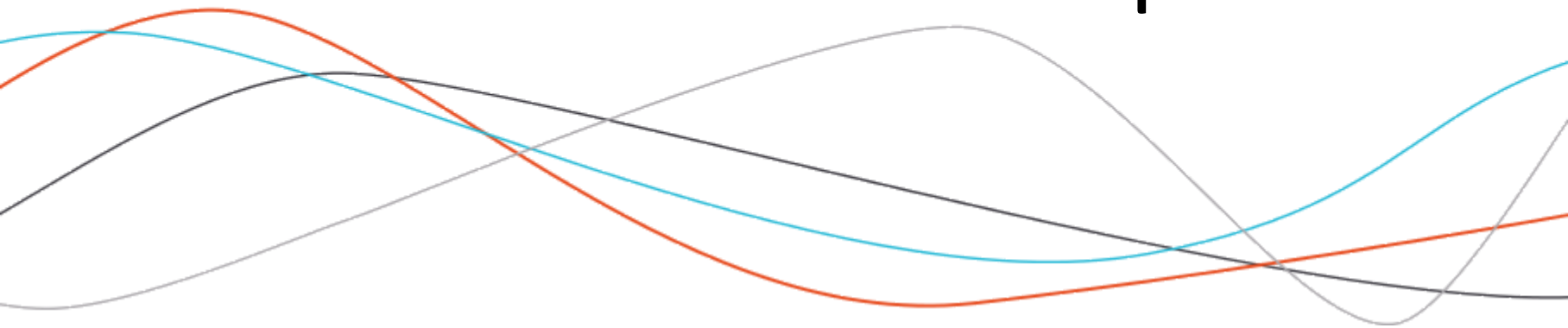
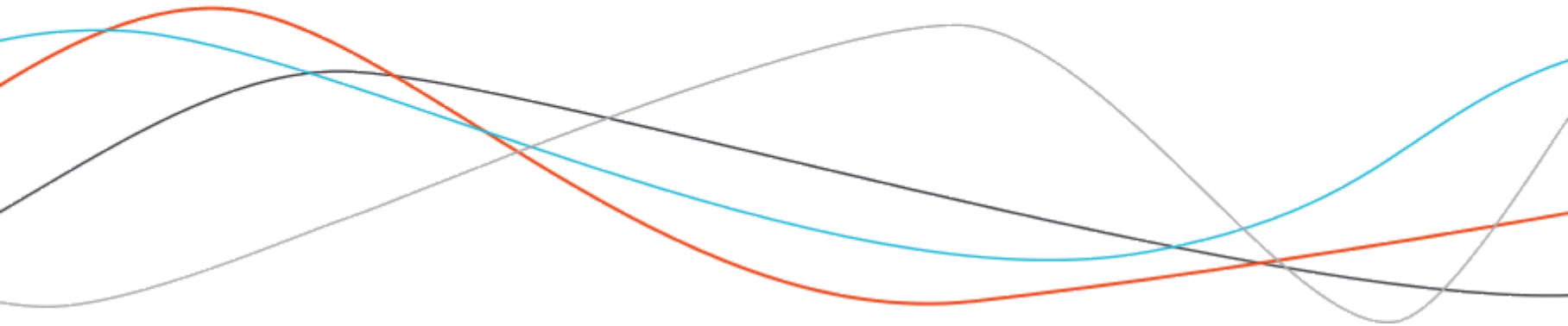
Building a package that lasts

Part 4: Optimisation



Developer oriented optimisation



Developer oriented optimisation

The biggest rule to remember:

