This article describes how you can apply a programming technique, called Memoization, to speed up your R code and solve performance bottlenecks. Wikipedia says:

*In computing, […] memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.*

Source: <https://en.wikipedia.org/wiki/Memoization>

If you want to fast forward and only apply the technique, you find two R packages on CRAN implementing it: [R.cache](https://cran.r-project.org/web/packages/R.cache/index.html) and [memoise](https://cran.r-project.org/web/packages/memoise/index.html). In the following I want to implement different versions to (1) demonstrate that it is not magic and a fairly straight forward technique (2) to show that R is faster than C++ after all!

**Performance optimizations in R**

If I read about high performance computing, R seems to have a bad reputation when it comes to raw computing speed. Thanks to Rcpp the integration of C++ into your R project is as simple as it can be. Not necessarily easy though: you still have to learn some C++. Oftentimes I feel that discussions around this topic neglect that implementation would cost anything at all, and so recommendations too quickly imply that *you have to use a different language*. However, very often we can use [plain old R](https://arxiv.org/abs/1409.5827) and squeeze run time performance combined with ease of implementation out of it. How? Very often by redefining the problem (cheat), splitting computations into pieces (divide-and-conquer) or find the bottleneck and optimize the heck out of it (make the most out of the language)!

Of course any of those steps may force us to finally use C++ or something else; but we can push this point a bit further away and compensate it with some brain power and a toolbox of optimization strategies. There is a lot to learn when it comes to optimization. Memoization can be one technique in your bag of tricks to magically make R code run faster. And this is done by avoiding unnecessary computations; or unnecessarily compute things twice.

**When R is slow**

To begin this exercise, let’s see when R is slow and later how we can improve it. I took this example from the [Rcpp documentation](http://dirk.eddelbuettel.com/code/rcpp/html/Misc_2fibonacci_8r-example.html) which apparently has been created in response to [a question on StackOverflow](https://stackoverflow.com/questions/6807068/why-is-my-recursive-function-so-slow-in-r) asking why R’s recursive function calls are so slow.

The challenge is to compute the fibonacci numbers using one of the most computationally inefficient definitions, efficiently. Of course the concrete example is not really the point. There are other, more efficient ways, to compute the fibonacci sequence. But the recursive definition makes your CPU fan spin up and that’s why it is interesting. In the following you can find the ‘naive’ implementation of the algorithm. Both languages really look more or less the same. But, as you can see, the timings are already different. We can change N just a bit and the implementation in R will reach it’s limits, fast.

N <- 35 ## the position in the fibonacci sequence to compute ## The C++ version fibRcpp <- Rcpp::cppFunction(' int fibonacci(const int x) { if (x == 0) return(0); if (x == 1) return(1); return (fibonacci(x - 1)) + fibonacci(x - 2); }') fibR <- function(x) { ## Non-optimised R if (x == 0) return(0) if (x == 1) return(1) Recall(x - 1) + Recall(x - 2) } rbenchmark::benchmark( baseR = fibR(N), Rcpp = fibRcpp(N), columns = c("test", "replications", "elapsed", "user.self"), order = "elapsed", replications = 1 )

## test replications elapsed user.self ## 2 Rcpp 1 0.07 0.06 ## 1 baseR 1 29.27 27.69

**When R is fast(er)**

Now let’s see how we can define a function, still computing the same numbers, avoiding all the unnecessary computations. Note that in the recursive definition, to compute position N we have to compute position N – 1 and N – 2. This will lead to an explosion in the number of computations. At the same time, if we want to compute the fibonacci number for N = 35 and we already have the result for N = 34 and N = 33 we do not have to re-compute them, we just use what we already know instead. Let’s see how this can be done:

fibRMemoise <- local({ memory <- list() function(x) { valueName <- as.character(x) if (!is.null(memory[[valueName]])) return(memory[[valueName]]) if (x == 0) return(0) if (x == 1) return(1) res <- Recall(x - 1) + Recall(x - 2) memory[[valueName]] <<- res # store results res } })

What we do is:

1. Check if we know the results already
   * If so, return the result and stop here (avoid computing anything)
   * If not, continue with 2
2. Compute what we have to compute (e.g. fibonacci number)
   * Before we exit the function, we memorize the result
   * Then, return the result

So the idea is pretty simple. An additional complexity is, that we need a closure. Here we use local for that. local will create a new scope (environment) and run (evaluate) the code in that environment. Thus the function will have access to that environment, i.e. it has access to memory, but we do not see memory in the global environment: it is local to the function definition. Further, we need the super assignment operator so we can make assignments to memory. Alright, let’s see what we’ve gained aside from more abstraction and code:

rbenchmark::benchmark( baseR = fibR(N), Rcpp = fibRcpp(N), memoization = fibRMemoise(N), columns = c("test", "replications", "elapsed", "user.self"), order = "elapsed", replications = 1 )

## test replications elapsed user.self ## 3 memoization 1 0.00 0.00 ## 2 Rcpp 1 0.04 0.04 ## 1 baseR 1 32.03 31.67

See? R is faster than C++ after all! Should you have the time to wait for the C++ implementation we can check how far we can go, and it is the implementation in C++ that reaches limits, fast.

