Introduction to Microbenchmark

Why `microbenchmark`

`Microbenchmark` is an R package that allows you to measure the run time of a small block of code (mostly a function run). The most imporatant use of this package is to compare the run time of different functions/algorithms that do the same or similar things.

Here are two common use cases that I find:
1.To find the function that scales better with data or input size.
2.To understand a function's run time performance with different parameters. For instance, using the time series modeling function `arima` with an additional autoregressive parameter can increase run time by a factor of 5 to 10 but not much prediction performance increase (see more details from my project report on github).

Function `microbenchmark` Parameters

Here is the skeleton code of a `microbenchmark` function call.

```
microbenchmark(
  expression_1 or function_call_1, # e.g print("hello world")
  expression_2 or function_call_2, # e.g print("hello rstat")
  expression_3 or function_call_3, # e.g print("hello tidyverse")
  times = 100,
  unit = "ms"
)
```

First `microbenchmark` takes as many expressions as you want to speed tests on , then you specify the configurations of the speed tests via the following two parameters (some complicated parameters are omitted in this article) :

times: Number of times to evaluate the expression. By default `microbenchmark` runs each expressions **100 times.** If you expect your expressions will take about 5 seconds to run a single time, you should change this parameter accordingly so you don't have to wait too long.
unit: Specify the units of time used. You can also change the `unit` to "ns" (10^{-9} second), "us" (10^{-6} second), "ms" (10^{-3} second), "s" (second).

Microbenchmark in Action

Comparing a single function: substr vs substring

Assuming you are trying to extract the prefix of a string (e.g taking date from a timestamp).
From stack overflow you find that you can use either `substr` or `substring`. If you are too lazy to read the documentation and find the exact difference, you can try to use microbenchmark to compare their speed before making your choice.

First, we can see both `substr` and `substring` can take the first 3(n) characters from a string.

```
substr("hello", 1, 3)
```

```
## [1] "hel"
```

```
substring("hello", 1, 3)
```

```
## [1] "hel"
```

Then, we can use microbenchmark to see who is faster.

```
str_vs_string = microbenchmark(
  substr("hello", 1, 3), #expression 1
  substring("hello", 1, 3), #expression 2
  times=1000 #run some more times since the function is fast to run,
)
```

For a quick comparison, you can use `print`, which shows the basic summary statistics(min/max/mean/median) of each expression. Function `summary` outputs the same information without showing the unit used. So you wouldn't want to use `summary`.

```
print(str_vs_string)
```

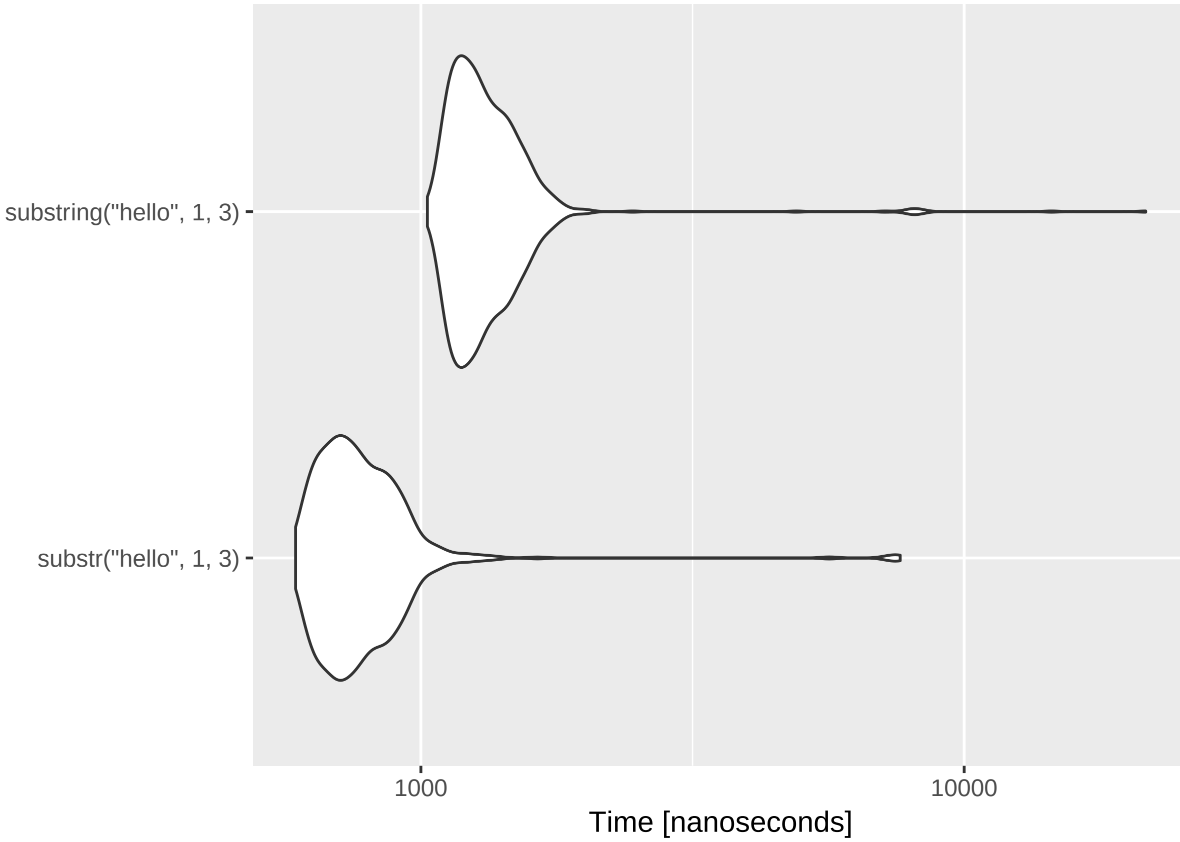
```
## Unit: nanoseconds
##                      expr  min     lq     mean median     uq   max neval
##     substr("hello", 1, 3)  588  678.5  830.971  749.5  854.0  7625  1000
##  substring("hello", 1, 3) 1028 1174.0 1394.250 1274.0 1426.5 21585  1000
```

```
summary(str_vs_string)
```

