Analyse des algorithmes de Maximum Subarray 1D
M2 Data Science Algorithmique

Manal Derghal, Khalil Ounis, Taqwa Ben Romdhane

Lundi 7 avril 2025

Table des matières

---

1 Description du problème et objectif

Le problème du Maximum Subarray 1D consiste à trouver la sous-séquence contiguë d'un tableau numérique dont la somme des éléments est maximale. Ce problème classique en algorithmique a des applications en analyse de données financières, bioinformatique et traitement du signal.

La page Wikipedia du Maximum Subarray présente plusieurs approches algorithmiques pour résoudre ce problème. Nous nous concentrons sur deux méthodes :

1. Algorithme naïf : complexité $O(n^2)$
2. Algorithme de Kadane : complexité optimale $O(n)$

Nos objectifs sont :
a. d'implémenter ces algorithmes en R et C++ et évaluer le gain de temps.
b. de confirmer les complexités théoriques par des simulations intensives.

---

2 Un premier exemple

Le package se télécharge ainsi :

```
devtools::install_github("AMATERASU11/MaximumSubarray")
```

et ses fonctions sont rendues disponibles sur Rstudio ainsi :

```r
library(MaximumSubarray)
```

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

```r
set.seed(123)
v <- sample(-100:100, 100, replace = TRUE)
```

On teste les 4 algorithmes implémentés avec des noms explicites :

- `max_subarray_sum_naive`
- `max_subarray_sum_opt`
- `max_subarray_sum_naive_Rcpp`
- `max_subarray_sum_opt_Rcpp`

Cela donne :

```r
v
```

```
##   [1]  58  78 -87  94  69 -51  17 -58 -87  17  52 -11 -10  96 -10  84  -9  36
##  [19]  -2 -29 -75 -94  69  36  63 -23 -20 -58   2  16 -25  42 -69   8 -94  36
##  [37]  68 -27 -78  54  87 -48  34 -48  54  65 -67 -32 -29 -25 -38  40  -4 -10
##  [55]  52 -63 -80 -60  74 -11 -41 -85  15  -7 -95  99 -15 -15 -62  58  17 -51
##  [73] -67 -97 -88 -32  26  52 -49 -79 -12  59 -76 -66  67  11 -71  39  58  20
##  [91]   9  57 -37  41  98 -34  50  21 -22 -16
```

```r
max_subarray_sum_naive(v)
```

```
## $sum
## [1] 329
##
## $subarray
##  [1]  67  11 -71  39  58  20   9  57 -37  41  98 -34  50  21
```

```r
max_subarray_sum_naive_Rcpp(v)
```

```
## $sum
## [1] 329
##
## $subarray
##  [1]  67  11 -71  39  58  20   9  57 -37  41  98 -34  50  21
```

```r
max_subarray_sum_opt(v)
```

```
## $sum
## [1] 329
##
## $subarray
##  [1]  67  11 -71  39  58  20   9  57 -37  41  98 -34  50  21
```

```r
max_subarray_sum_opt_Rcpp(v)
```

```
## $sum
## [1] 329
##
## $subarray
##  [1]  67  11 -71  39  58  20   9  57 -37  41  98 -34  50  21
```

---

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`, on retourne le temps en ms

```
library(microbenchmark)

one.simu.time <- function(n, func) {

  v <- sample(-100:100, n, replace = TRUE)

  if (func == "Naive1D") {
    t <- microbenchmark(max_subarray_sum_naive(v), times = 1)$time / 1e6
  } else if (func == "Naive1D_cpp") {
    t <- microbenchmark(max_subarray_sum_naive_Rcpp(v), times = 1)$time / 1e6
  } else if (func == "Kadane1D") {
    t <- microbenchmark(max_subarray_sum_opt(v), times = 1)$time / 1e6
  } else if (func == "Kadane1D_cpp") {
    t <- microbenchmark(max_subarray_sum_opt_Rcpp(v), times = 1)$time / 1e6
  } else {
    stop("fonction inconnue")
  }

  return(round(t, 2))
}
```