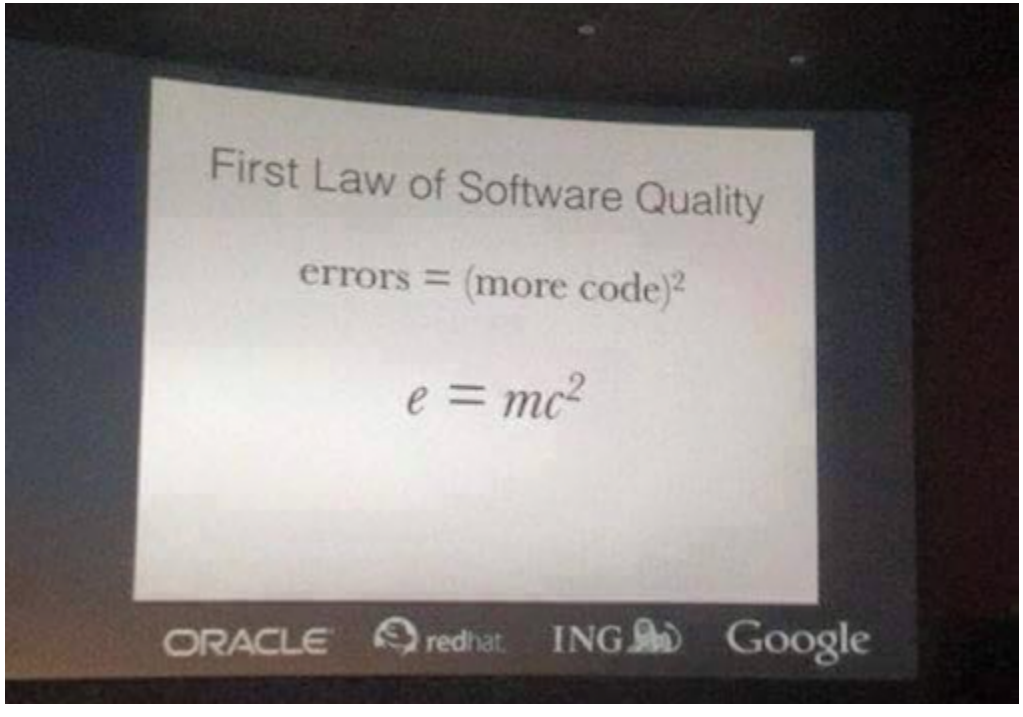
> If you need to copy and paste something, write a function.

```
a <- function(num) {
  res <- num * 10 / pi
  round(res, 2)
}
b <- function(num) {
  res <- num * 20 / pi
  round(res, 2)
}
c <- function(num) {
  res <- num * 30 / pi
  round(res, 2)
}
```

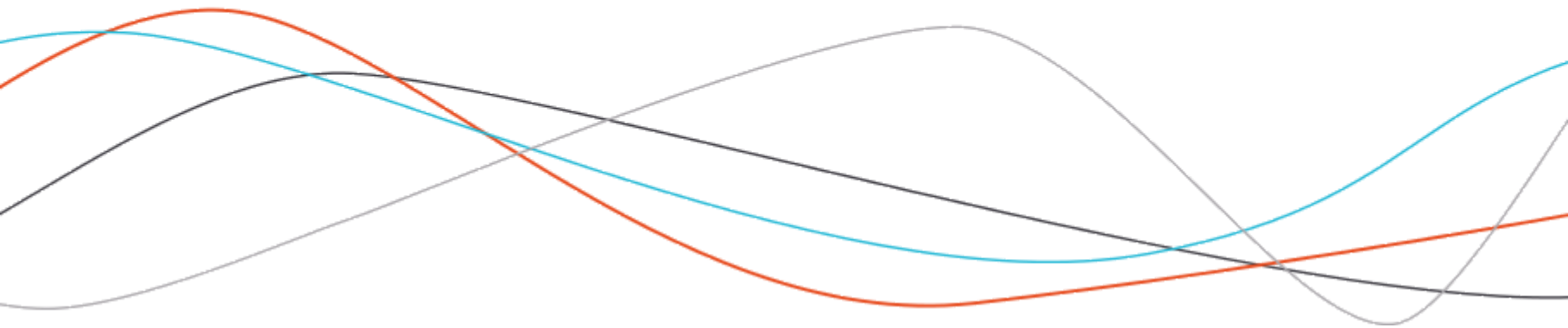
```
rt <- function(n, m, d = 2) {
  round( (n * m / pi), d )
}
a <- function(num) {
  rt(num, 10)
}
b <- function(num) {
  rt(num, 20)
}
c <- function(num) {
  rt(num, 30)
}
```

