Analyse de deux algorithmes de tri par comparaison

M2 Data Science Algorithmique

Vincent Runge

jeudi 27 mars 2025

Table des matières

---

1 Description du problème et objectif

Nous étudions dans ce document le problème très classique du tri des éléments d'un vecteur. Ce problème consiste à trier par ordre croissant les éléments d'un vecteur initialement non-trié.

Il est intéressant de remarquer que de nombreuses méthodes algorithmiques répondent à ce problème. Elles se distinguent par leur temps d'exécution et par la mémoire nécessaire à résoudre la tâche. La question centrale est ici celle du temps d'exécution.

La page wikipédia du tri rassemble de nombreux algorithmes de tri avec leurs avantages et inconvénients respectifs. La complexité du problème de tri (par comparaison) est de $O(n \log n)$ pour un vecteur de longueur n. Cela signifie que, théoriquement, aucun algorithme ne pourra jamais être plus rapide que cette borne asymptotique. Le *radix sort* et quelques autres algorithmes sont en temps $O(n)$ mais demandent des hypothèses supplémentaires sur les données (ce n'est plus un tri par comparaison).

Dans ce document, nous concentrons notre attention sur deux algorithmes de tri :

1) le tri par insertion, de complexité $O(n^2)$
2) le tri par tas (*heap sort*), de complexité $O(n\log(n))$. Des détails sur le tri par tas sont donnés dans le lien, en particulier les animations donnent une bonne idée du fonctionnement d'un tas.

Nous avons donc ici deux méthodes l'une naïve, l'autre plus évoluée. Nos objectifs sont alors :

a) d'implémenter ces algorithmes en R et en C++ et évaluer le gain de temps ;
b) de confirmer les complexités théoriques trouvées sur le papier (ce n'est pas fait dans ce document mais a été fait dans le cours) par des simulations intensives.

À noter que le (b) se termine toujours par l'évaluation d'une pente sur une régression linéaire en échelle log-log. Cela donne une évaluation de la valeur x dans la complexité de type $O(n^x)$ ou $O(n^x \log^y(n))$.

Nous allons ajouter une étape en plus. En effet, l'évaluation de la complexité se fait sur des données simulées, issues d'une certaine distribution de probabilité. Nous souhaitons étudier le temps d'exécution dans un cas plus favorable à l'algorithme d'insertion.

---

2 Un premier exemple

Le package se télécharge ainsi :

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

et ses fonctions sont rendues disponibles sur Rstudio ainsi :

```
library(M2algorithmique)
```

On simule un petit exemple d'un vecteur `v` de taille `100`

```
n <- 100
v <- sample(n)
```

On teste les 4 algorithmes implémentés avec des noms explicites :

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

Cela donne :

```
v
```

```
##   [1]  35  57  87  14  50  51  61  44  86   1  79  72  91  83  73  48  19   2
##  [19]  36  54   5  93  99  16  65  90  27   7  37  18  97  10  21  13  38  26
##  [37]  84  15  31  30  74  80  55  25  71  41  29  12  49  56  63  40  45  62
##  [55]  81  60   3  53   4  89  67  58  11  32 100  69  33  82  94  20  52  92
##  [73]  42  23  22  47  85  28  66  96   6  24  34  95  70  76  68  88  46   9
##  [91]  59  39  77  75  17  78   8  64  43  98
```

```
insertion_sort(v)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
heap_sort(v)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
insertion_sort_Rcpp(v)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
heap_sort_Rcpp(v)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

---

3 Comparaison R avec C++

On va faire des comparaisons pour les deux types d'algorithme en R et C++ pour quantifier leur différence de performance.

La fonction `one.simu.time` retourne le temps recherché, et `one.simu` sera utilisé par `microbenchmark`

```
one.simu.time <- 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)
}

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

3.1 Un essai

Sur un exemple, on obtient :

```
n <- 10000
one.simu.time(n, func = "insertion_sort")
```

```
## [1] 1.815
```

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

```
## [1] 0.542
```

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

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

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

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

3.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste :

```
nbSimus <- 10

time1 <- rep(0, nbSimus); time2 <- rep(0, nbSimus);
time3 <- rep(0, nbSimus); time4 <- rep(0, nbSimus)

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

Gain C++ versus R

```
mean(time1)/mean(time2)
```

```
## [1] 208.1494
```

```
mean(time3)/mean(time4)
```

```
## [1] 763.1429
```

Gain tas versus insertion

```
mean(time1)/mean(time3)
```

```
## [1] 3.389929
```

```
mean(time2)/mean(time4)
```

```
## [1] 12.42857
```

On recommence avec `n = 20000` seulement pour le gain avec C++ pour le tas