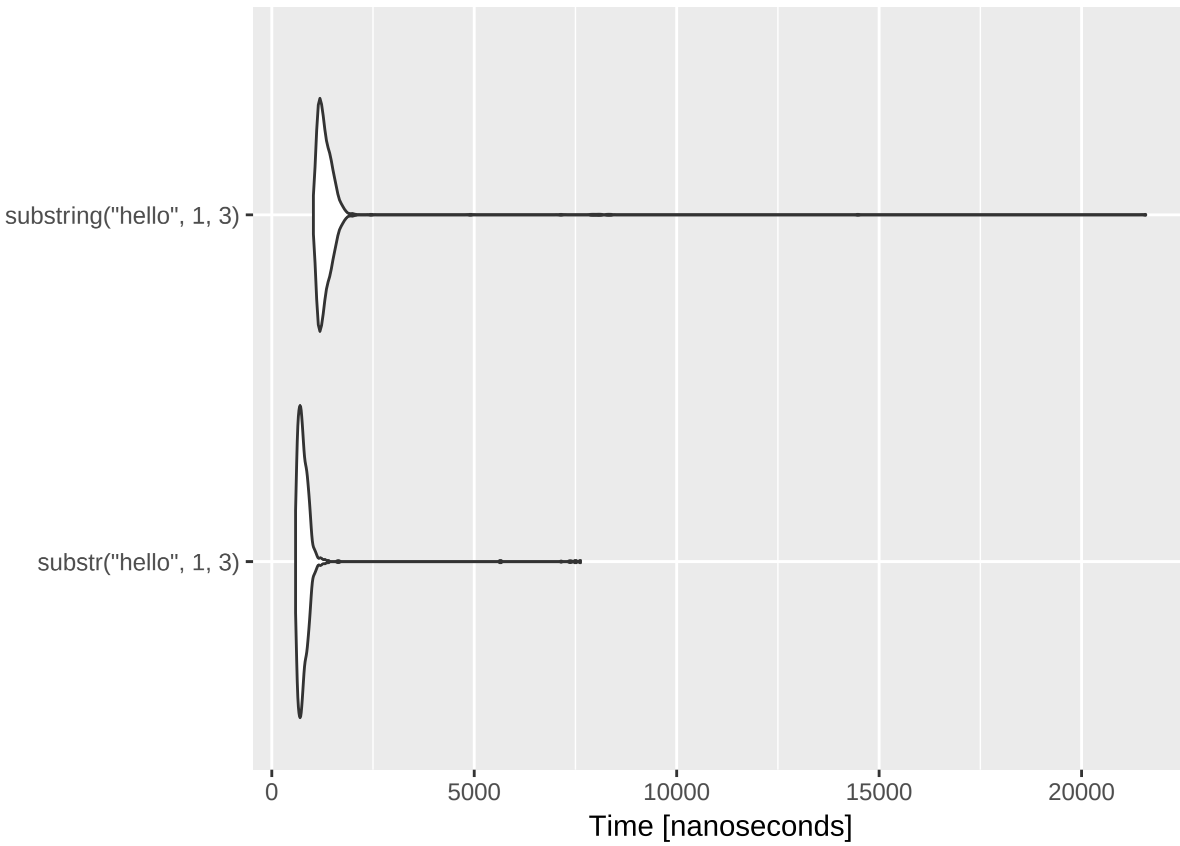
```
##                        expr  min     lq     mean median     uq   max neval
## 1    substr("hello", 1, 3)  588  678.5  830.971  749.5  854.0  7625  1000
## 2 substring("hello", 1, 3) 1028 1174.0 1394.250 1274.0 1426.5 21585  1000
```