Developer oriented optimisation

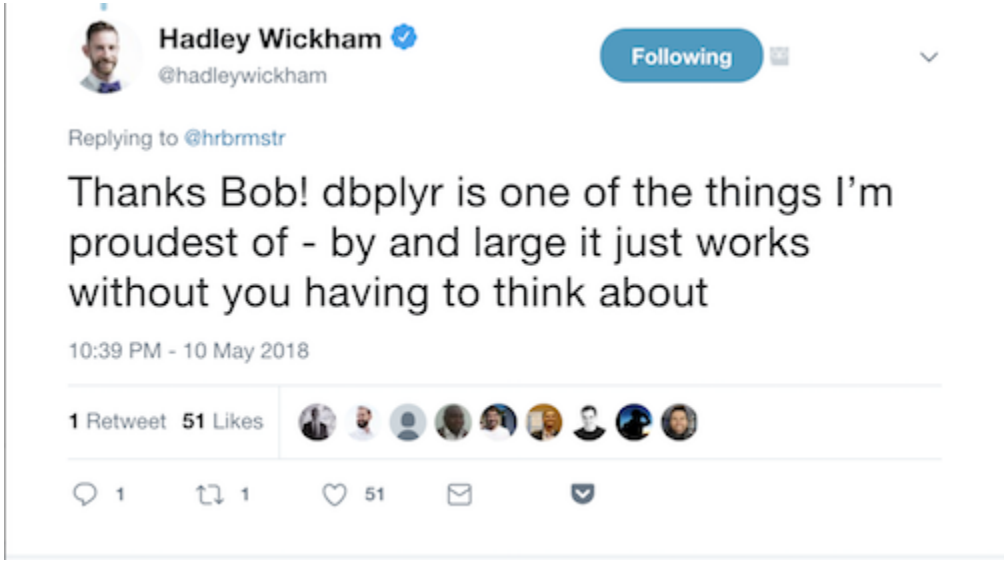
The more code you have, the more errors you will get



UX oriented optimisation



UX oriented functions



The "unfair burden" of the package developer: **you should make the complex tasks seem easy / invisible for the user**.

Anticipate errors

Two things you should keep in mind:

- R is not reknown for its clear error messages.
- The end user will, at some point, try to run your function with weird arguments.

So basically, you should expect a user to use a weird input, to get back a cryptic message, and to:

- open a Stackoverflow question (in the best case scenario)
- open an issue on your GitHub
- simply stop using your package because "it doesn't work"

Defensive programming

> Debugging is the art and science of fixing unexpected problems in your code.
>
> *Wickham H., Advanced R*

Not all errors are unexpected. To prevent errors, adopt a "defensive programming" strategy: **anticipate errors and/or unexpected behaviors, in order to manage them upstream and to inform the user**.

Three types of alerts exist:

- `stop` : an error, stops the execution of the program.
- `warning` : an alert, informs of a potential error, does not however prevent the program from working.
- `message` : a message printed on the console, for information purposes.

Using {attempt} for defensive programming

```
# remote::install_github("ColinFay/attempt")
# install.packages("attempt")
library(attempt)
my_sqrt <- function(num) {
  stop_if_not(.x = num, .p = is.numeric,
              msg = "You should enter a number")
  sqrt(num)
}
my_sqrt(1)
```

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

```
my_sqrt("1")
```

```
#> Error: You should enter a number
```

Stop, alert, inform

```
grep("stop", ls("package:attempt"),value = TRUE)
```

```
#> [1] "stop_if"      "stop_if_all"  "stop_if_any"  "stop_if_none"
#> [5] "stop_if_not"
```

```
grep("warn", ls("package:attempt"),value = TRUE)
```

```
#> [1] "warn_if"         "warn_if_all"     "warn_if_any"     "warn_if_none"
#> [5] "warn_if_not"     "with_warning"    "without_warning"
```

```
grep("message", ls("package:attempt"),value = TRUE)
```

```
#> [1] "message_if"      "message_if_all"  "message_if_any"  "message_if_none"
#> [5] "message_if_not"  "with_message"    "without_message"
```

Speed optimisation

Not all functions need to be optimised.