```
n <- 20000
nbSimus <- 10
time3 <- rep(0, nbSimus); time4 <- rep(0, nbSimus)
for(i in 1:nbSimus){time3[i] <- one.simu.time(n, func = "heap_sort")}
for(i in 1:nbSimus){time4[i] <- one.simu.time(n, func = "heap_sort_Rcpp")}
median(time3)/median(time4)
```

```
## [1] 1966
```

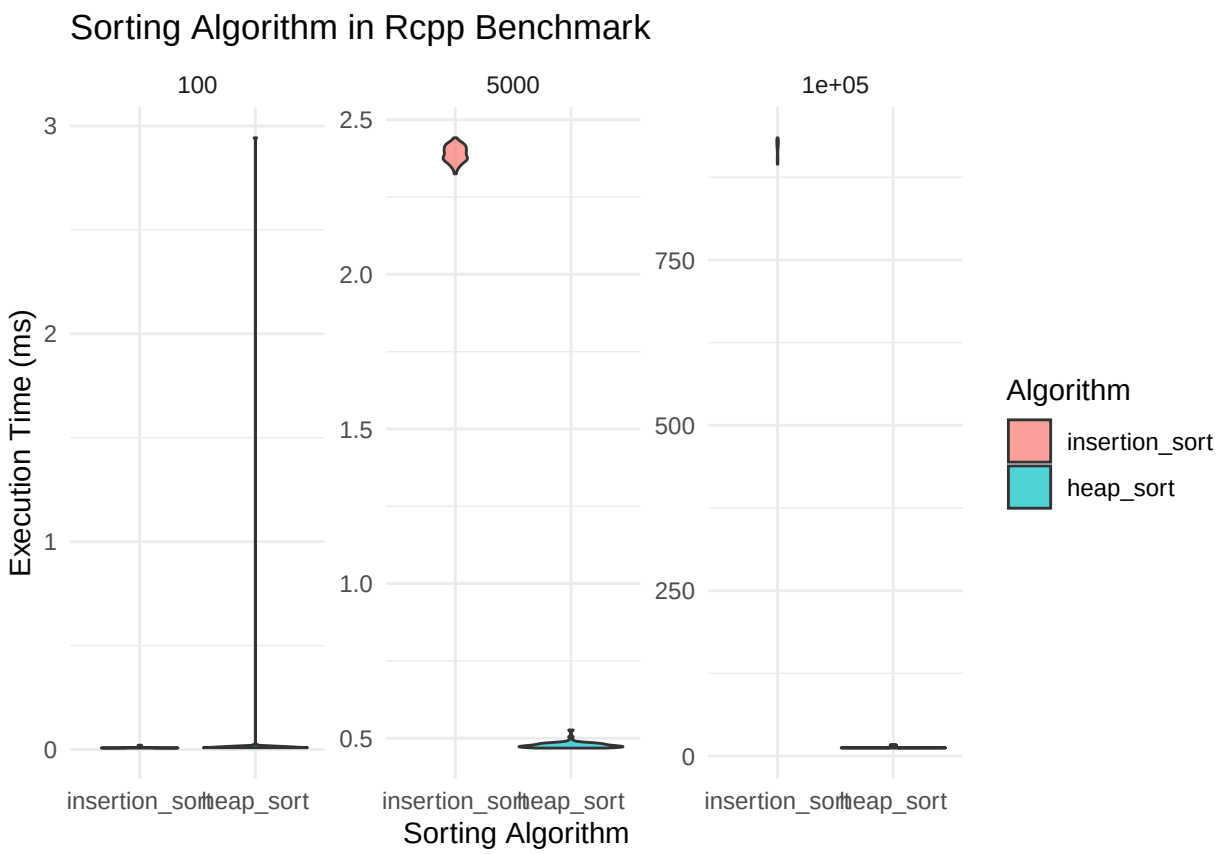
Conclusion :

- pour une taille de `10000` la gain avec C++ atteint un facteur 500 entre le tas C++ et le tas R. Ce gain semble augmenter avec la taille.
- Sans surprise, le tri par tas est plus rapide que le tri par insertion.

3.3 Simulations avec `microbenchmark`

Vous avez besoin des packages `microbenchmark` et `ggplot2` pour exécuter les simulations et afficher les résultats (sous forme de diagrammes en violon). Nous comparons `insertion_sort_Rcpp` avec `heap_sort_Rcpp` pour des tailles de données `n = 1000` et `n = 10000`.

