Coding Efficiency Demo

Jimmy Zhang

2022-10-17

1. What Is Coding Efficiency?

Efficiency means "working well to avoid wasting energy, efforts, time, money, etc." It has a formal definition:

$$\eta = \frac{W(Work)}{Q(Effort)}.$$

In words, it means how much work has been done in a unit of effort. A high efficiency means more work has been done in a unit of effort.

Two types of efficiency are defined:

1. *Algorithmic Efficiency*: how quickly the computer can undertake a piece of work given a particular piece of code.
2. *Programmer Productivity*: the amount of useful work a person **(not a computer)** can do per unit time.

We must make a balance between algorithmic efficiency and programmer productivity: It is possible that we may pay too much human efforts to optimize an algorithm (e.g., spend 50 hours to make a function generalizable). The works that we complete for this optimization may seem not enough given the efforts we make, although we complete a more efficient algorithmic.

2. Why can we achieve efficient programming in R?

R has some significant advantages:

- R is not compiled but it calls compiled code. It removes the laborious stage of compiling your code before being able to run it.
- R is a functional and object orientated language. It is possible to write complex and flexible functions in R that get a huge amount of work done with a single line of code.

Object-oriented programming (OOP): a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects. Python, Java, C++ are all examples of OOP languages.

- R uses RAM for memory.

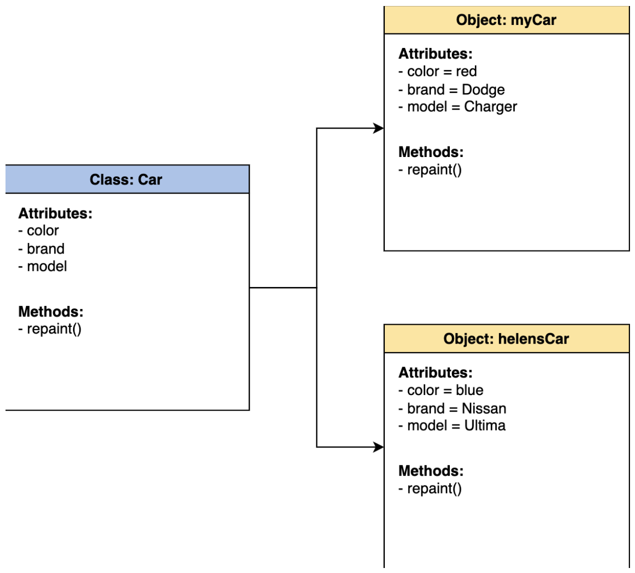


Figure 1: An illustration of an OOP example.

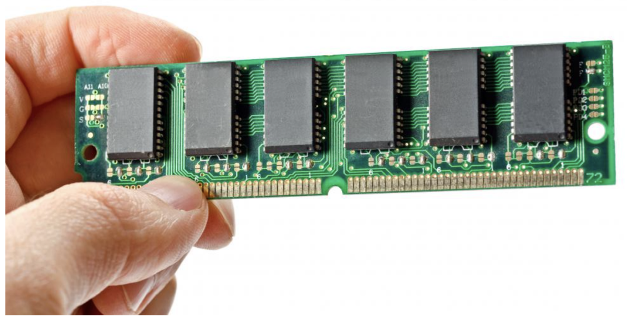
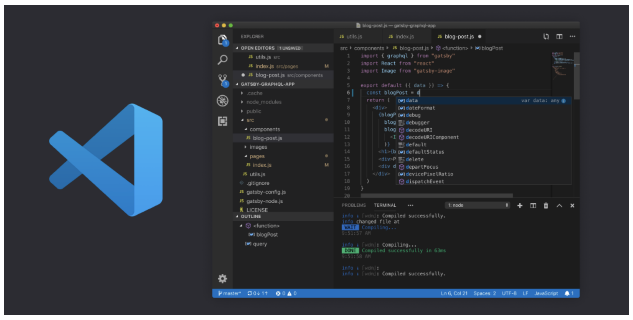


Figure 2: An illustration of RAM.

RAM (random access memory) is a computer's short-term memory, where the data that the processor is currently using is stored. Your computer can access RAM memory much faster than data on a hard disk, SSD, or other long-term storage device, which is why RAM capacity is critical for system performance.

- R is supported by excellent Integrated Development Environments (IDEs).

An integrated development environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools and a debugger.



```
import { graphql } from "gatsby"
import React from "react"
import Image from "gatsby-image"

export default ({ data }) => {
  const blogPost = d
  return (
    <div>
      {blogP
        blog
        blog
          <I
        })
      <h1>{b
      <div>P
      <div d
    </div>
  )
}
```

```
data                     var data: any
dateFormat
debug
debugger
decodeURI
decodeURIComponent
default
defaultStatus
delete
departFocus
devicePixelRatio
dispatchEvent
```

```
info ｢wdm｣: Compiled successfully.
info changed file at
WAIT Compiling...
9:51:57 AM

info ｢wdm｣: Compiling...
DONE Compiled successfully in 63ms
9:51:58 AM

info ｢wdm｣:
info ｢wdm｣: Compiled successfully.
```

Figure 3: An illustration of IDE.

