Analyse des algorithmes de tri

M2 Data Science Algorithmique

Vincent Runge

lundi 17 mars 2025

Table des matières

---

1 Description du problème et objectif

Insertion sort is of time complexity $O(n^2)$ as heap sort is $O(n \log(n))$ (worst case complexity). We aim at highlighting two important features with this package :

1. Rcpp algorithms are **much more efficient** than their R counterpart
2. Time complexities **can be compared to** one another

All the simulations presented in this README file are available in the `myTests.R` *file in the forStudents folder which also contains the Rmd file generating this README.md.*

Details on the heapsort algorithm can be found on its wikipedia page. This gif provides a graphical representation of its mechanisms.

1.0.1 Package installation

You first need to install the `devtools` package, it can be done easily from Rstudio. We install the package from Github (remove the # sign) :

```
#devtools::install_github("vrunge/M2algorithmique")
library(M2algorithmique)
```

1.0.2 A first simple test

We simulate simple data as follows, with `v` a vector as size `n` containing all the integers from `1` to `n` (exactly one time) in any order.

```
n <- 10
v <- sample(n)
```

We've implemeted 4 algorithms :

- `insertion_sort`
- `heap_sort`
- `insertion_sort_Rcpp`
- `heap_sort_Rcpp`

They all have a unique argument : the unsorted vector `v`.

```
v
```

```
## [1]  7  5  9  3  6 10  4  2  1  8
```

```
insertion_sort(v)
```

```
## [1]  1  2  3  4  5  6  7  8  9 10
```

`insertion_sort(v)` returns the sorted vector from `v`.

1.1 The 4 algorithms at fixed data length

We run all the following examples at fixed vector length `n = 10000`.

1.1.1 One simulation

We define a function `one.simu` to simplify the simulation study for time complexity.

```
one.simu <- function(n, type = "sample", func = "insertion_sort")
{
  if(type == "sample"){v <- sample(n)}else{v <- n:1}
  if(func == "insertion_sort"){t <- system.time(insertion_sort(v))[[1]]}
  if(func == "heap_sort"){t <- system.time(heap_sort(v))[[1]]}
  if(func == "insertion_sort_Rcpp"){t <- system.time(insertion_sort_Rcpp(v))[[1]]}
  if(func == "heap_sort_Rcpp"){t <- system.time(heap_sort_Rcpp(v))[[1]]}
  return(t)
}
```

We evaluate the time with a given `n` over the 4 algorithms. We choose

```
n <- 10000
```

and we get :

```r
one.simu(n, func = "insertion_sort")
```

```
## [1] 1.821
```

```r
one.simu(n, func = "heap_sort")
```

```
## [1] 0.608
```

```r
one.simu(n, func = "insertion_sort_Rcpp")
```

```
## [1] 0.009
```

```r
one.simu(n, func = "heap_sort_Rcpp")
```

```
## [1] 0.001
```

1.1.2 Some comparisons

we compare the running time with repeated executions (`nbSimus` times)

```r
nbSimus <- 10
time1 <- 0; time2 <- 0; time3 <- 0; time4 <- 0

for(i in 1:nbSimus){time1 <- time1 + one.simu(n, func = "insertion_sort")}
for(i in 1:nbSimus){time2 <- time2 + one.simu(n, func = "heap_sort")}
for(i in 1:nbSimus){time3 <- time3 + one.simu(n, func = "insertion_sort_Rcpp")}
for(i in 1:nbSimus){time4 <- time4 + one.simu(n, func = "heap_sort_Rcpp")}
```

Rcpp is 100 to 200 times faster than R for our 2 algorithms.

```r
#gain R -> Rcpp
time1/time3
```

```
## [1] 203.8791
```

```r
time2/time4
```

```
## [1] 678.875
```

With the data length of `10000`, heap_sort runs 10 to 20 times faster than insert_sort.

```r
#gain insertion -> heap
time1/time2
```

```
## [1] 3.41613
```

```r
time3/time4
```

```
## [1] 11.375
```