3.1 Un essai

3.1.1 Temps d'exécution en R

Sur un exemple, on obtient :

```
# Simulation sur une matrice de taille n
n <- 10000

# Exécuter la simulation
res_naive <- one.simu.time(n,"Naive1D")
res_kadane <- one.simu.time(n, "Kadane1D")

# Afficher les résultats
cat("time_naive:", res_naive,"ms\n")
```

```
## time_naive: 2167.17 ms
```

```
cat("time_kadane:", res_kadane, "ms")
```

```
## time_kadane: 0.89 ms
```

3.1.2 Temps d'exécution en C++

sur une matrice de taille 30x50, on obtient les résultats suivants :

```
# Simulation sur une matrice de taille n
n <- 10000
```

```r
res_naive_cpp <- one.simu.time(n,"Naive1D_cpp")
res_Kadane_cpp <- one.simu.time(n,"Kadane1D_cpp")

# Afficher les résultats
cat("time_naive_cpp:" ,res_naive_cpp,"ms\n")
```

```
## time_naive_cpp: 24.98 ms
```

```r
cat("time_kadane_cpp:",res_Kadane_cpp, "ms")
```

```
## time_kadane_cpp: 0.07 ms
```

3.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste :

```r
nbSimus <- 10

time_naive <- rep(0, nbSimus); time_naive_cpp <- rep(0, nbSimus);
time_kadane <- rep(0, nbSimus); time_kadane_cpp <- rep(0, nbSimus)

for(i in 1:nbSimus){time_naive[i] <- one.simu.time(n, func = "Naive1D")}
for(i in 1:nbSimus){time_naive_cpp[i] <- one.simu.time(n, func = "Naive1D_cpp")}
for(i in 1:nbSimus){time_kadane[i] <- one.simu.time(n, func = "Kadane1D")}
for(i in 1:nbSimus){time_kadane_cpp[i] <- one.simu.time(n, func = "Kadane1D_cpp")}
```

3.2.1 Gain R versus C++

```r
naive_speedup_cpp <- mean(time_naive) / mean(time_naive_cpp)
kadane_speedup_cpp <- mean(time_kadane) / mean(time_kadane_cpp)
cat("le gain R vs cpp pour naif:", round(naive_speedup_cpp,2),"ms\n")
```

```
## le gain R vs cpp pour naif: 71.46 ms
```

```r
cat("le gain R vs cpp pour Kadane:", round(kadane_speedup_cpp,2),"ms\n")
```

```
## le gain R vs cpp pour Kadane: 9.88 ms
```

3.2.2 Gain Naif Versus Kadane en R et C++

```r
kadane_vs_naive_R <- mean(time_naive) / mean(time_kadane)
kadane_vs_naive_Rcpp <- mean(time_naive_cpp) / mean(time_kadane_cpp)
cat("le gain naif vs Kadane en R est:",round(kadane_vs_naive_R,2), "ms\n")
```

```
## le gain naif vs Kadane en R est: 2030.39 ms
```

```r
cat("le gain cpp est:",round(kadane_vs_naive_Rcpp,2), "ms\n")
```

```
## le gain cpp est: 280.81 ms
```

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

```r
set.seed(123)
n <- 20000
nbSimus <- 10
time_kadane <- rep(0, nbSimus); time_kadane_cpp <- rep(0, nbSimus)
for(i in 1:nbSimus){time_kadane[i] <- one.simu.time(n, func = "Kadane1D")}
```

```r
for(i in 1:nbSimus){time_kadane_cpp[i] <- one.simu.time(n, func = "Kadane1D_cpp")}
median_kadane_R_vs_Rcpp <- median(time_kadane) / median(time_kadane_cpp)
cat("le gain Kadane en R vs Kadane en C++ est:",round(median_kadane_R_vs_Rcpp,2), "ms\n")
```

```
## le gain Kadane en R vs Kadane en C++ est: 11.86 ms
```

Conclusion :