- R has a strong user community. If you encounter a problem that has not yet been solved, you can simply ask the community:
  - StackOverflow
  - R-bloggers

3. Some basic skills and conventions to promote efficient coding in R

1. Conventions

- Packages names are in bold: **lavaan**.
- Functions and R objects are in a code font: `ggplot()`, `x`

2. Benchmark

- It allows you to keep track of the performce of your function: **microbenchmark**

```
library("microbenchmark")
df = data.frame(v = 1:4, name = letters[1:4])
microbenchmark(df[3, 2], df[3, "name"], df$name[3])
```

```
## Unit: nanoseconds
##            expr  min     lq     mean median     uq   max neval
##        df[3, 2] 7187 7698.0 10620.47   8063 9001.5 61780   100
##   df[3, "name"] 7414 7766.0 10658.96   8135 9332.5 65978   100
##      df$name[3]  709  822.5  1740.55    868 1027.0 32235   100
```

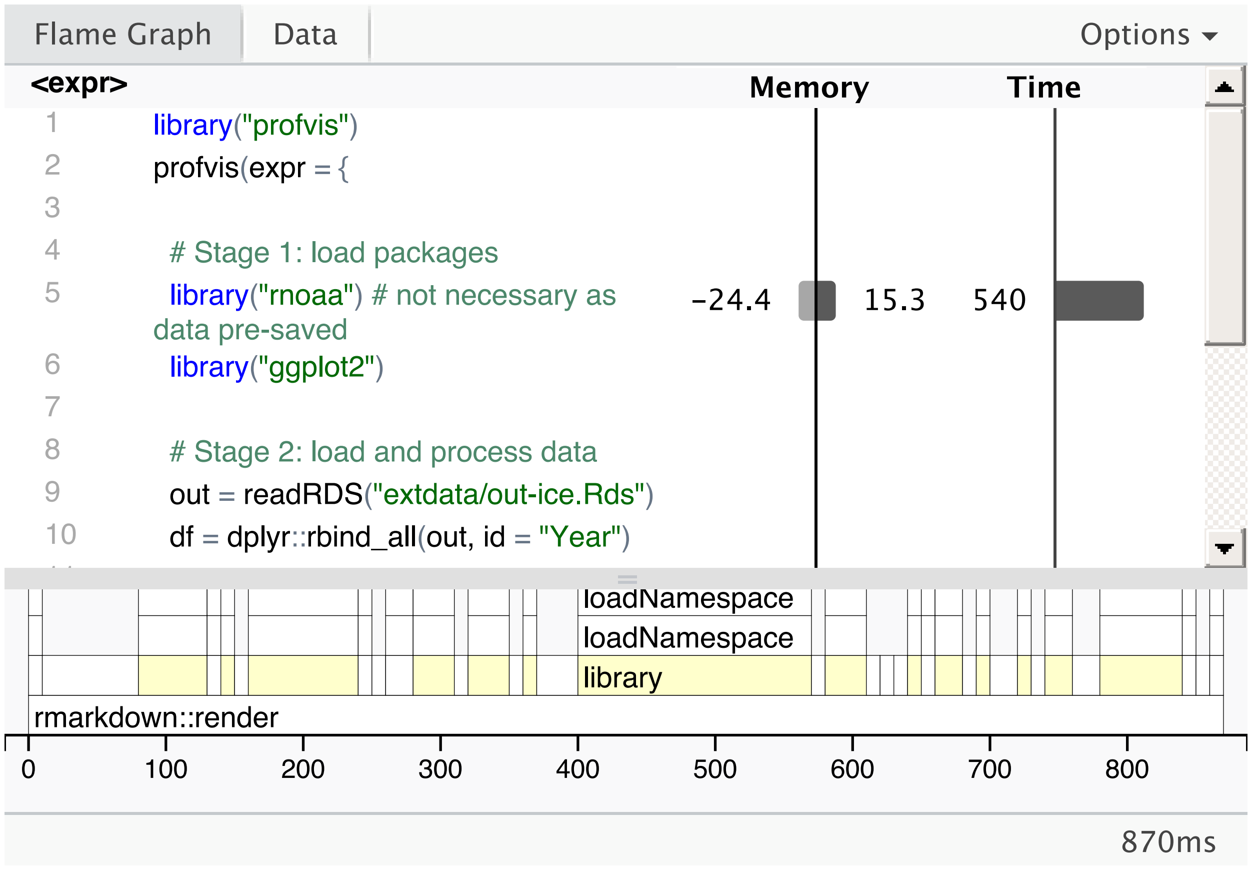
3. Profiling

- As R users, we may want our codes to run faster. However, it's not always clear how to accomplish this.
- Profiling is about testing large chunks of code. It is able to identify bottlenecks in your R scripts.

```r
library("profvis")
profvis(expr = {

  # Stage 1: load packages
  library("rnoaa") # not necessary as data pre-saved
  library("ggplot2")

  # Stage 2: load and process data
  out = readRDS("extdata/out-ice.Rds")
  df = dplyr::rbind_all(out, id = "Year")

  # Stage 3: visualise output
  ggplot(df, aes(long, lat, group = paste(group, Year))) +
    geom_path(aes(colour = Year))
  ggsave("figures/icesheet-test.png")
}, interval = 0.01, prof_output = "ice-prof")
```

```
## Warning in gzfile(file, "rb"): cannot open compressed file 'extdata/out-
## ice.Rds', probable reason 'No such file or directory'
```

```
## profvis: code exited with error:
## cannot open the connection
```



```
library("profvis")
profvis(expr = {

  # Stage 1: load packages
  library("rnoaa") # not necessary as
data pre-saved
  library("ggplot2")

  # Stage 2: load and process data
  out = readRDS("extdata/out-ice.Rds")
  df = dplyr::rbind_all(out, id = "Year")
```

4. Efficient Programming in R

1. Some broad but fundamental ideas

Ultimately calling an R function always ends up calling some underlying *C/Fortran code*, which are low level languages that demand more from the programmer. They force you to declare the type of every variable used, give you the burdensome responsibility of memory management and have to be compiled.

For example, if you call a base R function `runif`, you can view the source code of this function:

```
> getAnywhere(runif())
A single object matching ‘runif’ was found
It was found in the following places
  package:stats
  namespace:stats
with value

function (n, min = 0, max = 1)
.Call(C_runif, n, min, max)
<bytecode: 0x7f91a0bbf520>
<environment: namespace:stats>
```

Figure 4: An illustration of runif.

