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 :

```
library(MaximumSubarray)
```

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

```
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 :

```
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
```

```
max_subarray_sum_naive(v)
```

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

```
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
```

```
max_subarray_sum_opt(v)
```

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

```
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

```r
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 :

```r
# 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: 2717.72 ms
```

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

```
## time_kadane: 0.95 ms
```

3.1.2 Temps d'exécution en C++

sur un vecteur de taille 10000 on obtient les résultats suivants :

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

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: 26.93 ms
```

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

```
## time_kadane_cpp: 0.1 ms
```

3.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste :

```
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++

```
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: 81.66 ms
```

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

```
## le gain R vs cpp pour Kadane: 9.4 ms
```

3.2.2 Gain Naif Versus Kadane en R et C++

```
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: 2490.92 ms
```

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

```
## le gain cpp est: 286.8 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")}
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: 11.69 ms
```

Conclusion :

3.2.3 Performances C++ vs R :

— Naïf : C++ 82× plus rapide

— Kadane : C++ 9× plus rapide → 12× pour n=20k

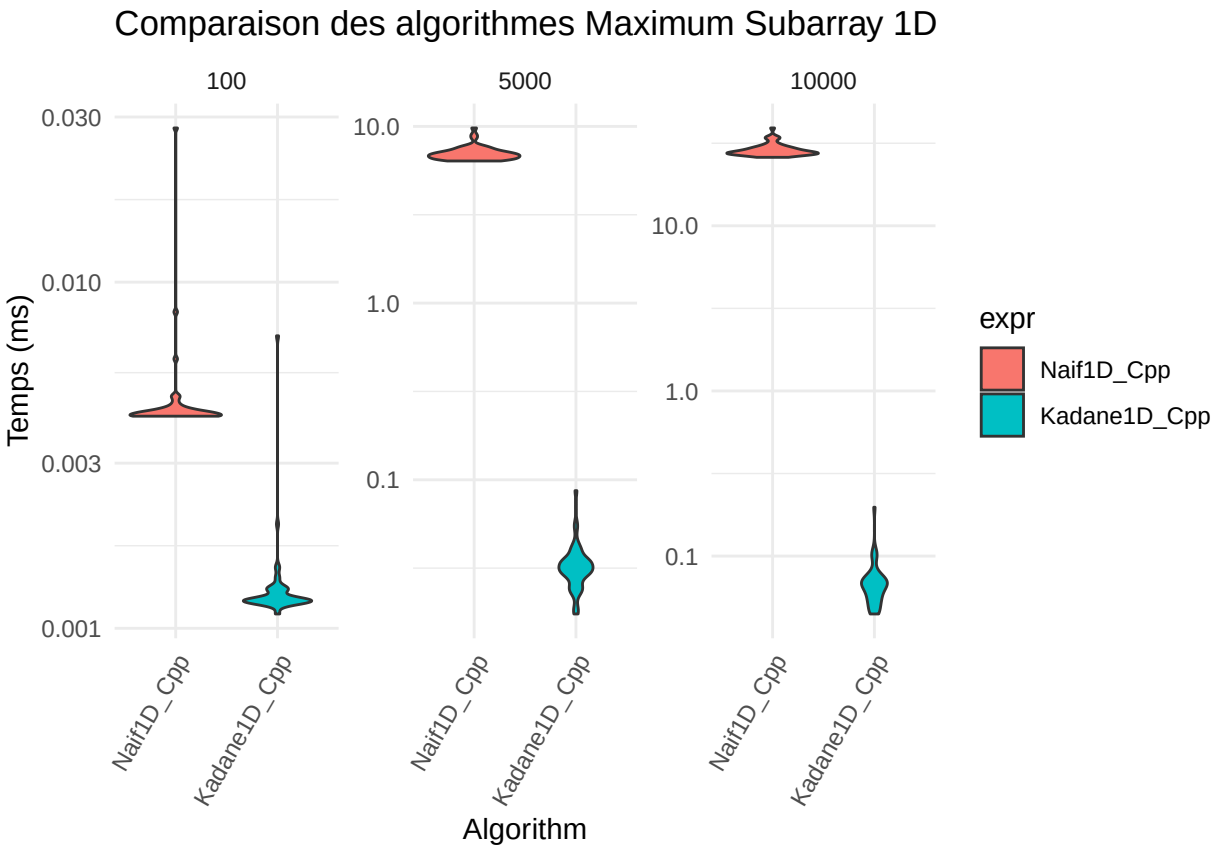
3.2.4 Efficacité algorithmique :

— Kadane 2491× mieux que naïf en R

— Kadane 287× 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`.



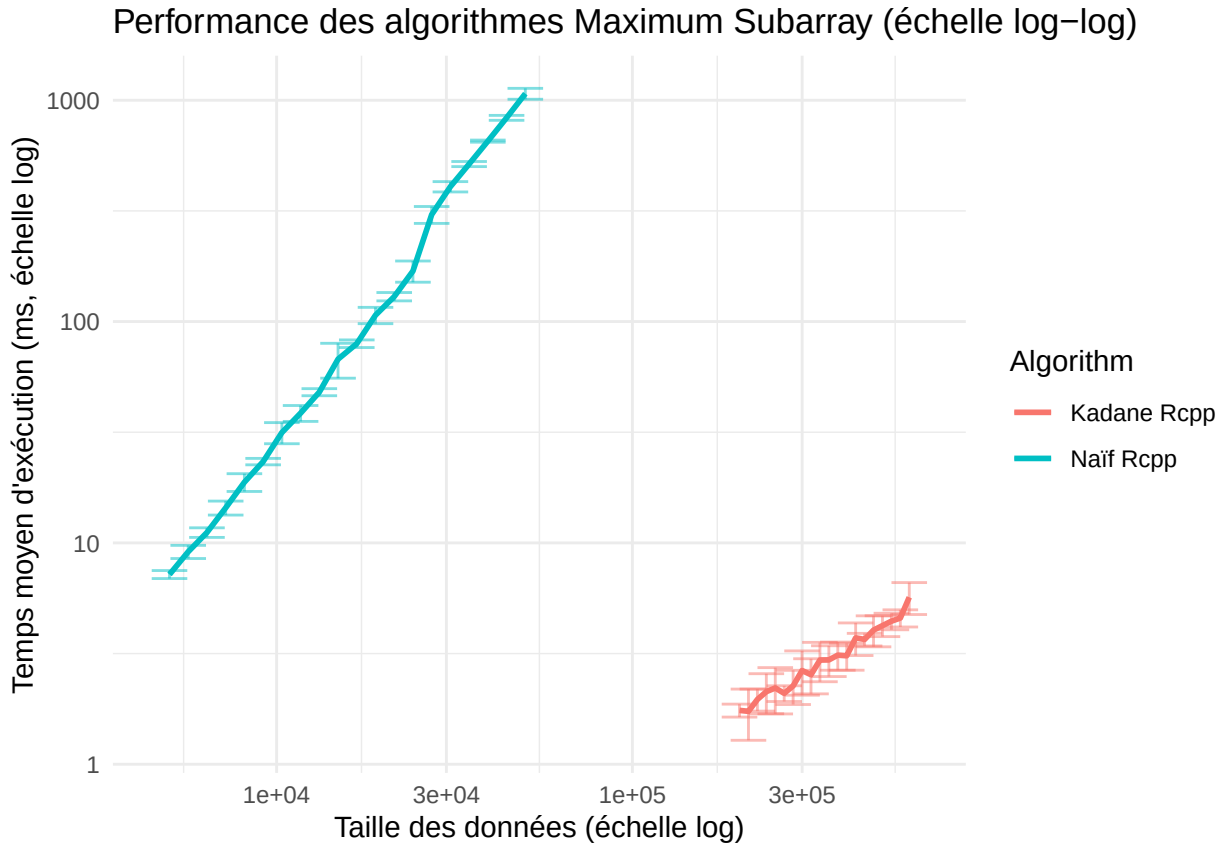