Don't spend a week optimising for 10 microseconds.

You can always do more, you need to know when to stop.

Unless really needed, do not spend too much time optimising for speed.

Benchmarking your code

How can I know if my code is slow? => BENCHMARK!

You can do this in base R with `system.time`:

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

```
#>    user  system elapsed
#>   0.007   0.005   1.002
```

But there are more automated ways to do this.

Benchmark with {microbenchmark}

{microbenchmark} is a wrapper around `system.time` that automates benchmark.

More efficient : allows the comparison to be repeated many times, and displays the maximum, minimum, average and median of the elapsed time as a result.

We always start by making sure the two results are `all.equal`.

```
all.equal(sum(1,2), 1+2)
```

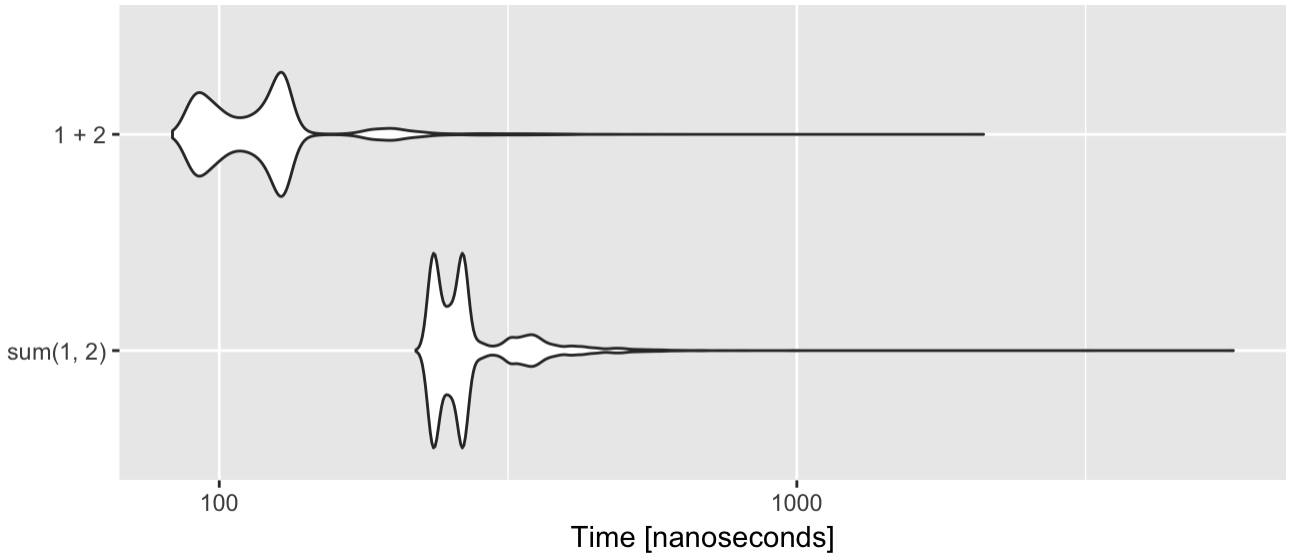
```
#> [1] TRUE
```

```
microbenchmark::microbenchmark(sum(1,2), 1 + 2,
               times = 10000)
```

```
#> Unit: nanoseconds
#>       expr min  lq     mean median  uq   max neval cld
#>  sum(1, 2) 168 181 226.7462    201 243 13262 10000   b
#>      1 + 2  61  74 101.0710     92  99 40196 10000  a
```

Benchmark with {microbenchmark}

```
library(ggplot2)
library(microbenchmark)
microbenchmark(sum(1,2), 1 + 2,
               times = 10000) %>% autoplot()
```



Benchmark with {bench}

More recent package (mid-april 2018).