We can see that `runif` only contains a single line that calls `C_runif()`. With this being said, R is a higher level language that somehow “simplifies” our coding process.

A golden rule in R programming is to access the underlying C/Fortran routines as quickly as possible: the fewer functions calls required to achieve this, the better.

```
x <- c(1, 2, 3, 4, 5)

# Method 1:

x <- x + 1

# Method 2:

for(i in seq_len(length(x)))
  x[i] = x[i] + 1

microbenchmark(x <- x + 1,
for(i in seq_len(length(x)))
  x[i] = x[i] + 1 )
```

```
## Unit: nanoseconds
##                                          expr     min        lq       mean
##                                    x <- x + 1     180     252.5    1611.25
##  for (i in seq_len(length(x))) x[i] = x[i] + 1 1519695 1817231.0 2168807.87
##   median      uq     max neval
##     1353    2886    7160   100
##  1933282 2204868 5892340   100
```

Another useful rule is to pre-allocate your vector and then fill in values. A best way to avoid massive computation is to avoid building an object (e.g., vector, data frame, list, etc).

```
method1 = function(n) {
  vec = NULL # Or vec = c()
  for(i in seq_len(n))
    vec = c(vec, i)
  vec
}

method2 = function(n) {
  vec = numeric(n)
  for(i in seq_len(n))
    vec[i] = i
  vec
}

method3 = function(n) seq_len(n)
```

Method 1 creates a null vector that contains nothing.

```
vec = NULL
length(vec)
```

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

Then it adds values in this vector and builds up the vector gradually.

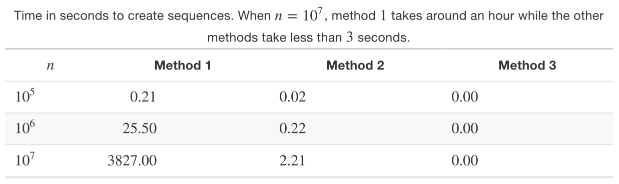
Method 2, however, creates a an object of the final length and then changes the values in the object by subscripting.

```r
n = 100
vec = numeric(n)
length(vec)
```

```
## [1] 100
```

Method 3 directly creates the final object.

It is not difficult to guess which one wins in terms of efficiency.



Time in seconds to create sequences. When $n = 10^7$, method 1 takes around an hour while the other methods take less than 3 seconds.

n	Method 1	Method 2	Method 3
10^5	0.21	0.02	0.00
10^6	25.50	0.22	0.00
10^7	3827.00	2.21	0.00

Figure 5: An illustration of Three Methods Comparison.

2. Vectorization

Given the faster speed of accessing the underlying C/Fortran routines, we can reduce the number of function calls required by working on vectorization.

```r
# Using for loop
log_sum = 0
for(i in 1:length(x))
  log_sum = log_sum + log(x[i])

# Using vectorized method
log_sum = sum(log(x))
```

One of my own challenges is that I particularly like `for loop` and `while loop` to construct complex functions because this is one of the simplest ways to get it started. Although this way of thinking works, sometimes it'll greatly increase the consumption of computation works. We can spend a little bit more time to challenge ourselves: how to make it simpler by vectorization?