```
## # A tibble: 6 x 8
##       n expr         min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>           <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1   100 Naif1D_Cpp     0.0041  0.0041      0.0042   0.00480  0.0043   0.0279
## 2   100 Kadane1D_Cpp   0.0011  0.0012      0.0012   0.00137  0.0013   0.007
## 3  5000 Naif1D_Cpp     6.37    6.61        6.88     7.05     7.22     9.80
## 4  5000 Kadane1D_Cpp   0.0174  0.0282      0.0314   0.0330   0.0350   0.0868
## 5 10000 Naif1D_Cpp    26.0    27.3        28.0     28.9     29.7     39.2
## 6 10000 Kadane1D_Cpp   0.0446  0.0563      0.0656   0.0691   0.0729   0.198
```

---

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     7.200  0.2991841
## 2   5644     9.123  0.6236817
## 3   6371    11.145  0.5563622
## 4   7192    14.405  1.0457028
## 5   8119    18.808  1.7312603
## 6   9165    23.307  0.7777610
## 7  10346    31.505  3.4447424
## 8  11679    38.573  3.1766336
## 9  13183    47.995  1.7669072
## 10 14882    67.621 12.1391876
## 11 16799    79.465  3.1507398
## 12 18963   106.873  9.0534046
## 13 21407   129.639  5.6214538
## 14 24165   169.168 18.5884318
## 15 27278   304.334 26.6861459
## 16 30792   406.982 21.8300582
## 17 34760   514.950 13.7058917
## 18 39238   653.446  7.1864088
## 19 44293   832.811 21.2956383
```

```
## 20 50000  1070.974 60.7800569
```

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

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

```
res_Kadane
```

```
##         n mean_time   sd_time
## 1  200000     1.751 0.1177993
## 2  211905     1.734 0.4513240
## 3  224519     1.962 0.2241428
## 4  237884     2.128 0.4366489
## 5  252044     2.210 0.5221749
## 6  267047     2.091 0.1706491
## 7  282944     2.259 0.3977841
## 8  299786     2.651 0.6001009
## 9  317631     2.538 0.4588827
## 10 336539     2.956 0.5970520
## 11 356571     2.960 0.4703190
## 12 377797     3.111 0.4523875
## 13 400285     3.091 0.4317780
## 14 424113     3.724 0.6257476
## 15 449359     3.663 0.2389351
## 16 476107     4.037 0.6447403
## 17 504448     4.229 0.4569087
## 18 534476     4.429 0.3795158
## 19 566291     4.577 0.4072687
## 20 600000     5.682 0.9350793
```

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
## -0.19432 -0.07602  0.02344  0.08923  0.14727
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -17.00864    0.34524  -49.27   <2e-16 ***
## log(res_Naive$n)   2.21288    0.03562   62.13   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1113 on 18 degrees of freedom
## Multiple R-squared:  0.9954, Adjusted R-squared:  0.9951
## F-statistic:  3860 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 2.21288
```

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

```
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.086471 -0.033685 -0.008933  0.039289  0.119393
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -11.62733    0.45467  -25.57 1.33e-15 ***
## log(res_Kadane$n)   0.99553    0.03563   27.94 2.81e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05313 on 18 degrees of freedom
## Multiple R-squared:  0.9775, Adjusted R-squared:  0.9762
## F-statistic: 780.6 on 1 and 18 DF,  p-value: 2.815e-16
```