For more details of the distribution of the run times/speed, you can use `autoplot`. By default, `autoplot` is in log10 scale, to see the original scale, you have to set parameter `log` to `FALSE`. It is better to see the plot in log scale because the distribution is usually heavily right skewed, log10 scale gives you a better visual of the run time distribution.

```
autoplot(str_vs_string)
```




```r
autoplot(str_vs_string, log=F)
```




```
hello_vec_short = sample(rep(as.character(iris$Species), 1))
vec_ben_short = microbenchmark(
  substr(hello_vec_short, 1, 3), #expression 1
  substring(hello_vec_short, 1, 3), #expression 2
  times=100
)

print(vec_ben_short)
```

```
## Unit: microseconds
##                              expr   min     lq     mean median     uq
##     substr(hello_vec_short, 1, 3) 8.412 8.6425  9.23676 8.7395 8.8525
##  substring(hello_vec_short, 1, 3) 8.949 9.1485 10.08902 9.2810 9.4260
##     max neval
##  25.801   100
##  43.167   100
```

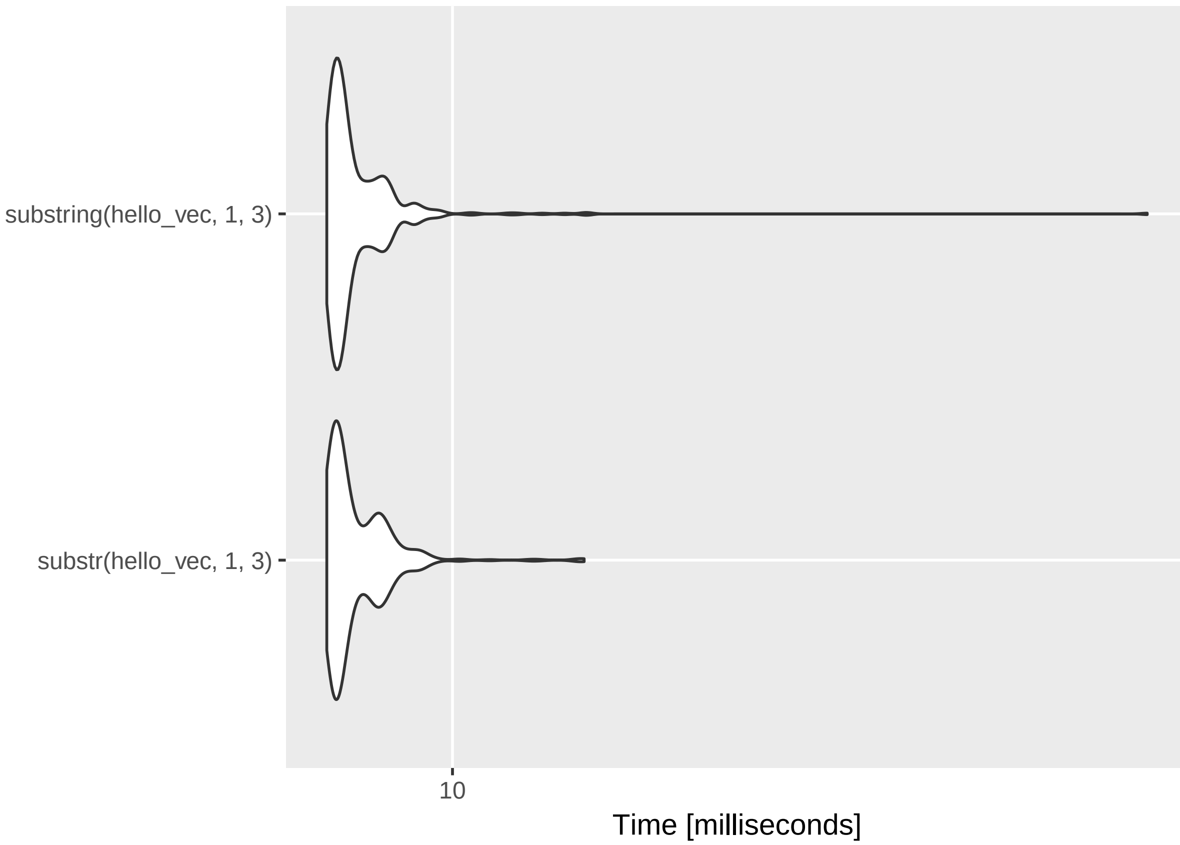
```
hello_vec = sample(rep(as.character(iris$Species), 1000))
vec_ben = microbenchmark(
  substr(hello_vec, 1, 3), #expression 1
  substring(hello_vec, 1, 3), #expression 2
  times=100
)

print(vec_ben)
```

```
## Unit: milliseconds
##                        expr      min        lq      mean    median        uq
```

```
##     substr(hello_vec, 1, 3) 7.701202 7.875874 8.200409 8.021961 8.532497
##  substring(hello_vec, 1, 3) 7.697988 7.858176 8.146987 7.979854 8.273009
##       max neval
##  10.05628   100
##  10.57804   100
```