N <- 50 # not very far, but with memoization Int64 is the limit. rbenchmark::benchmark( # baseR = fibR(N), # not good anymore! Rcpp = fibRcpp(N), memoization = fibRMemoise(N), columns = c("test", "replications", "elapsed", "user.self"), order = "elapsed", replications = 1 )

## test replications elapsed user.self ## 2 memoization 1 0.00 0.00 ## 1 Rcpp 1 87.67 87.24

Nice, pretty efficient hack that is. It also demonstrates why performance comparisons between languages is like comparing apples and, yeah whatever, R is just fast 

**Memoization in R**

There are some issues with the above definition: it is not very generic. Our definition of memoization is still tied to the definition of the fibonacci numbers. So, instead, we can define a higher order function, which will separate memoization from the algorithm:

memoise <- function(fun) { memory <- list() function(x) { valueName <- as.character(x) if (!is.null(memory[[valueName]])) return(memory[[valueName]]) res <- fun(x) memory[[valueName]] <<- res res } }

This is a perfectly fine definition of the technique and not very long or complex. In principle it is the very same thing you will find in R.cache and memoise. Obviously these two packages add some features, e.g. how and where do you want to store the memory, maybe on disk? The above only allows for one argument, which is also solved in the mentioned packages; among some other useful things.

**Tweaks, hacks, and more abstraction**

I really like the definition above. It is so easy to implement and can be so very useful. For our example there is an important caveat: It does not memoize recursive function definitions! This is also true for the packages I mentioned. When you go into the details it is actually not that simple to write a higher order function implementing memoization for recursive function calls. Maybe you come up with something simpler than me, then I would really be interested to see the solution.

Here is one way to do it. It ain’t pretty but it will work:

memoise2 <- function(fun) { functionTemplate <- templates::tmpl( "local({ memory <- list() function() { valueName <- as.character(openssl::md5(serialize(list(), NULL))) if (!is.null(memory[[valueName]])) return(memory[[valueName]]) res <- memory[[valueName]] <<- res res } })" ) templates::tmplEval( functionTemplate, args = paste(names(formals(fun)), collapse = ", "), body = paste(deparse(body(fun)), collapse = "\n"), .envir = environment(fun) ) }

Let’s not go too much into the details, I’ll leave that to you. The templates package comes in handy but it is not too hard to implement without. Basically we “paste”-together a function definition. Before you start yelling, let’s just look at the benchmark (and move on):

N <- 50 rbenchmark::benchmark( # baseR = fibR(N), Rcpp = fibRcpp(N), memoization = fibRMemoise(N), memoization2 = memoise2(fibR)(N), columns = c("test", "replications", "elapsed", "user.self"), order = "elapsed", replications = 1 )

## test replications elapsed user.self ## 2 memoization 1 0.00 0.00 ## 3 memoization2 1 0.08 0.08 ## 1 Rcpp 1 92.03 91.69

Nice. I do not know if it is still more efficient to come up with something like memoise2 or just learn some basic C++. Anyways, these are some tricks for your bag of tools.

**When to use Memoization**

When and why should you use Memoization? It is not like we implement something like the fibonacci sequence on a daily basis. And even if, we would do it differently. Actual use cases I have in mind are quite different. To give you some ideas:

* We can reduce the number of calls against an API. Most providers, e.g., Google for Google Maps, will restrict the number of calls per day you are allowed to make. You can use memoization to quickly build up an in-memory or on-disk cache. This will allow you to quickly switch back to an “old” configuration without the need to query the API again.
* Calls against databases or generally loading data. Think of an Shiny application where a change in the UI will trigger a call to a database. E.g., when you have parameterized queries. When you cache the results of these queries you may speed up the app considerably when a user is changing back and forth between settings.

Whatever we do there needs to be an important property so that memoization is useful. Wikipedia says:

*A function can only be memoized if it is referentially transparent; that is, only if calling the function has exactly the same effect as replacing that function call with its return value. (Special case exceptions to this restriction exist, however.)*

Source: <https://en.wikipedia.org/wiki/Memoization#Overview>

In other words: we need to be careful that the results of our functions really only depend on the input parameters. Do you trust your database connections or API calls to have this property? If so, memoization can be useful. But be careful: memoization leads to caching, caching leads to state management (when and how do you update the cache?), and that leads to hard-to-debug problems: A lot of fun.