```
## Exposant estimé (Kadane): 0.9955319
```

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

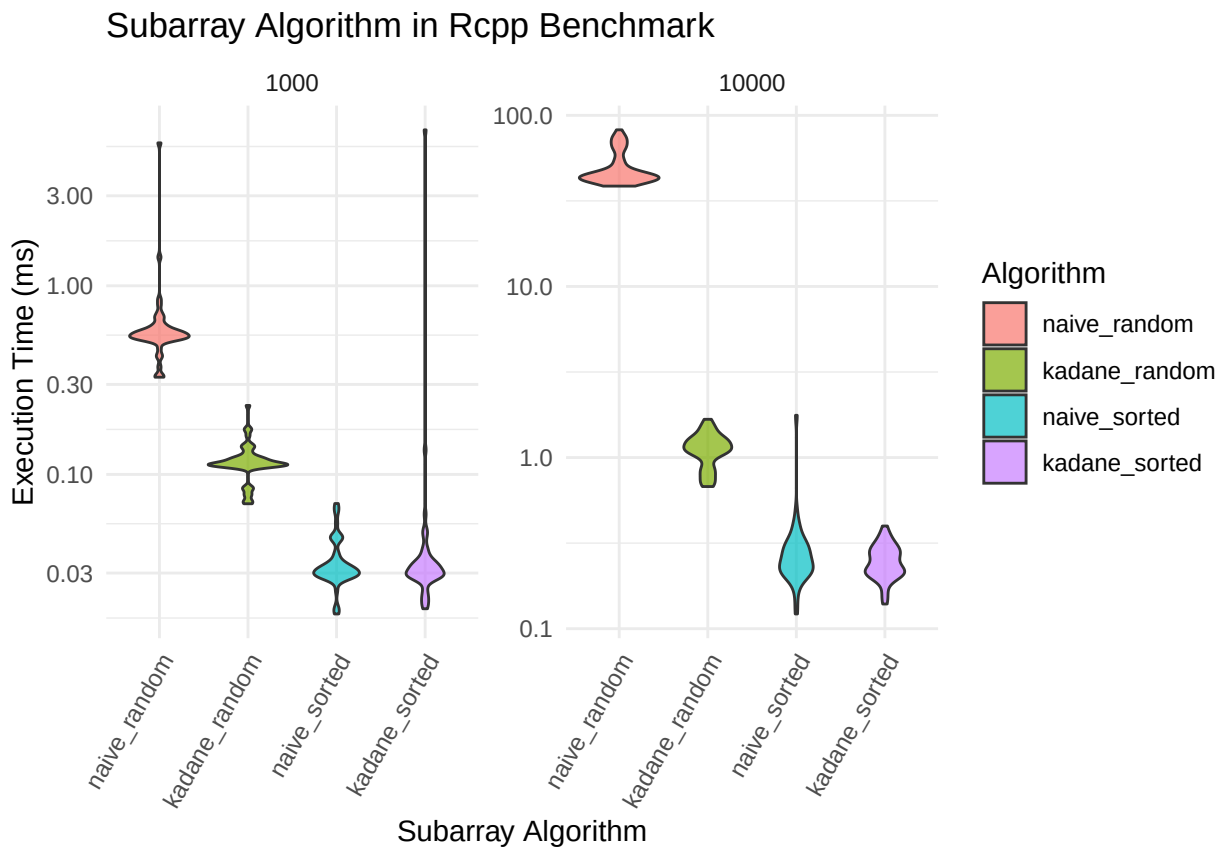
On considère des données triées avec 5% de valeurs échangées au hasard.

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  59  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  17  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  87  86  85  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```r
# Fonctions de simulation
one.simu <- function(n, func) {
  v <- sample(-100:100, n, replace = TRUE)
  if (func == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "Kadane1D_cpp") 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 == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "Kadane1D_cpp") 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~   0.328   0.524       0.551     0.673   0.589    5.73   rand~
