'value' not found
```

Note:

- *R* always gives you the benefit of the doubt and this sometimes gives bad results. . .

Function vs. Global Variable Scope

If a variable is not found, it will search the global environment.

```
# Define value in global environment
# (e.g. outside of the function)
value = 3

multiple_constant = function(x) {
  # `value` is not been defined in the function.
  return(value * x)
}

multiple_constant(5)
```

```
## [1] 15
```

Note:

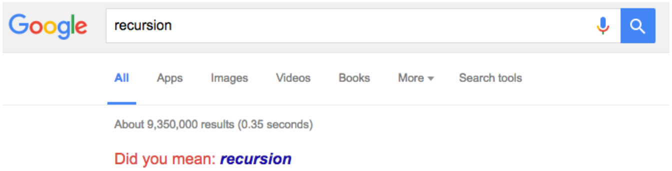
- This behavior can be very problematic. Try to keep your scope limited.

Summary of Functions

- Talked about the steps to creating a function.
- Explored ideal function designs.
- Acknowledged R's environment scoping habits.

What is Recursion?

Definition: Defining a process that is in terms of itself such that the problem is able to be *simplified* and *delegated* in a reduced state.



About 9,350,000 results (0.35 seconds)

Did you mean: ***recursion***

- Search Google for Recursion and try to click on the "Did you mean: *recursion*" part.

What is Recursion? (Loosely Stated)

Loosely, recursion can be considered in the following light:

- If a problem is easy to solve enough, then simply solve it.
- Else, reduce the problem so that there is *one or more simpler versions* of the **same problem** and try to solve them.

Example of Recursion

Consider the objective of going to **Illini Hall**.
Let's define a function called "Travel to Illini Hall" and implement it recursively.

1. If you are in Illini Hall, stop.
2. Ask a random person where Illini Hall is and move in the direction suggested until you are unsure.
3. Travel to Illini Hall

Properties of a Recursive Function

Inorder for a function to be recursive, it must have the following:

1. **Base Case**
   - Condition to stop
   - Also, the **simpliest** solution to the problem.
2. **Work** toward **Base Case**
   - Make the problem simplier
3. **Recursive Call**
   - Call to itself

Power function

Another example of recursion would be that of the **power function** that acts to simplify repetitve multiplication of one object.

$$\begin{aligned} x^k &= x \cdot x^{k-1} \\ &= x \cdot x \cdots x \\ &= \prod_{i=1}^{k} x \end{aligned}$$

Power Function with Loops

This behavior can easily be modeled using a loop:

```
pow_loop = function(x, n){ # Define function

  result = 1               # Setup output variable

  for(i in seq_len(n)){    # Loop over the power
    result = result*x
  }

  result                   # Return the result
}
```

Note: This implementation does *not* support negative exponents!

Power Function with Recursion

However, we're more interested in recursively defining the function. That is, we want to define the power function in terms of *itself*.

```
pow = function(x, n){ # Define function

    if (n <= 0){       # Set a base case
      return(1)        # No recursive call
    }

    # Solve a small part of the larger problem
    return(x * pow(x, n - 1)) # Call ourself
}
```

Note: This implementation does *not* support negative exponents!

Behind the Scenes of the Power Function in R

What we are saying in this particular case is that:

- `pow(x, n) = x*pow(x, n - 1)`

So that, if `pow(2,5)`, we have:

- `pow(2,5)` is equal to `x * pow(2,4)`
- `pow(2,4)` is equal to `x * pow(2,3)`
- `pow(2,3)` is equal to `x * pow(2,2)`
- `pow(2,2)` is equal to `x * pow(2,1)`
- `pow(2,1)` is equal to `x * pow(2,0)`
- `pow(2,0)` is equal to 1

That is, we have broken down each of the components to obtain the power and have end up with the solution. The key is the last stage of the problem, where we invoke the base case to stop the recursion.

Viewing Recursion from a Functional Perspective