3.2.3 Performances C++ vs R :

— Naïf : C++ 71× plus rapide

— Kadane : C++ 10× plus rapide → 12× pour n=20k

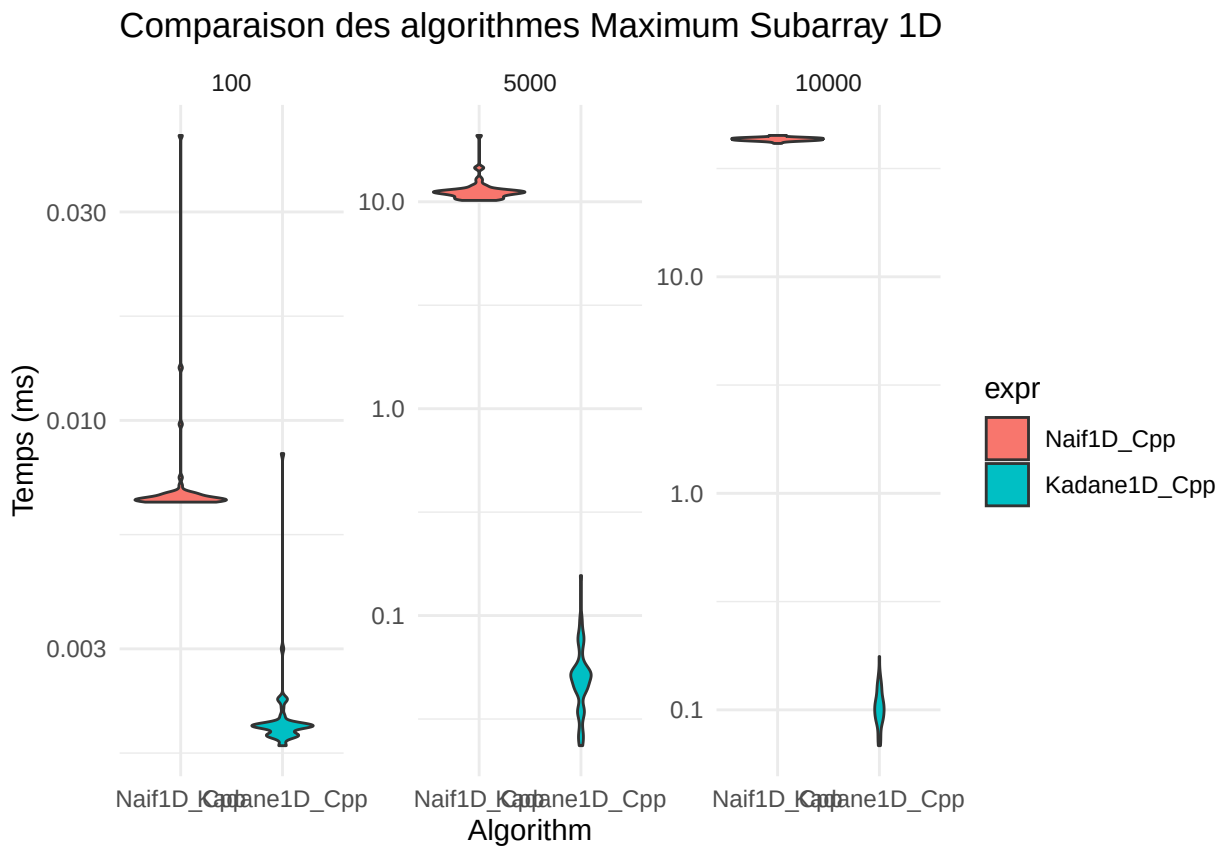
3.2.4 Efficacité algorithmique :

— Kadane 2030× mieux que naïf en R

— Kadane 281× mieux que naïf en C++

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 `naive_Rcpp` avec `opt_Rcpp` pour des tailles de données `n = 1000`, `n = 5000` et `n = 10000`.




```
## # A tibble: 6 x 8
##       n expr         min_time  q1_time median_time mean_time  q3_time max_time
```