With {bench}, you don't have to test for equality of results before running the benchmark.

More human readable results.

Only on GitHub for now.

Read more : http://bench.r-lib.org/

Benchmark with {bench}

```
bench::mark(sum(1,3), 1 + 2,
     iterations = 10000)
```

```
#> Error: All results must equal the first result:
#>   `sum(1, 3)` does not equal `1 + 2`
```

```
bench::mark(sum(1,2), 1 + 2,
     iterations = 10000)
```

```
#> # A tibble: 2 x 14
#>   expression     min    mean median    max `itr/sec` mem_alloc  n_gc n_itr
#>   <chr>      <bch:t> <bch:t> <bch:> <bch:>     <dbl> <bch:byt> <dbl> <int>
#> 1 sum(1, 2)    169ns   259ns  234ns 31.8µs  3863516.        0B    0. 10000
#> 2 1 + 2         55ns   114ns   67ns 10.6µs  8741809.        0B    0. 10000
#> # ... with 5 more variables: total_time <bch:tm>, result <list>,
#> #   memory <list>, time <list>, gc <list>
```

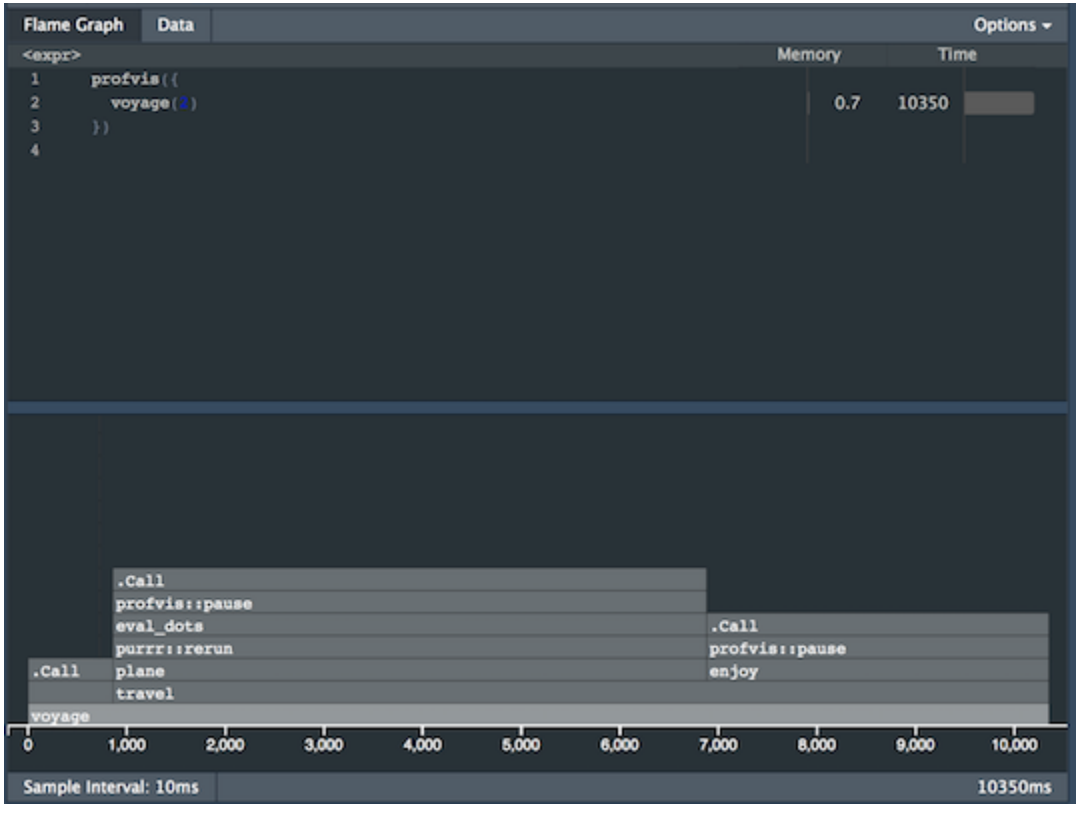
Identify bottlenecks with profiling

```
voyage <- function(days) {
  print("take your breath")
  profvis::pause(1)
  print("Let's go!")
  travel(transport = 2, stay = days)
}
travel <- function(transport, stay) {
  plane(transport) + enjoy(stay)
}
plane <- function(times){
  purrr::rerun(times, profvis::pause(sample(1:5, 1)))
}
enjoy <- function(stay){
  beer <- stay ^ 2
  profvis::pause(beer)
}
```