```
library(microbenchmark)
library(ggplot2)
```



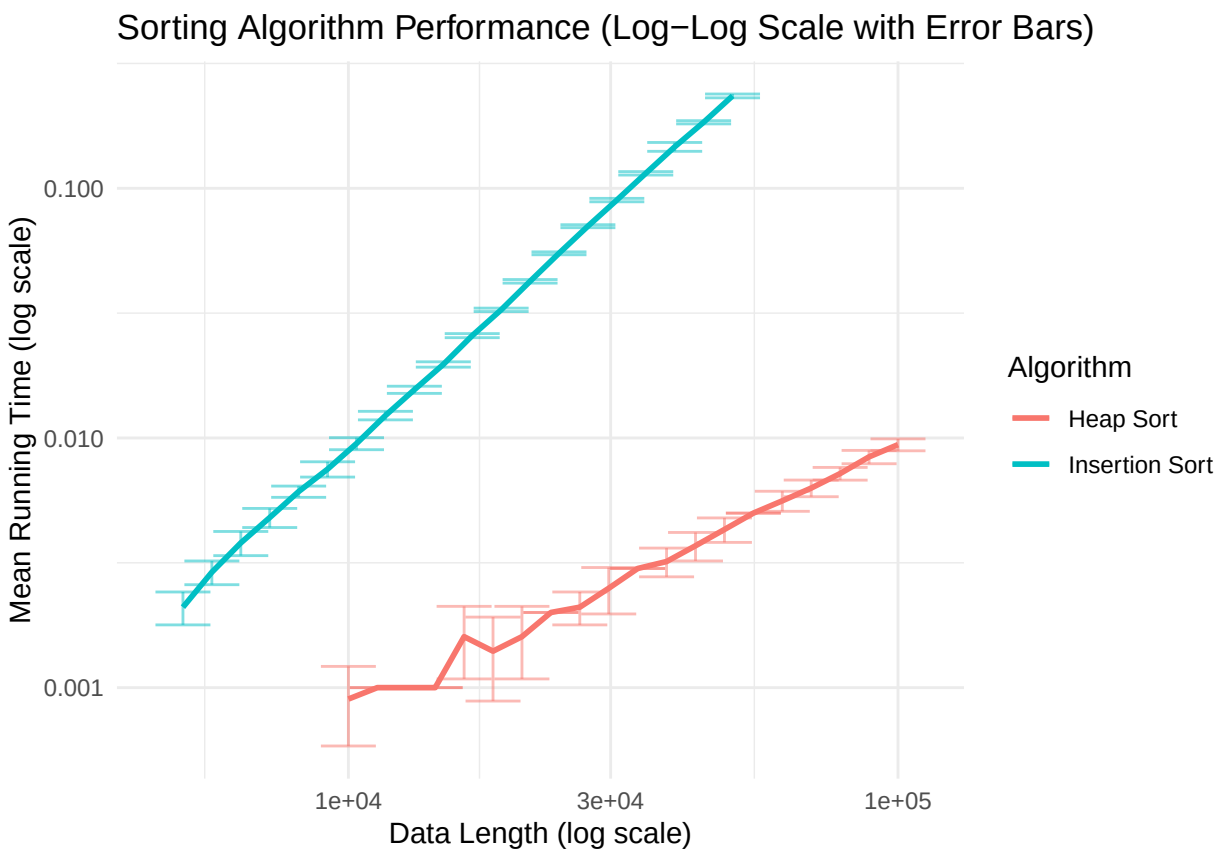

```
## # A tibble: 6 x 8
##        n expr            min_time q1_time median_time mean_time q3_time max_time
##    <dbl> <fct>              <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1    100 insertion_sort   0.00709 7.54e-3     0.00789   0.00846 8.76e-3   0.0207
## 2    100 heap_sort        0.00894 9.61e-3     0.00994   0.0692  1.05e-2   2.94
## 3   5000 insertion_sort   2.32    2.37e+0     2.39      2.39    2.41e+0   2.44
## 4   5000 heap_sort        0.468   4.71e-1     0.475     0.477   4.82e-1   0.527
## 5 100000 insertion_sort 895.      9.08e+2   924.      919.      9.30e+2 935.
## 6 100000 heap_sort       12.0     1.22e+1    12.3      12.4     1.23e+1  16.7
```

---

4 Evaluation de la complexité

Les vecteurs de longueurs `vector_n_insertion` et `vector_n_heap` (`n` dans les dataframes) sont choisis sur l'echelle logarithmique afin d'avoir un pas constant sur l'échelle logarithmique en abscisse pour la régression.

On réalise 10 répétitions pour chaque valeur de `n` et pour chaque algorithme. Les barres d'erreur sont placées en "mean +/- sd".