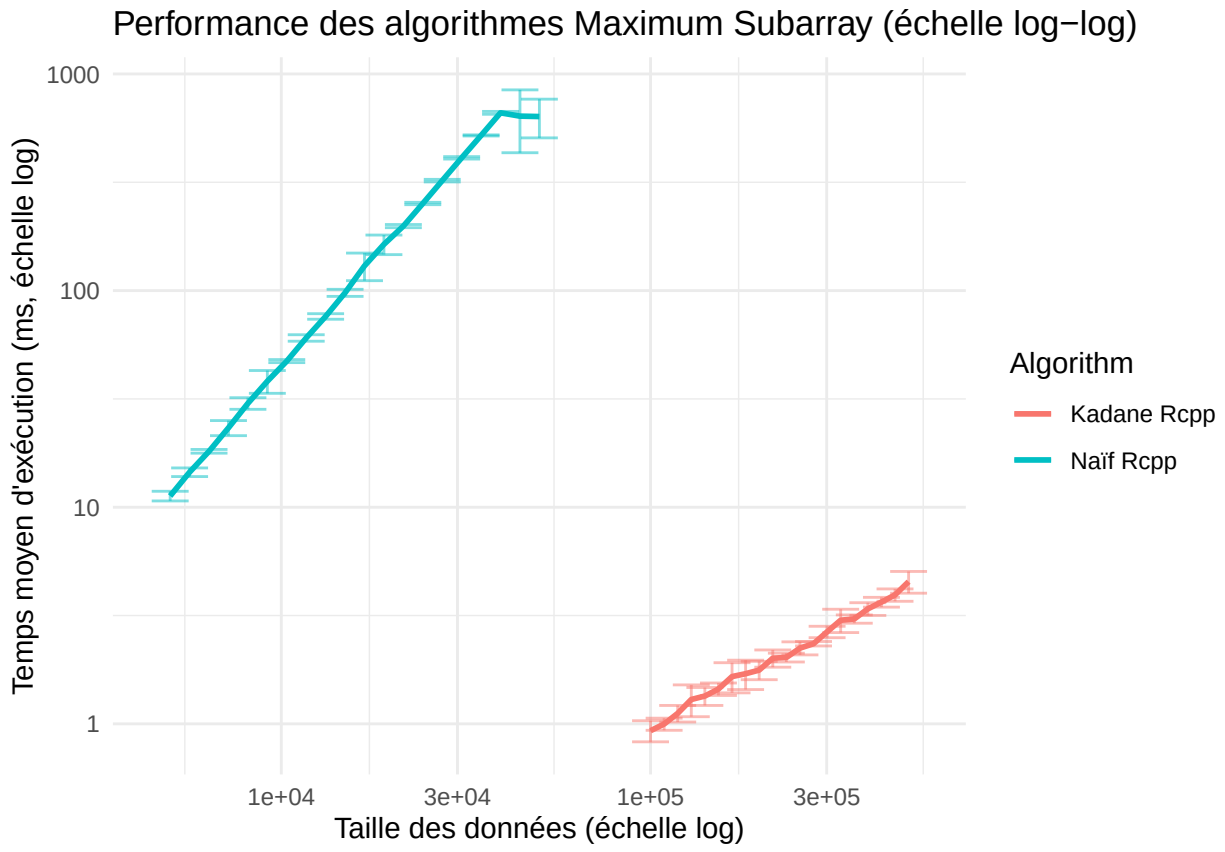
```
##   <dbl> <fct>           <dbl>    <dbl>       <dbl>     <dbl>    <dbl>    <dbl>
## 1   100 Naif1D_Cpp    0.00650  0.00653     0.00660   0.00761  0.00678  0.0449
## 2   100 Kadane1D_Cpp  0.00180  0.00190     0.00200   0.00214  0.00200  0.00840
## 3  5000 Naif1D_Cpp   10.2     10.7        11.1      11.4     11.4     20.9
## 4  5000 Kadane1D_Cpp  0.0235   0.0432      0.0504    0.0527   0.0549   0.156
## 5 10000 Naif1D_Cpp   41.3     42.8        43.2      43.2     43.7     45.0
## 6 10000 Kadane1D_Cpp  0.0683   0.0910      0.102     0.104    0.114    0.176
```