Identify bottlenecks with profiling

```
library(profvis)
profvis({
  voyage(2)
})
```

Identify bottleneck with profiling



Optimisation, some ground rules

return and stop as soon as possible

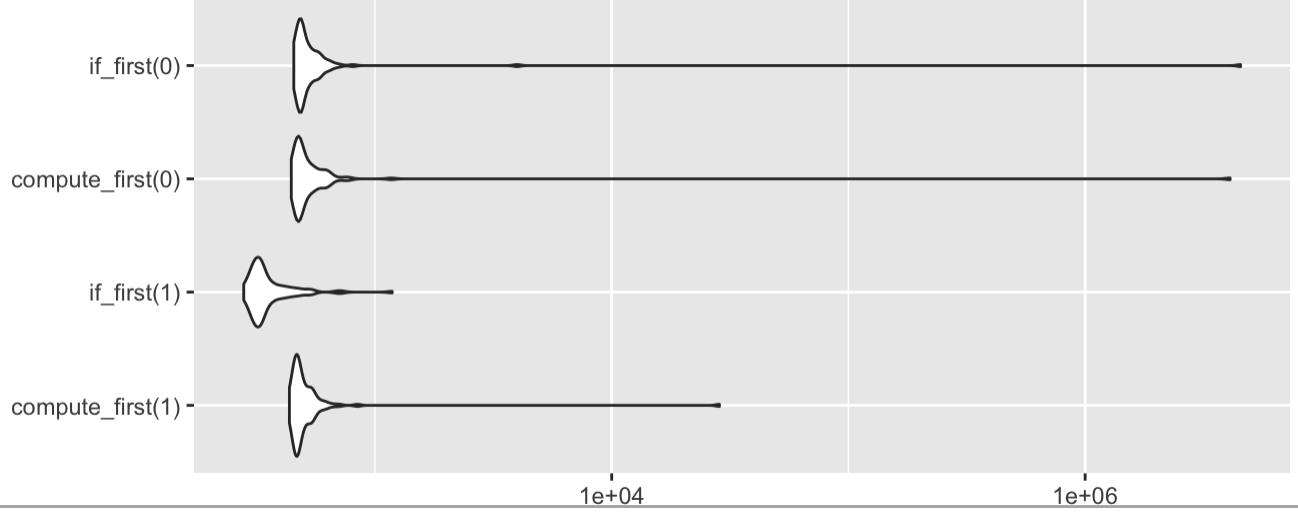
If a test stops the execution, do it first:

```
compute_first <- function(a){
  d <- log(a) * 10 + log(a) * 100
  if( a == 1 ) return(0)
  return(d)
}
if_first <- function(a){
  if( a == 1 ) return(0)
  d <- log(a) * 10 + log(a) * 100
  return(d)
}
```