```
res_Heap
```

```
##         n mean_time      sd_time
## 1   10000    0.0009 3.162278e-04
## 2   11288    0.0010 4.493867e-15
## 3   12743    0.0010 7.338454e-15
## 4   14384    0.0010 4.493867e-15
## 5   16238    0.0016 5.163978e-04
## 6   18330    0.0014 5.163978e-04
## 7   20691    0.0016 5.163978e-04
## 8   23357    0.0020 6.864495e-15
## 9   26367    0.0021 3.162278e-04
## 10  29764    0.0025 5.270463e-04
## 11  33598    0.0030 0.000000e+00
## 12  37927    0.0032 4.216370e-04
## 13  42813    0.0037 4.830459e-04
## 14  48329    0.0043 4.830459e-04
## 15  54556    0.0050 7.338454e-15
## 16  61585    0.0056 5.163978e-04
## 17  69519    0.0063 4.830459e-04
## 18  78476    0.0072 4.216370e-04
## 19  88587    0.0084 5.163978e-04
## 20 100000    0.0094 5.163978e-04
```

```
res_Insertion
```

```
##        n mean_time      sd_time
## 1   5000    0.0021 0.0003162278
## 2   5644    0.0029 0.0003162278
## 3   6371    0.0038 0.0004216370
## 4   7192    0.0048 0.0004216370
## 5   8119    0.0061 0.0003162278
## 6   9165    0.0075 0.0005270463
## 7  10346    0.0095 0.0005270463
## 8  11679    0.0123 0.0004830459
## 9  13183    0.0156 0.0005163978
## 10 14882    0.0197 0.0004830459
## 11 16799    0.0257 0.0004830459
## 12 18963    0.0326 0.0005163978
## 13 21407    0.0424 0.0006992059
## 14 24165    0.0549 0.0007378648
## 15 27278    0.0705 0.0009718253
## 16 30792    0.0897 0.0014181365
## 17 34760    0.1148 0.0018135294
## 18 39238    0.1466 0.0059665736
## 19 44293    0.1838 0.0027406406
## 20 50000    0.2344 0.0041686662
```

On vérifie la valeur du coefficient directeur pour les deux méthodes :

```
##
## Call:
## lm(formula = log(res_Insertion$mean_time) ~ log(res_Insertion$n))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.056330 -0.013396 -0.004171  0.013740  0.045334
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)          -23.381572   0.075886  -308.1   <2e-16 ***
## log(res_Insertion$n)   2.027908   0.007828   259.0   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02447 on 18 degrees of freedom
## Multiple R-squared:  0.9997, Adjusted R-squared:  0.9997
## F-statistic: 6.711e+04 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Estimated exponent: 2.027908
```

```
##
## Call:
## lm(formula = log(res_Heap$mean_time) ~ log(res_Heap$n))
##
## Residuals:
```