```r
pow = function(x, n, d){ # Define function
    # Add a tracer into the function
    spacer = "";
    for (i in seq_len(d)) {
        spacer = paste0(spacer, " ", collapse = "")
    }
    cat(spacer, "pow(",x,",",n,")\n", sep="")

    # --------- Same code as before

    if (n <= 0){        # Set a base case
      return(1)         # No recursive call
    }

    # Solve a small part of the larger problem
    return(x * pow(x, n - 1, d + 4)) # Call ourself
}
```

Viewing Recursion from a Functional Perspective - Output

```r
cat(pow(7, 2, 0))
```

```
## pow(7,2)
##     pow(7,1)
##         pow(7,0)
## 49
```

Factorials

A more useful recursive function is that of a **factorial**.
Factorials are given by:

$$\begin{aligned} n! &= n \cdot (n-1)! \\ &= n\,(n-1)\,(n-2)\cdots 3 \cdot 2 \cdot 1 \\ &= \prod_{i=1}^{n} i \end{aligned}$$

Note: Consider viewing Proofs of the Gamma distribution to understand the recursive nature of the Gamma function for integers $n! = \Gamma\left(n+1\right)$

Use cases

Factorials are commonly used in Combinatorics for:

- **Permutations:** Order of selection matters without replication

$$P\left(n, k\right) = \frac{n!}{(n-k)!}$$

- **Combinations:** Order of selection does not matter without replication

$$\begin{aligned} C\left(n, k\right) &= \binom{n}{k} \\ &= \frac{P\left(n, k\right)}{P\left(k, k\right)} \\ &= \frac{n!}{k!\left(n-k\right)!} \end{aligned}$$

Factorial

Here is a custom implementation of the factorial function:

```r
factorial_r <- function(x){    # Function Definition
  if(x <= 1){
    return(1)
  } else {
    return(x*factorial_r(x-1))
  }
}
```

- Identify the 3 parts of the recursive algorithm

Note: Factorials in *R* are given by the `factorial(x)` function and not `x!` (Why?)

Misc note on Recursion

- Examples provided deal with **tail** recursion.
  - That is, the last statement of the function is calling itself.
- Only **tail** recursion can be converted to using **loops**.
- **Avoid**, like the plauge, using **global variables** with recursion.

Summary on Recursion

- Recursive functions call themselves
- Recursion exists in everyday statistics

Benchmarking

Coming up next... **Benchmarking** in R!

Code Implementations

- There are many ways to implement an idea..
    - The question is:
    - What implementation is the best?

Benchmark

To answer what implementation works the best, one needs to employ a benchmark to obtain **quantifiable results**. Benchmarks are an ideal way to quantify how well a method performs because it has the ability to:

1. Show the amount of time the code has been running
2. Bottlenecks
3. Sloppy coding

Plus, it appeals to our quest for data driven decisions.

A Note on Benchmarks..

Benchmarks are like a microscope.

Magnification is able to be adjusted... But what are you really observing?
The truth? Or just a clever lie?

Writing a Benchmark

Benchmarks can be very tempting to use to solely make decisions. However, in order for the benchmark to be beneficial, the benchmark must be done properly.

Properly Writing a Benchmark

To ensure the code is properly time benchmarked, do the following:

1. Only have R open
2. Do not use the computer while the benchmark is active.
3. Warm up code prior to benchmarking (code will be “hot” during benchmark)
4. Have enough replications ($N = 100$)
5. Fix the data used per replication (e.g. same seed)
6. Check code provides same output

With this being said, let’s create one (or two) to see how it works!

The Simpliest Benchmark

The simpliest benchmark is given with R's built in `system.time()` function.
The key time is the "elapsed" time as it is the summation of the others.