Optimisation, some ground rules

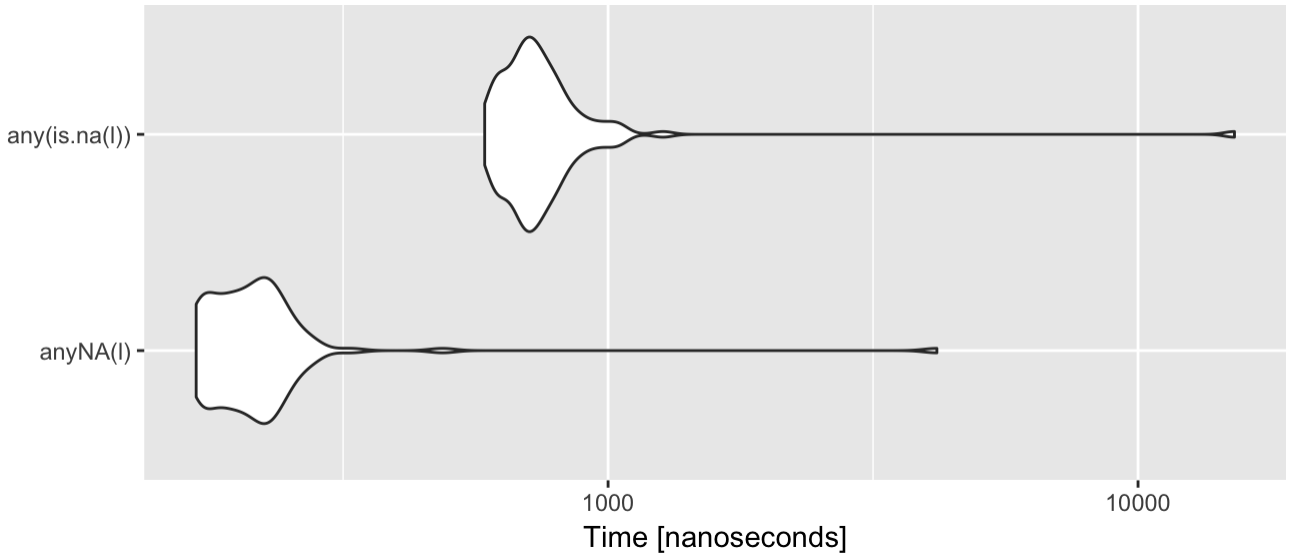
return and stop as soon as possible

```
microbenchmark(compute_first(1), if_first(1),
               compute_first(0), if_first(0), times = 100) %>%
  autoplot()
```



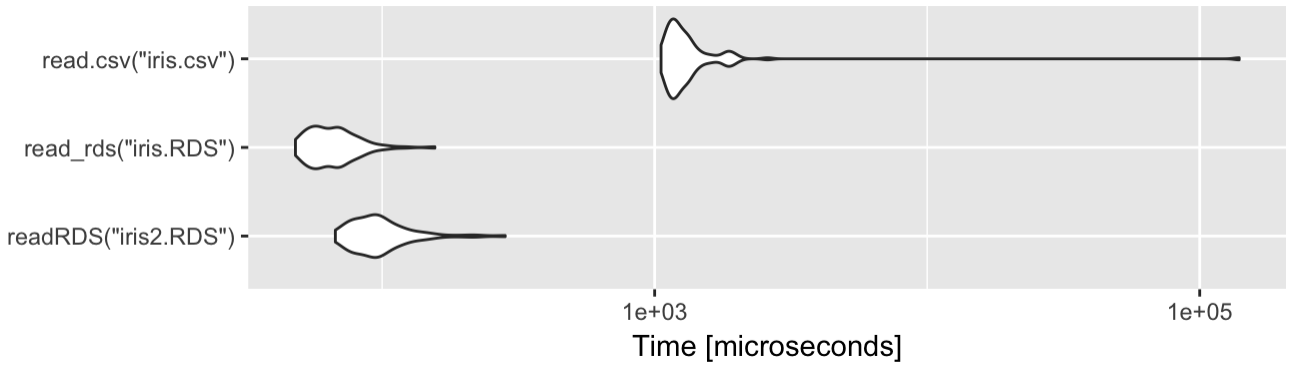
Minimise the number of function calls

```
l <- c(1:100, NA)
microbenchmark(anyNA(l),
               any(is.na(l)),
               times = 100) %>% autoplot()
```



Use R formats

```
library(readr)
write_rds(iris, "iris.RDS")
saveRDS(iris, "iris2.RDS")
write.csv(iris, "iris.csv")
microbenchmark(readRDS("iris2.RDS"),
               read_rds("iris.RDS"),
               read.csv("iris.csv")) %>% autoplot()
```



Let's practice !