```
##       Min        1Q    Median        3Q       Max
## -0.167735 -0.034582  0.005972  0.028454  0.173666
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)     -16.89548    0.25406  -66.50   <2e-16 ***
## log(res_Heap$n)   1.06075    0.02446   43.36   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07645 on 18 degrees of freedom
## Multiple R-squared:  0.9905, Adjusted R-squared:   0.99
## F-statistic:  1880 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Estimated exponent: 1.060748
```

Les coefficients dfirecteurs trouvés sont bien ceux que l'on attendait. La valeur 2 pour l'insertion et 1 pour le tas.

---

5 Cas particulier des données presque triées

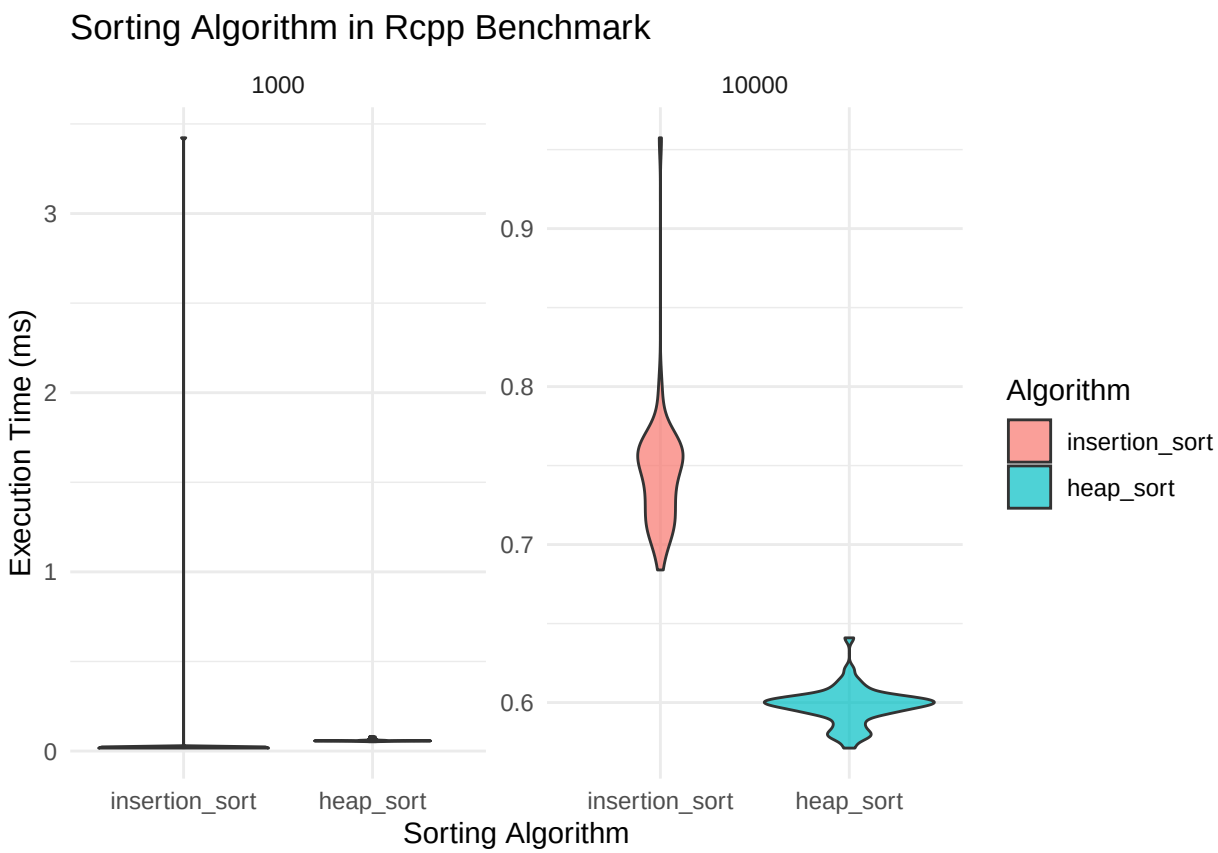
On considère des données triées avec 5% de valeurs échangées au hasard.

Sur un exemple cela donne :

```
v <- 1:100
n_swap <- floor(0.05 * length(v))
swap_indices <- sample(length(v), n_swap)
v[swap_indices] <- sample(v[swap_indices])
v
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  79
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  97  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  41  80  18  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  81  98  99 100
```

```
one.simu2 <- function(n, func)
{
  v <- 1:n
  n_swap <- floor(0.05 * length(v))
  swap_indices <- sample(length(v), n_swap)
  v[swap_indices] <- sample(v[swap_indices])
  if(func == "insertion_sort"){insertion_sort(v)}
  if(func == "heap_sort"){heap_sort(v)}
  if(func == "insertion_sort_Rcpp"){insertion_sort_Rcpp(v)}
  if(func == "heap_sort_Rcpp"){heap_sort_Rcpp(v)}
}
```




```
## # A tibble: 4 x 8
##       n expr           min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>             <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1  1000 insertion_sort   0.0166  0.0178      0.0186    0.0868  0.0195   3.42
## 2  1000 heap_sort        0.0516  0.0555      0.0568    0.0579  0.0579   0.0795
## 3 10000 insertion_sort   0.684   0.723       0.748     0.745   0.759    0.957
## 4 10000 heap_sort        0.571   0.594       0.599     0.598   0.603    0.641
```

L'algorithme d'insertion est ici plus rapide pour la longueur `1000`. Cela est dû au fait que pour un vecteur déjà trié, l'algorithme d'insertion est linéaire et nous sommes dans un cas proche du linéaire.