From the plot we can see `substr` and `substring` both have similar run time distribution but `substr` is faster. What about running them at scale?

```
hello_vec = sample(rep(as.character(iris$Species), 1000)) # a vector with length 1.5 * 10^5 elements
autoplot(microbenchmark(
  substr(hello_vec, 1, 3), #expression 1
  substring(hello_vec, 1, 3), #expression 2
  times=500
))
```



We can see when applied to a vector of strings, `substr` and `substring` have very little difference. It is because difference in each run time is so small (in nanoseconds) that the difference is still small when multiple by a big factor. Next I will show you an example which two functions that do tha same but scale very differently.

Sapply vs Mutate

`sapply` and `mutate` can both apply a function to a vector. Lets compare their performances when applied to a small vector with 150 elements and a big vector with 1.5*10^5 elements.

```
iris_extended = iris
for (i in 1:10) {
  iris_extended = rbind(iris_extended, iris_extended)
```

```
}
```

```
nrow(iris)
```

```
## [1] 150
```

```
nrow(iris_extended)
```

```
## [1] 153600
```

```
# wrapper function of three_char so it can be used as a parameter
three_char = function(string) {
  substr(string, 1, 3)
}
```

When you benchmarking your functions with a large input, make sure you adjust the `times` parameter. Otherwise you may have to wait a long time for 100 evaluations to complete.

```
applys_compare = microbenchmark(
  iris$result <- sapply(iris[,5], FUN = three_char),
  iris_extended$result <- sapply(iris_extended[,5], FUN = three_char),
  iris %>% mutate(result = three_char(Species)),
  iris_extended %>% mutate(result = three_char(Species)),
  times = 1) # reduce the times run because I expect the expressions take some time to run
```

```
print(applys_compare)
```

```
## Unit: milliseconds
##                                                                   expr
##                    iris$result <- sapply(iris[, 5], FUN = three_char)
##  iris_extended$result <- sapply(iris_extended[, 5], FUN = three_char)
##                         iris %>% mutate(result = three_char(Species))
##                iris_extended %>% mutate(result = three_char(Species))
##          min          lq        mean      median          uq         max
##     1.178121    1.178121    1.178121    1.178121    1.178121    1.178121
##  1483.074464 1483.074464 1483.074464 1483.074464 1483.074464 1483.074464
##     1.841920    1.841920    1.841920    1.841920    1.841920    1.841920
##     9.331275    9.331275    9.331275    9.331275    9.331275    9.331275
##  neval
##      1
##      1
##      1
##      1
```

```
autoplot(applys_compare)
```

From the graph we can see, `mutate` is slightly better than `apply` in small datasets. However, when the length of the vector increase by a factor of 1000, `mutate`'s run time only increased by roughly a factor of 10, while `sapply`'s run time increases by about factor of 1000, linearly with the vector length.

Conclusion

R functions that do the same thing can have different run times and different scaling behaviors. Sometimes the difference is small while sometimes the difference is too big to ignore. Therefore you may want to quantify the exact different using microbenchmark.