## 2  1000 kadane_ra~   0.0699  0.111       0.113     0.117   0.123    0.232  rand~
## 3  1000 naive_sor~   0.0182  0.0291      0.0312    0.0346  0.0346   0.0701 sort~
## 4  1000 kadane_so~   0.0194  0.0286      0.0306    0.168   0.0346   6.71   sort~
## 5 10000 naive_ran~  38.6    42.3        44.0      49.4    52.2     82.3    rand~
## 6 10000 kadane_ra~   0.677   1.01        1.12      1.13    1.28     1.67   rand~
## 7 10000 naive_sor~   0.122   0.215       0.246     0.292   0.301    1.77   sort~
## 8 10000 kadane_so~   0.139   0.212       0.241     0.249   0.286    0.398  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.

6 Cas particulier des données presques toutes positives

On considère un vecteur contenant des valeurs positives, avec 5% de valeurs négatives insérées aléatoirement dans le vecteur.

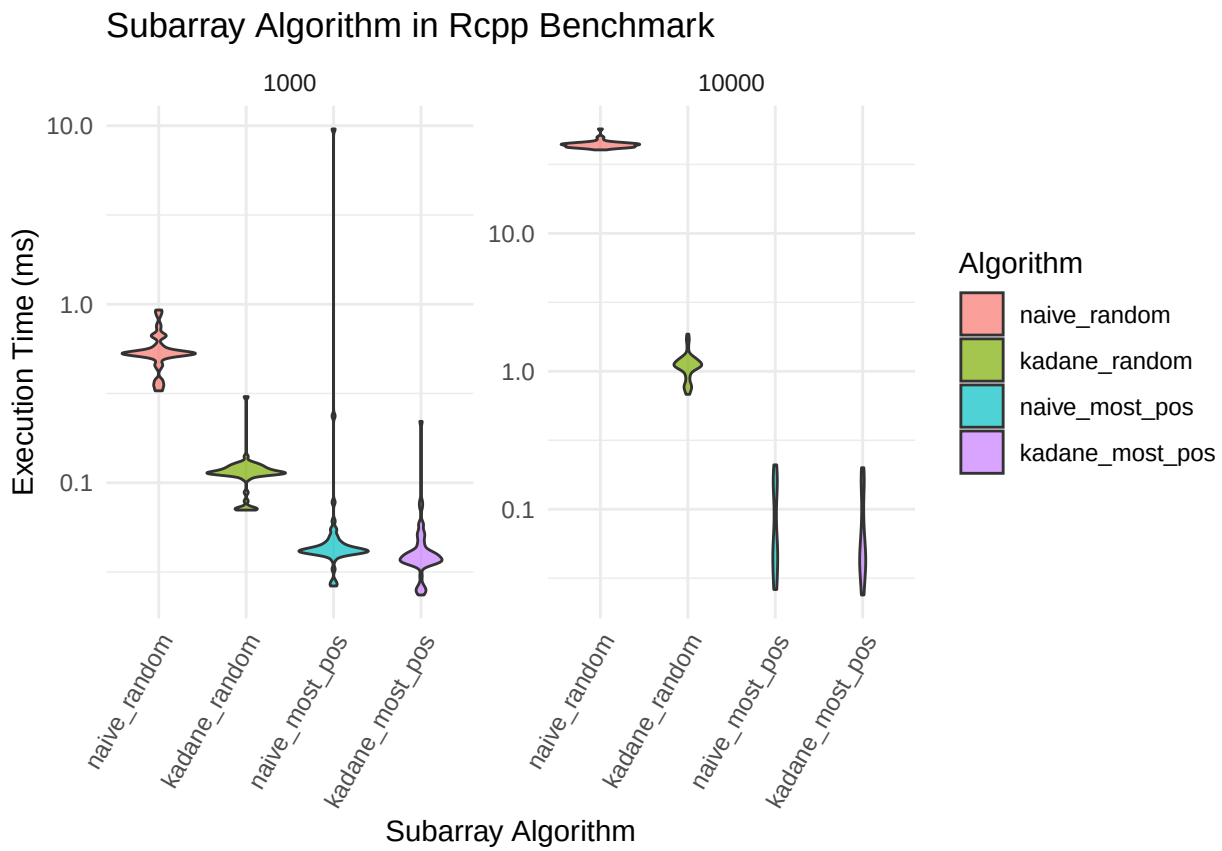
Sur un exemple cela donne :