```
out = system.time({Sys.sleep(1)})
out
```

```
##    user  system elapsed
##   0.001   0.000   1.005
```

```
out[3]
```

```
## elapsed
##   1.005
```

Note: Adding, removing, or modifying a variable will impact the timing!

Setting the Stage

In this case, we are going to be interested in two ideas:

1. The cost of operating on a `matrix` vs. a `dataframe`.
2. The cost of using () vs {}. (Credit to Dirk Eddelbuettel)

Setting the Stage

For the first benchmark case, we are going to create the following objects:

```r
# Set seed for reproducibility
set.seed(1337)

# Construct large matrix object
matrix.op = matrix(rnorm(10000*100), 10000, 100)

# Convert matrix object to data.frame
dataframe.op = as.data.frame(matrix.op)
```

Cost of using Parentheses vs. Curly Brackets

```r
# Different R implementations
f = function(n, x=1) for (i in 1:n) x=1/(1+x)
g = function(n, x=1) for (i in 1:n) x=(1/(1+x))
h = function(n, x=1) for (i in 1:n) x=(1+x)^(-1)
j = function(n, x=1) for (i in 1:n) x={1/{1+x} }
k = function(n, x=1) for (i in 1:n) x=1/{1+x}
```

Cost of using Parentheses vs. Curly Brackets

To illustrate the benefits of compiler vs. interpreter, we're also going to call upon C++ using Rcpp.
Do not worry about understanding the code presented next...

```r
# Load Rcpp
library(Rcpp)

# Define a version in C++ called
cppFunction(code='int d(int n, double x = 1.0){
                  for (int i=0; i<n; i++) x=1/(1+x);
                  return x;
                  }')
```

R Package: rbenchmark

The `rbenchmark` R package provides a way to obtain the total amount of time elapsed for a given piece of code run over N replications.
The benchmark is initiated by:

```r
# Load Library
library('rbenchmark')

# Run benchmark
benchmark(testfun1 = somefun(),
          testfun2 = otherfun())
```

R Package: rbenchmark

The results returned are:

- The user, system, and total elapsed times for the active R process
- The cumulative sum of user and system times of any child processes spawned by it on which it has waited. (*.child,* .self)

R Package: rbenchmark - Case 1

```
# Load Library
library('rbenchmark')

# Run benchmark
out = benchmark(mat.op = apply(matrix.op, 2, sd),
                df.op = apply(dataframe.op, 2, sd))
```

```
# Table Object
out
```

test	replications	elapsed	relative	user.self	sys.self	user.child
df.op	100	5.20	2.07	4.95	0.18	0
mat.op	100	2.52	1.00	2.38	0.10	0

```
# Average time spent per iteration
out[,"elapsed"]/out[,"replications"]
```

R Package: rbenchmark - Case 2

```
N = 1e6 # Number of Times to Run Loop
out = benchmark(f(N, 1), g(N, 1), h(N, 1), j(N, 1), k(N, 1
          columns = c("test", "replications", "elapsed", "r
          order="relative",  # Order the results by speed
          replications=20)   # Number of runs for each func
```

	test	replications	elapsed	relative
6	d(N, 1)	20	0.184	1.000
5	k(N, 1)	20	5.705	31.005
1	f(N, 1)	20	5.925	32.201
4	j(N, 1)	20	6.621	35.984
2	g(N, 1)	20	7.383	40.125
3	h(N, 1)	20	9.964	54.152

R Package: `rbenchmark` - Case 2

The average amount of time a replication takes can be obtained via:

```
out[,"elapsed"] / out[,"replications"]
```

```
## [1] 0.00920 0.28525 0.29625 0.33105 0.36915 0.49820
```

Drilling down with `microbenchmark`

- The `microbenchmark` R package provides a way to record time for each iteration unsummed!
  - This provides more magnification on functions than `rbenchmark`.
- To use `microbenchmark` do the following:

```r
# Load Library
library('microbenchmark')