The gain between the slow insertsort R algorithm and the faster heapsort Rcpp algorithm is of order 2000 ! ! !

```r
#max gain
time1/time4
```

```
## [1] 2319.125
```

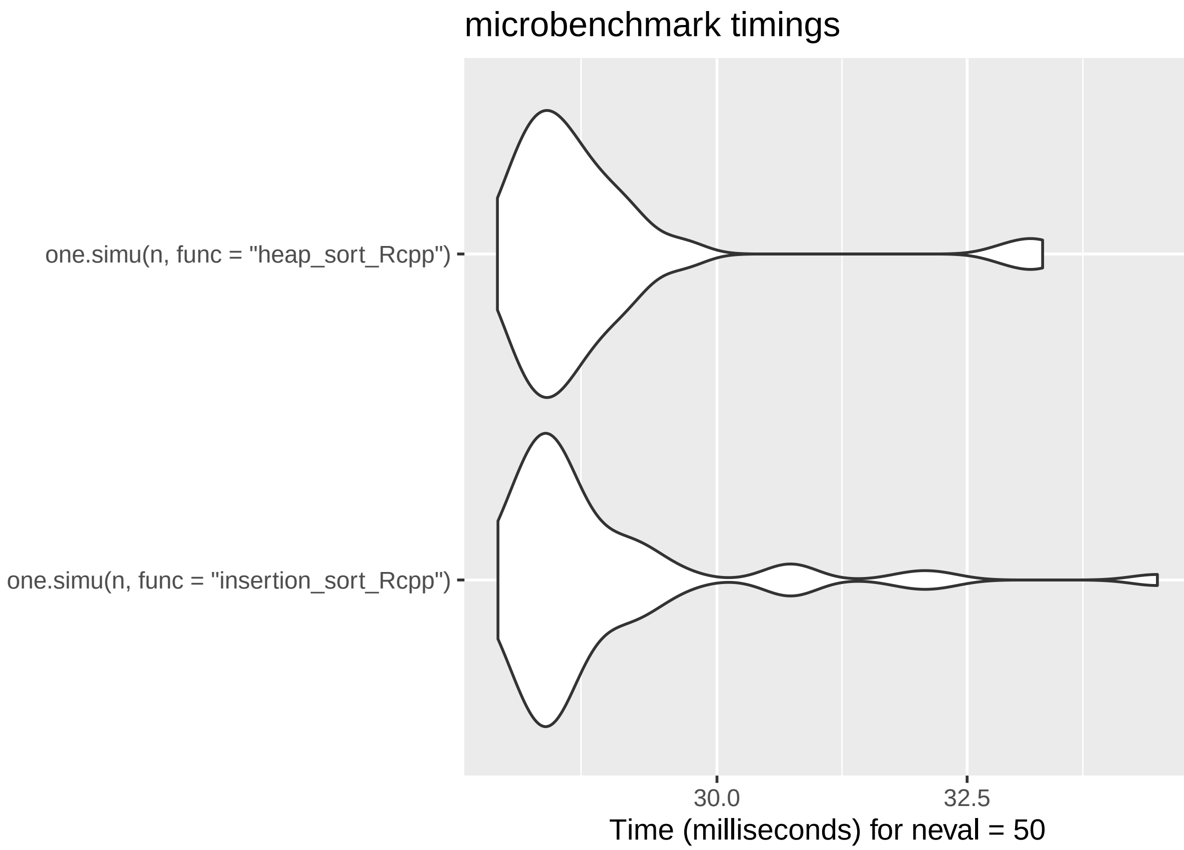
1.2 Microblenchmark

You need the packages `microbenchmark` and `ggplot2` to run the simulations and plot the results (in violin plots). We compare `insertion_sort_Rcpp` with `heap_sort_Rcpp` for data lengths `n = 1000` and `n = 10000`.

```r
library(microbenchmark)
library(ggplot2)
n <- 1000
res <- microbenchmark(one.simu(n, func = "insertion_sort_Rcpp"), one.simu(n, func = "heap_sort_Rcpp"), t
```

```
## Warning in microbenchmark(one.simu(n, func = "insertion_sort_Rcpp"),
## one.simu(n, : less accurate nanosecond times to avoid potential integer
## overflows
```