```r
set.seed(123)

# Vecteur de base : valeurs positives de 1 à 100
```

```r
v_mostly_pos <- sample(1:100)

# Introduire 5% de valeurs négatives aléatoires
n_neg <- floor(0.05 * length(v_mostly_pos))  # 5% de négatifs
neg_indices <- sample(length(v_mostly_pos), n_neg)

# Remplacer ces valeurs par des valeurs négatives aléatoires
v_mostly_pos[neg_indices] <- -sample(1:100, n_neg)

# Affichage du vecteur
v_mostly_pos
```

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

```r
# Fonctions de simulation
one.simu <- function(n, func) {
  v <- sample(-100:100, n, replace = TRUE)
  if (func == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "Kadane1D_cpp") return(max_subarray_sum_opt_Rcpp(v))
}

one.simu2 <- function(n, func) {
  v_mostly_pos <- sample(1:100)
  n_neg <- floor(0.05 * length(v_mostly_pos))
  neg_indices <- sample(length(v_mostly_pos), n_neg)
  v_mostly_pos[neg_indices] <- -sample(1:100, n_neg)
  if (func == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v_mostly_pos))
  if (func == "Kadane1D_cpp") return(max_subarray_sum_opt_Rcpp(v_mostly_pos))
}
```



7 Cas particulier des données presques toutes negatives

On considère un vecteur contenant des valeurs négatives, avec 5% de valeurs positives insérées aléatoirement dans le vecteur.

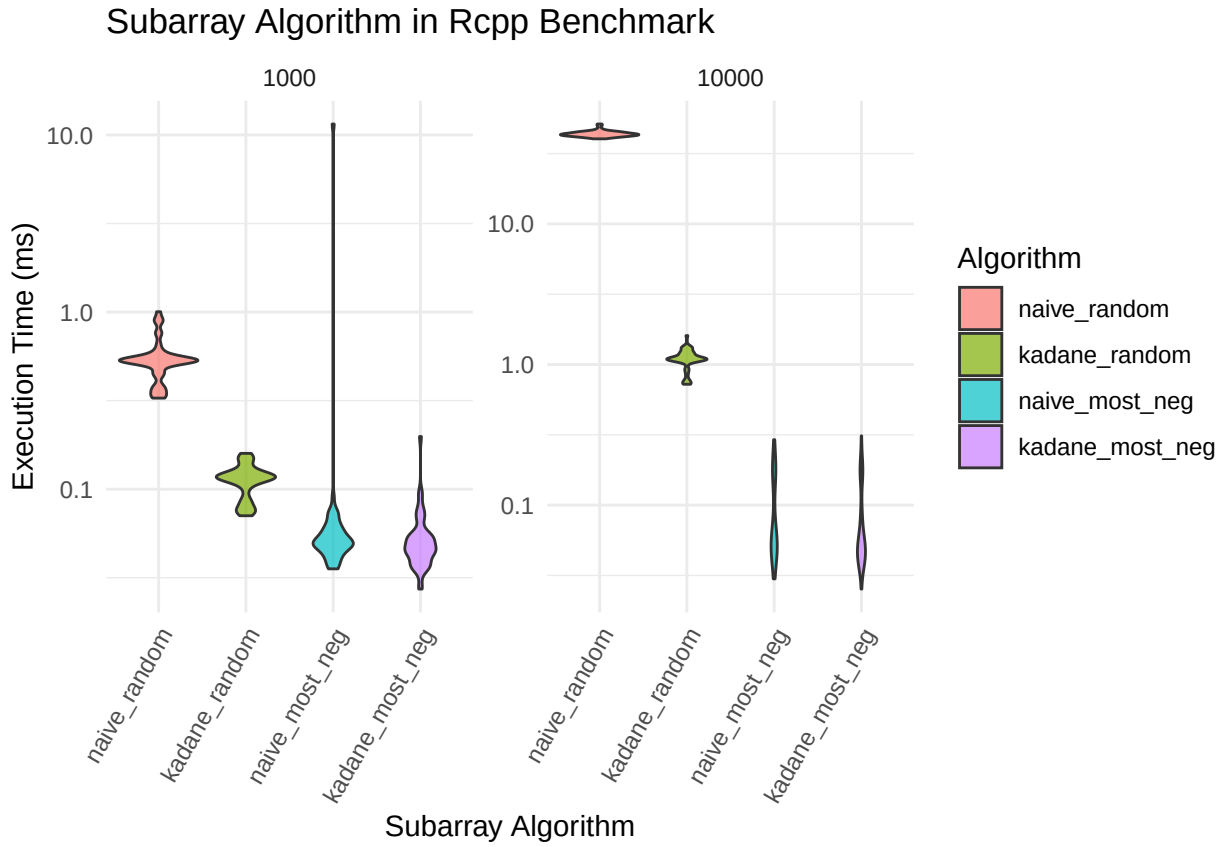
Sur un exemple cela donne :