---

4 Evaluation de la complexité

Les vecteurs de longueurs `vector_n_naive` et `vector_n_kadane` (`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".




```
# Affichage des résultats
cat("Les résultats pour la solution naïve:\n")
```

```
## Les résultats pour la solution naïve:
```

```
res_Naive
```

```
##        n mean_time     sd_time
## 1   5000    11.270   0.5793291
## 2   5644    14.518   0.6674129
```

```
## 3   6371    18.123   0.3564345
## 4   7192    23.257   1.8599286
## 5   8119    30.168   1.8597539
## 6   9165    38.156   4.6008651
## 7  10346    47.265   0.7614496
## 8  11679    60.469   2.0857530
## 9  13183    75.990   2.2458455
## 10 14882    97.693   3.6680060
## 11 16799   130.050  18.9493694
## 12 18963   163.516  16.9388588
## 13 21407   198.468   3.3364112
## 14 24165   252.345   3.1644949
## 15 27278   321.789   4.4165206
## 16 30792   410.089   5.2821954
## 17 34760   520.307   3.1626292
## 18 39238   661.620  10.1094609
## 19 44293   638.637 205.7220376
## 20 50000   635.950 129.0299659
```

```
cat("Les résultats pour la solution optimale:\n")
```

```
## Les résultats pour la solution optimale:
```

```
res_Kadane
```

```
##         n mean_time    sd_time
## 1  100000     0.930 0.10392305
## 2  108840     0.998 0.06425643
## 3  118461     1.118 0.09807027
## 4  128933     1.296 0.21700998
## 5  140330     1.344 0.12790622
## 6  152735     1.448 0.09600926
## 7  166237     1.653 0.26268063
## 8  180932     1.704 0.26412118
## 9  196926     1.771 0.17175241
## 10 214334     2.010 0.18342422
## 11 233281     2.025 0.09454570
## 12 253902     2.236 0.15471838
## 13 276347     2.344 0.05541761
## 14 300776     2.660 0.16096929
## 15 327364     3.010 0.37151342
## 16 356302     3.049 0.13395273
## 17 387799     3.395 0.22979459
## 18 422079     3.648 0.18960778
## 19 459391     3.938 0.25827634
## 20 500000     4.532 0.52119094
```

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

```
##
## Call:
## lm(formula = log(res_Naive$mean_time) ~ log(res_Naive$n))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.37749 -0.03897  0.00767  0.08380  0.11841
```

```
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -13.53834    0.35982  -37.62   <2e-16 ***
## log(res_Naive$n)   1.88275    0.03712   50.72   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.116 on 18 degrees of freedom
## Multiple R-squared:  0.9931, Adjusted R-squared:  0.9927
## F-statistic:  2573 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 1.882751
```

```
##
## Call:
## lm(formula = log(res_Kadane$mean_time) ~ log(res_Kadane$n))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.056489 -0.026127 -0.008281  0.020889  0.067536
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -10.7594     0.2071  -51.95   <2e-16 ***
## log(res_Kadane$n)   0.9312     0.0168   55.43   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0367 on 18 degrees of freedom
## Multiple R-squared:  0.9942, Adjusted R-squared:  0.9939
## F-statistic:  3073 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (Kadane): 0.9312324
```

Les coefficients directeurs trouvés sont bien ceux que l'on attendait. La valeur 2 pour la méthode naïve et 1 pour l'algorithme de Kadane

---

5 Cas particulier des données presques triées

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

Sur un exemple cela donne :