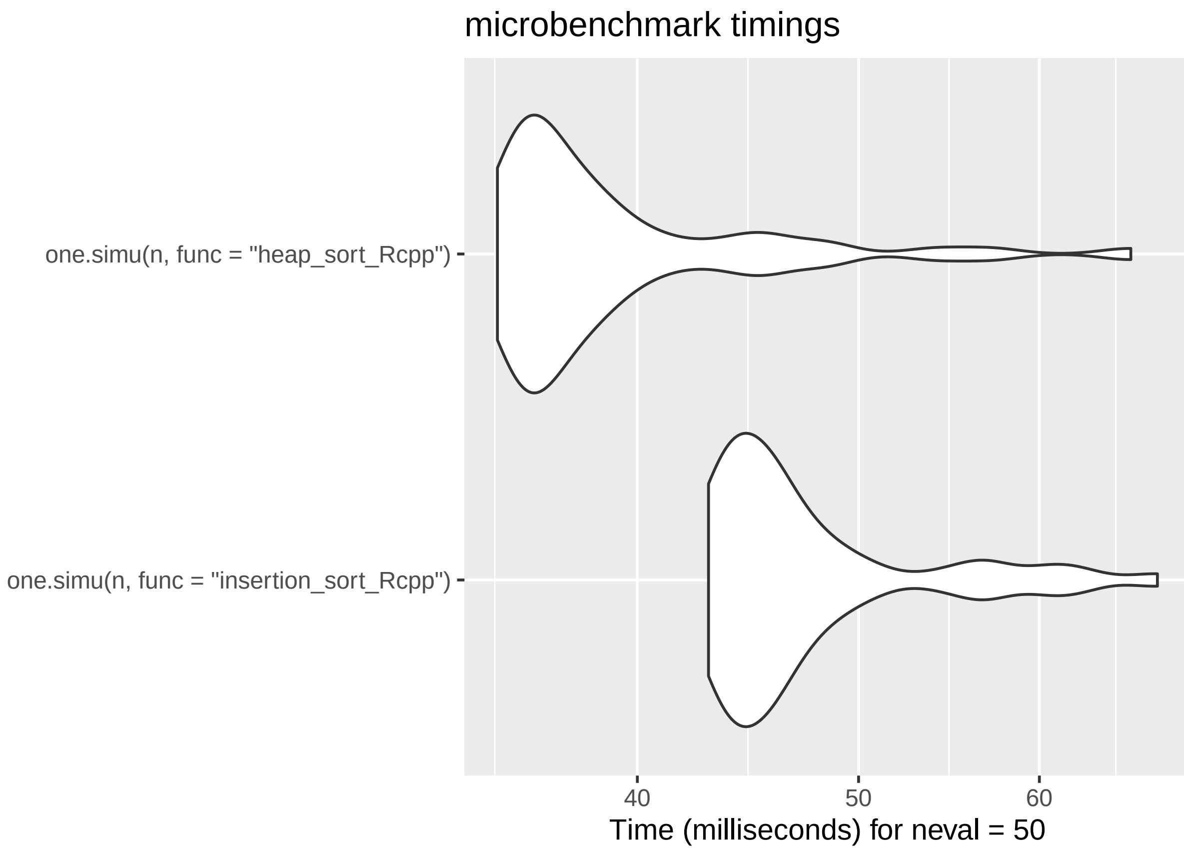
```r
autoplot(res)
```




```r
res
```

```
## Unit: milliseconds
##                                      expr      min       lq     mean   median
##  one.simu(n, func = "insertion_sort_Rcpp") 27.96967 28.32075 28.94554 28.49625
##       one.simu(n, func = "heap_sort_Rcpp") 27.96479 28.30283 28.86995 28.55609
##        uq      max neval
##  29.10926 34.54057    50
##  28.97987 33.29491    50
```

```r
n <- 10000
res <- microbenchmark(one.simu(n, func = "insertion_sort_Rcpp"), one.simu(n, func = "heap_sort_Rcpp"), t
autoplot(res)
```




```
res
```

```
## Unit: milliseconds
##                                    expr      min       lq     mean   median
##  one.simu(n, func = "insertion_sort_Rcpp") 42.97636 43.95380 47.68060 45.55670
##       one.simu(n, func = "heap_sort_Rcpp") 34.73479 35.73892 39.38135 36.70509
##        uq      max neval
##  48.18328 67.58534    50
##  39.43081 65.79742    50
```

At this data length `10000` we start having a robust difference in running time.