```r
set.seed(123)

# Vecteur de base : valeurs négatives de -1 à -100
v_mostly_neg <- -sample(1:100)

# Introduire 5% de valeurs positives aléatoires
n_pos <- floor(0.05 * length(v_mostly_neg))  # 5% de positifs
pos_indices <- sample(length(v_mostly_neg), n_pos)

# Remplacer ces valeurs par des valeurs positives aléatoires
v_mostly_neg[pos_indices] <- sample(1:100, n_pos)

# Affichage du vecteur
v_mostly_neg
```

```
##   [1]  -31  -79  -51  -14  -67  -42  -50  -43  -97  -25  -90  -69  -57   -9  -72
##  [16]  -26   94  -95  -87  -36  -78  -93  -76  -15  -32  -84  -82  -41  -23  -27
##  [31]  -60  -53  -75  -89  -71  -38  -91  -34  -29   -5   -8  -12  -13  -18  -33
##  [46]   35  -64  -65  -21  -77  -73  -47  -85 -100  -16  -30   -6  -99  -70  -22
```

```
##  [61]  -94   79  -49  -17  -63   -4  -58  -61  -40  -96  -19  -54  -20  -80  -62
##  [76]   46  -86   -3  -83  -46  -59  -48  -24   54  -81  -68  -88  -98  -44  -10
##  [91]  -56  -11  -55  -37   -2  -28  -74  -35  -52   -1
```

```r
# Fonctions de simulation
one.simu <- function(n, func) {
  v <- sample(-100:100, n, replace = TRUE)
  if (func == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v))
  if (func == "Kadane1D_cpp") return(max_subarray_sum_opt_Rcpp(v))
}

one.simu2 <- function(n, func) {

  v_mostly_neg <- -sample(1:100)
  n_pos <- floor(0.05 * length(v_mostly_neg))
  pos_indices <- sample(length(v_mostly_neg), n_pos)
  v_mostly_neg[pos_indices] <- sample(1:100, n_pos)
  if (func == "Naive1D_cpp") return(max_subarray_sum_naive_Rcpp(v_mostly_neg ))
  if (func == "Kadane1D_cpp") return(max_subarray_sum_opt_Rcpp(v_mostly_neg ))
}
```




```
## # 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~   0.327   0.476       0.530     0.541   0.553     1.00  rand~
## 2  1000 kadane_ra~   0.0708  0.0990      0.116     0.112   0.120     0.159 rand~
## 3  1000 naive_mos~   0.0355  0.0464      0.0498    0.282   0.0588   11.5   rand~
```

```
## 4  1000 kadane_mo~   0.0273  0.0419      0.0464    0.0532  0.0544    0.198 rand~
## 5 10000 naive_ran~  40.3    42.3        43.4      43.7    44.6      51.3   rand~
## 6 10000 kadane_ra~   0.724   1.06        1.09      1.09    1.19      1.61  rand~
## 7 10000 naive_mos~   0.0299  0.0476      0.0586    0.0983  0.169     0.292 rand~
## 8 10000 kadane_mo~   0.0253  0.0426      0.0523    0.0821  0.0822    0.311 rand~
## # 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 presques toutes négatif, Naïf et Kadane sont beaucoup plus rapides, avec un écart réduit entre les deux.