# Run benchmark
microbenchmark(testfun1 = somefun(), # List Functions
               testfun2 = otherfun(),
               times = 100) # Number of Replications
```

Note: This is not ideal for large functions. This is meant for pinpoint precision of elements.

Drilling down with `microbenchmark` - Case 1

```r
# Load Library
library('microbenchmark')

# Run benchmark
out = microbenchmark(mat.op = apply(matrix.op, 2, sd),
                     df.op = apply(dataframe.op, 2, sd))
```

```r
# Table Object
summary(out)
```

expr	min	lq	mean	median	uq	max	neval
mat.op	14.70	16.82	27.55	18.85	49.47	62.09	100
df.op	19.75	24.85	49.10	57.12	60.31	99.49	100

Drilling down with microbenchmark - Case 2

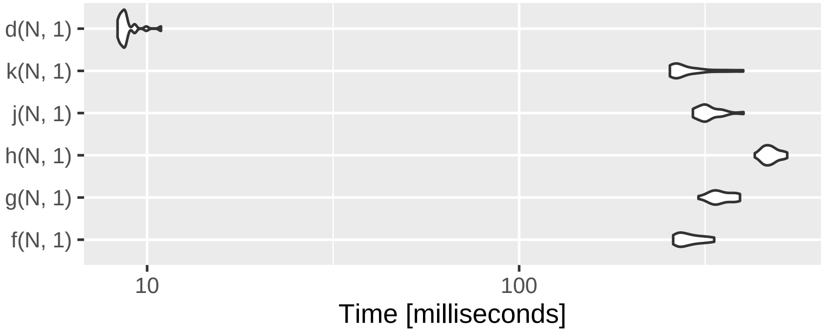
```
N = 1e6 # Number of Times to Run Loop
out = microbenchmark(f(N, 1), g(N, 1), h(N, 1), # Funcs
                     j(N, 1), k(N, 1), d(N, 1),
                     times = 20)    # Number of Reps
```

```
summary(out) # Obtain quantile information
```

expr	min	lq	mean	median	uq	max	neval
f(N, 1)	259.45	268.77	288.23	283.40	307.21	334.50	20
g(N, 1)	303.35	330.65	350.13	344.25	373.65	392.40	20
h(N, 1)	429.96	454.65	474.01	470.27	487.91	525.58	20
j(N, 1)	293.16	307.99	324.64	317.21	338.54	401.10	20
k(N, 1)	254.24	263.04	286.54	267.90	299.24	400.44	20
d(N, 1)	8.33	8.50	8.84	8.69	8.81	10.90	20

Visualization using `microbenchmark` and `ggplot2`