1.3 Time complexity

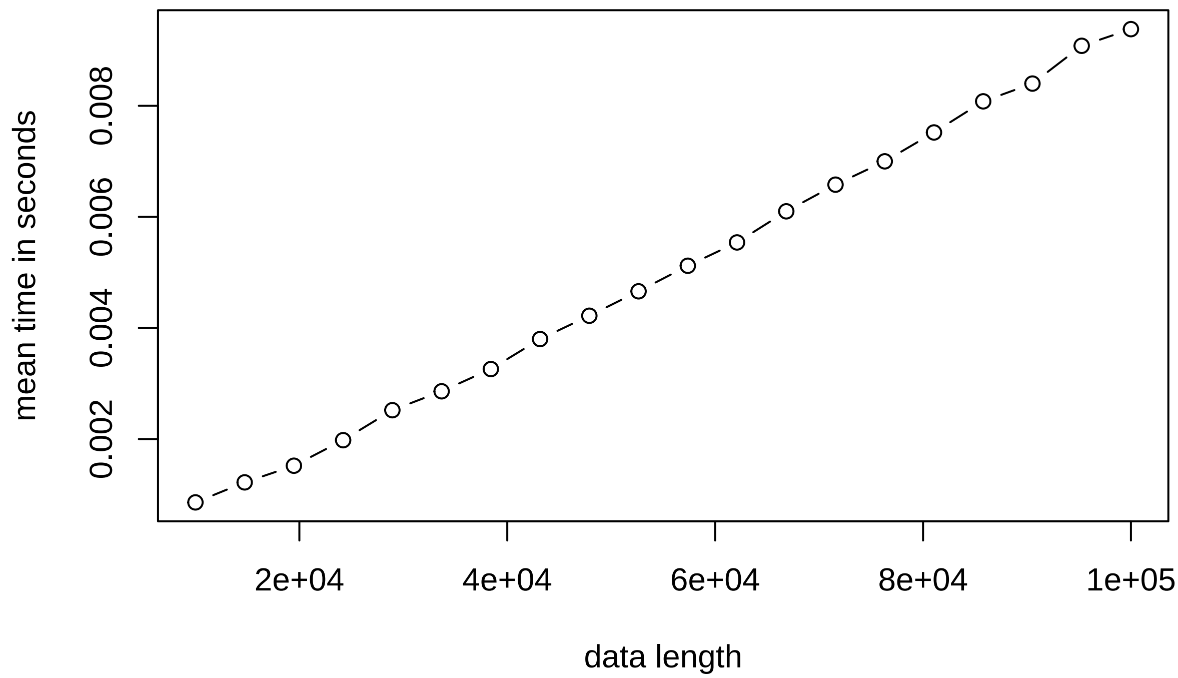
We run `nbRep = 50` times the `heap_sort_Rcpp` algorithm of each value of the `vector_n` vector of length `nbSimus = 20`. We show the plot of the mean running time with respect to data length.

```
nbSimus <- 20
vector_n <- seq(from = 10000, to = 100000, length.out = nbSimus)
nbRep <- 50
res_Heap <- data.frame(matrix(0, nbSimus, nbRep + 1))
colnames(res_Heap) <- c("n", paste0("Rep",1:nbRep))

j <- 1
for(i in vector_n)
```

```r
{
  res_Heap[j,] <- c(i, replicate(nbRep, one.simu(i, func = "heap_sort_Rcpp")))
  #print(j)
  j <- j + 1
}

res <- rowMeans(res_Heap[,-1])
plot(vector_n, res, type = 'b', xlab = "data length", ylab = "mean time in seconds")
```