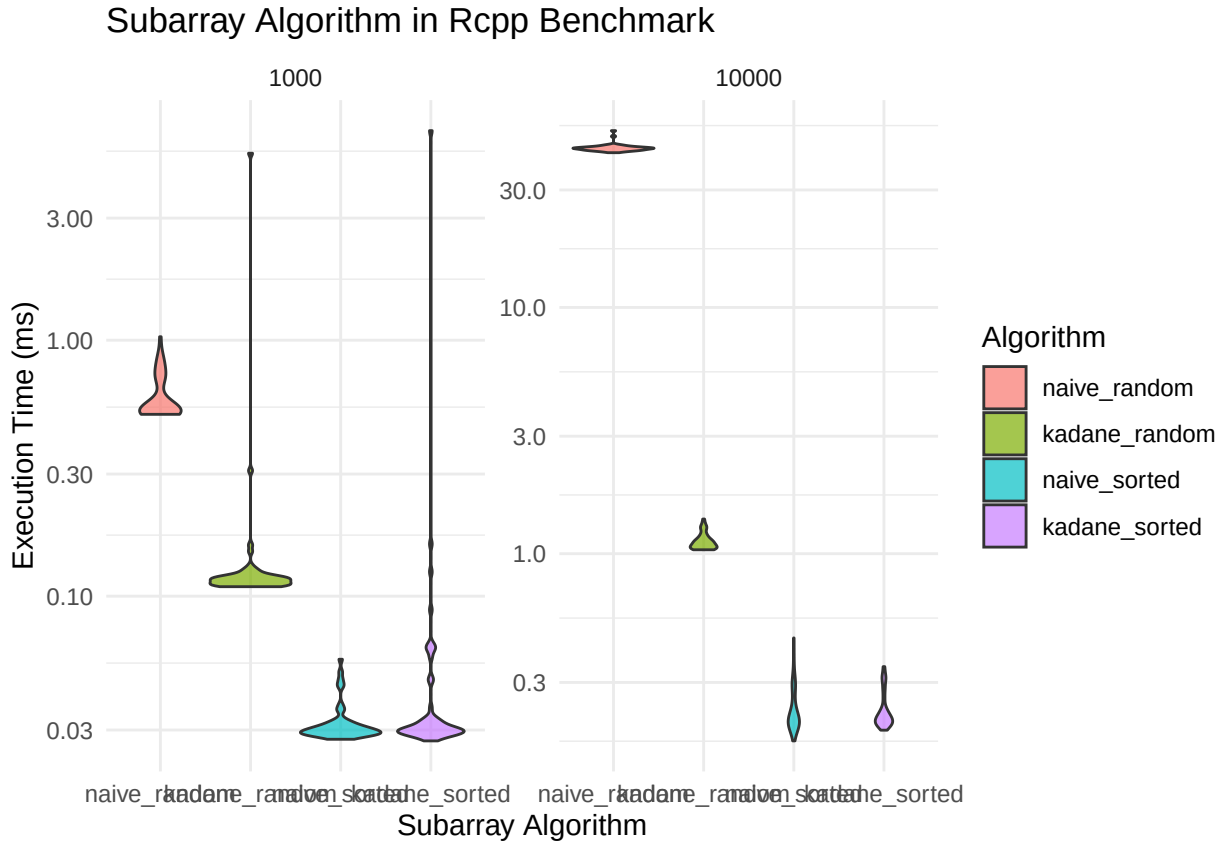
```r
v <- 1:100
n_swap <- floor(0.05 * length(v))
swap_indices <- sample(length(v), n_swap)
v[swap_indices] <- sample(v[swap_indices])
v
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  30
##  [19]  19  20  21  22  23  24  25  26  27  28  29  34  31  32  33  94  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  18  95  96  97  98  99 100
```

```r
# Fonctions de simulation
one.simu <- function(n, func) {
  v <- sample(-100:100, n, replace = TRUE)
  if (func == "naive_Rcpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "kadane_Rcpp") return(max_subarray_sum_opt_Rcpp(v))
}

one.simu2 <- function(n, func) {
  v <- 1:n
  n_swap <- floor(0.05 * n)
  swap_indices <- sample(n, n_swap)
  v[swap_indices] <- sample(v[swap_indices])
  if (func == "naive_Rcpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "kadane_Rcpp") return(max_subarray_sum_opt_Rcpp(v))
}
```




```
## # A tibble: 8 x 10
##       n expr       min_time q1_time median_time mean_time q3_time max_time type
##   <dbl> <fct>         <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl> <chr>
## 1  1000 naive_ran~   0.513   0.518       0.542     0.604   0.706    1.03   rand~
## 2  1000 kadane_ra~   0.109   0.111       0.116     0.226   0.120    5.38   rand~
## 3  1000 naive_sor~   0.0276  0.0290      0.0300    0.0321  0.0318   0.0567 sort~
## 4  1000 kadane_so~   0.0272  0.0295      0.0303    0.171   0.0331   6.59   sort~
## 5 10000 naive_ran~  42.5    43.7        44.3      44.4    44.7     52.2    rand~
## 6 10000 kadane_ra~   1.04    1.06        1.10      1.12    1.16     1.38   rand~
```

```
## 7 10000 naive_sor~   0.174   0.203       0.213     0.238   0.260    0.455  sort~
## 8 10000 kadane_so~   0.192   0.206       0.217     0.234   0.236    0.348  sort~
## # i 1 more variable: algo <chr>
```

Pour n = 1000, le temps d'exécution est plus rapide que pour n = 10000. Kadane est toujours plus rapide que Naïf, avec un écart plus important à n = 10000. Lorsque les tableaux sont triés, Naïf et Kadane sont beaucoup plus rapides, avec un écart réduit entre les deux.