```r
library(ggplot2)
autoplot(out)   # Creates a violin plot of the data
```



Summary of Benchmarking

- Covered usage reasons for benchmarking code
- Discussed areas of concern for obtaining benchmarks.

Memoization

Coming up next... **Memoization** in R!

Caching Values is Memoization

One of the common themes of this course is caching or storing values to be reused later during a computation.

- This is exactly the definition of **memoization**.
- Memoization is key to a lot of intense statistical computations. A recent entry on this can be found as a response to a question on StackOverflow.

Memoization in Depth

The goal of memoization is to cache function results under given parameters and reuse them at a later time.

- To cache the value of the computation, a key is created that relates the function and the parameters used within it.
- Thus, after one calculation under the given parameters the function no longer needs to recalculate the same result again if called for any other sequence in the computation.
- The reason why is the result can just retrieve the value from the cache.

Note This is particularly relevant for the recursive structure set up.

Memoization in R

- Hadley Wickham, Jim Hester, and Kirill Müller have created phenomenal package called `memoise`.
- The package brings in the ability to memoization functions with very little work.
  - You just need to be wary of the cache!

Fibonacci Sequence

The Fibonacci Sequence is a famous series that traditionally looks like so:

$$1, 1, 2, 3, 5, 8, 13, 21$$

The mathematical formulation is known as:

$$F(n) = F(n-1) + F(n-2)$$

where $F(1) = 1$ and $F(2) = 1$.

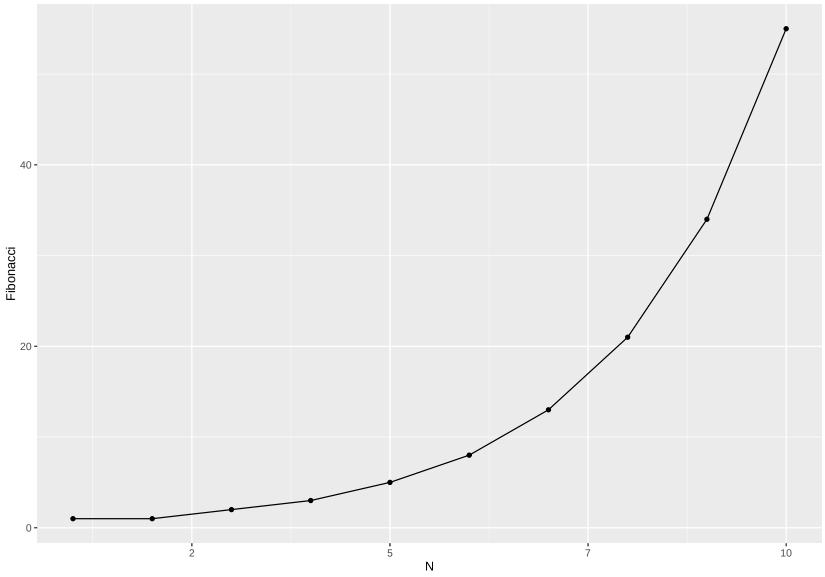
Example Fibonacci Sequence

We can write the Fibonacci Sequence like so:

```
fibonacci = function(n) {
   if (n <= 2) {
     return(1)
   }

   return(fibonacci(n-2) + fibonacci(n-1))
}
```

Fibonacci Sequence in an Image



Example Memoization

To memoize the Fibonacci function, we need to wrap place the function name in `memoise()`

```r
library("memoise")
mem_fib = memoise(fibonacci)
```

We then refer to the memoized function just like normal previously, e.g.

```r
fibonacci(5) # Normal
```

```
## [1] 8
```

```r
mem_fib(5)   # Memoized
```

```
## [1] 8
```

A Word on the Memoization Cache

In this case, the memoization will only apply to the top level function call.
So, the results obtained via `mem_fib(34)` will not effect `mem_fib(35)`.

```
system.time({mem_fib(34)}) # First time calculation
```

```
##    user  system elapsed
##   7.809   0.030   7.930
```

```
system.time({mem_fib(35)}) # First time calculation
```

```
##    user  system elapsed
##  12.753   0.037  12.994
```

Internal Memoization Cache

To cache the in-between levels, you will need to declare the memoization within the function. e.g.

```
internal_mem_fib = memoise(function(n) {
  if (n <= 2){
    return(1)
  }

  # Note the reference to the outside declaration
  return(internal_mem_fib(n - 2) + internal_mem_fib(n - 1))
})
```

Benchmarks

When we write it this way, we gain speed due to the cache given by memoization being used. e.g.

```r
system.time({mem_fib(25)}) # Registers value in map
```

```
##    user  system elapsed
##   0.107   0.002   0.110
```

```r
system.time({mem_fib(25)}) # Extracts value from map
```

```
##    user  system elapsed
##   0.000   0.000   0.001
```

```r
system.time({internal_mem_fib(25)}) # Built in Memoization
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

Clear the Cache

To remove memoization, use `forget()`

```r
forget(mem_fib)
```

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

```r
forget(internal_mem_fib)
```

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

Note: You will have to redeclare the memoization again with `memoise()`

Summary of memoization

- Memoization is used as a short term memory bridge that speeds up the calculation.
- Be wary of the cache! Are you using an internal or external cache?