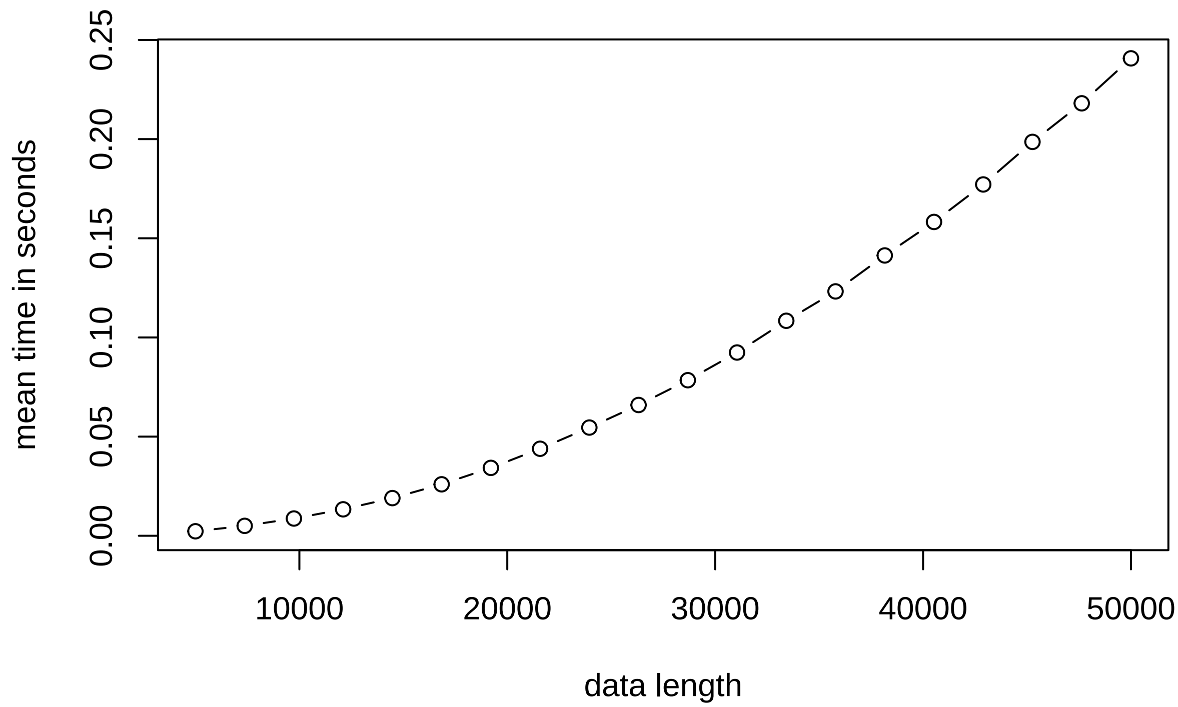


Same strategy but with the `insertion_sort_Rcpp` algorithm. We get the power in complexity model $O(n^r)$ by fitting a linear model in log scale. The slope coefficient r is very close to 2 as expected.

```r
nbSimus <- 20
vector_n <- seq(from = 5000, to = 50000, length.out = nbSimus)
nbRep <- 50
res_Insertion <- data.frame(matrix(0, nbSimus, nbRep + 1))
colnames(res_Insertion) <- c("n", paste0("Rep",1:nbRep))

j <- 1
for(i in vector_n)
{
  res_Insertion[j,] <- c(i, replicate(nbRep, one.simu(i, func = "insertion_sort_Rcpp")))
  #print(j)
  j <- j + 1
}

res <- rowMeans(res_Insertion[,-1])
plot(vector_n, res, type = 'b', xlab = "data length", ylab = "mean time in seconds")
```




```r
lm(log(res) ~ log(vector_n))
```

```
##
## Call:
## lm(formula = log(res) ~ log(vector_n))
##
## Coefficients:
##   (Intercept)  log(vector_n)
##       -23.417          2.033
```