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

Khalil Ounis, Manal Derghal, 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)
set.seed(123)

one.simu.time <- function(n, func, data_type = "random") {
  if (data_type == "random") {
    v <- sample(-100:100, n, replace = TRUE)
  } else if (data_type == "all_negative") {
    v <- sample(-100:-1, n, replace = TRUE)
  } else if (data_type == "all_positive") {
    v <- sample(0:100, n, replace = TRUE)
  } else {
    stop("data_type inconnu")
  }

  if (func == "naive") {
    t <- microbenchmark(max_subarray_sum_naive(v), times = 1)$time / 1e6
  } else if (func == "naive_Rcpp") {
    t <- microbenchmark(max_subarray_sum_naive_Rcpp(v), times = 1)$time / 1e6
  } else if (func == "opt") {
    t <- microbenchmark(max_subarray_sum_opt(v), times = 1)$time / 1e6
  } else if (func == "opt_Rcpp") {
    t <- microbenchmark(max_subarray_sum_opt_Rcpp(v), times = 1)$time / 1e6
  } else {
    stop("fonction inconnue")
  }

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

3.1 Un essai

Sur un exemple, on obtient :

```
set.seed(123)
n <- 10000
one.simu.time(n, func = "naive")
```

```
## [1] 2487.89
```

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

```
## [1] 29.42
```

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

```
## [1] 1.11
```

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

```
## [1] 0.13
```

3.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste :

```
set.seed(123)
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 = "naive")}
for(i in 1:nbSimus){time2[i] <- one.simu.time(n, func = "naive_Rcpp")}
for(i in 1:nbSimus){time3[i] <- one.simu.time(n, func = "opt")}
for(i in 1:nbSimus){time4[i] <- one.simu.time(n, func = "opt_Rcpp")}
```

Gain C++ versus R

```
naive_speedup_cpp <- mean(time1) / mean(time2)
kadane_speedup_cpp <- mean(time3) / mean(time4)
naive_speedup_cpp
```

```
## [1] 79.13895
```

```
kadane_speedup_cpp
```

```
## [1] 4.864253
```

Gain naive versus optimisé

```
kadane_vs_naive_R <- mean(time1) / mean(time3)
kadane_vs_naive_Rcpp <- mean(time2) / mean(time4)
kadane_vs_naive_R
```

```
## [1] 2163.18
```

```
kadane_vs_naive_Rcpp
```

```
## [1] 132.9593
```

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

```
set.seed(123)
n <- 20000
nbSimus <- 10
time3 <- rep(0, nbSimus); time4 <- rep(0, nbSimus)
for(i in 1:nbSimus){time3[i] <- one.simu.time(n, func = "opt")}
for(i in 1:nbSimus){time4[i] <- one.simu.time(n, func = "opt_Rcpp")}
median_kadane_R_vs_Rcpp <- median(time3) / median(time4)
median_kadane_R_vs_Rcpp
```

```
## [1] 7.215686
```

Conclusion :

3.2.1 Performances C++ vs R :

— Naïf : C++ 79× plus rapide

— Kadane : C++ 5× plus rapide → 7× pour n=20k

3.2.2 Efficacité algorithmique :

— Kadane 2163× mieux que naïf en R

— Kadane 133× mieux que naïf en C++
— Confirme $O(n^2)$ naïf vs $O(n)$ Kadane

3.2.3 Recommandations :

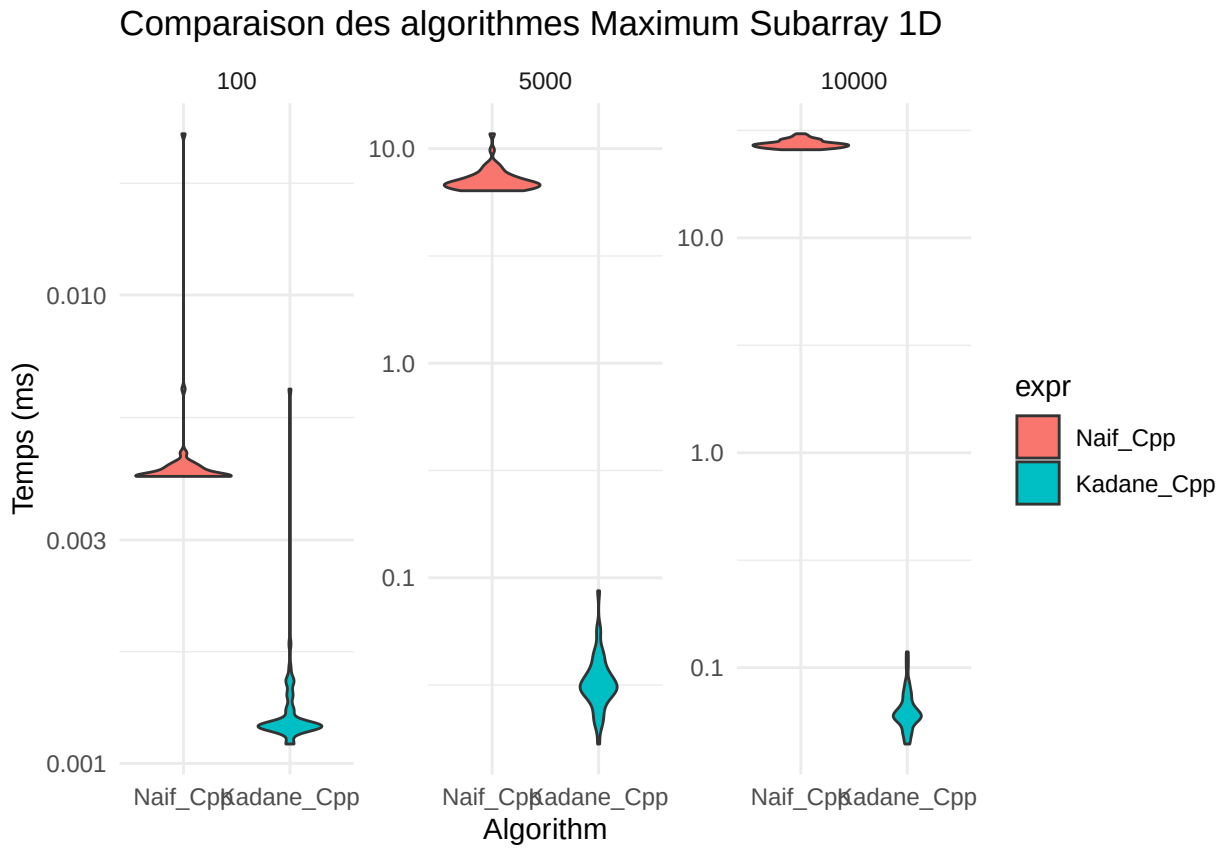
— n > 1k : Toujours préférer Kadane

— n > 10k : Obligatoire d'utiliser Rcpp

— Très grands n : Seul Kadane+Rcpp reste viable

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` 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 Naif_Cpp     0.0041  0.0041      0.0041   0.00459  0.0043   0.0221
## 2   100 Kadane_Cpp   0.0011  0.0012      0.0012   0.00136  0.0013   0.0063
## 3  5000 Naif_Cpp     6.37    6.64        6.88     7.17     7.39    11.7
## 4  5000 Kadane_Cpp   0.0168  0.0281      0.0310   0.0337   0.0363   0.087
```

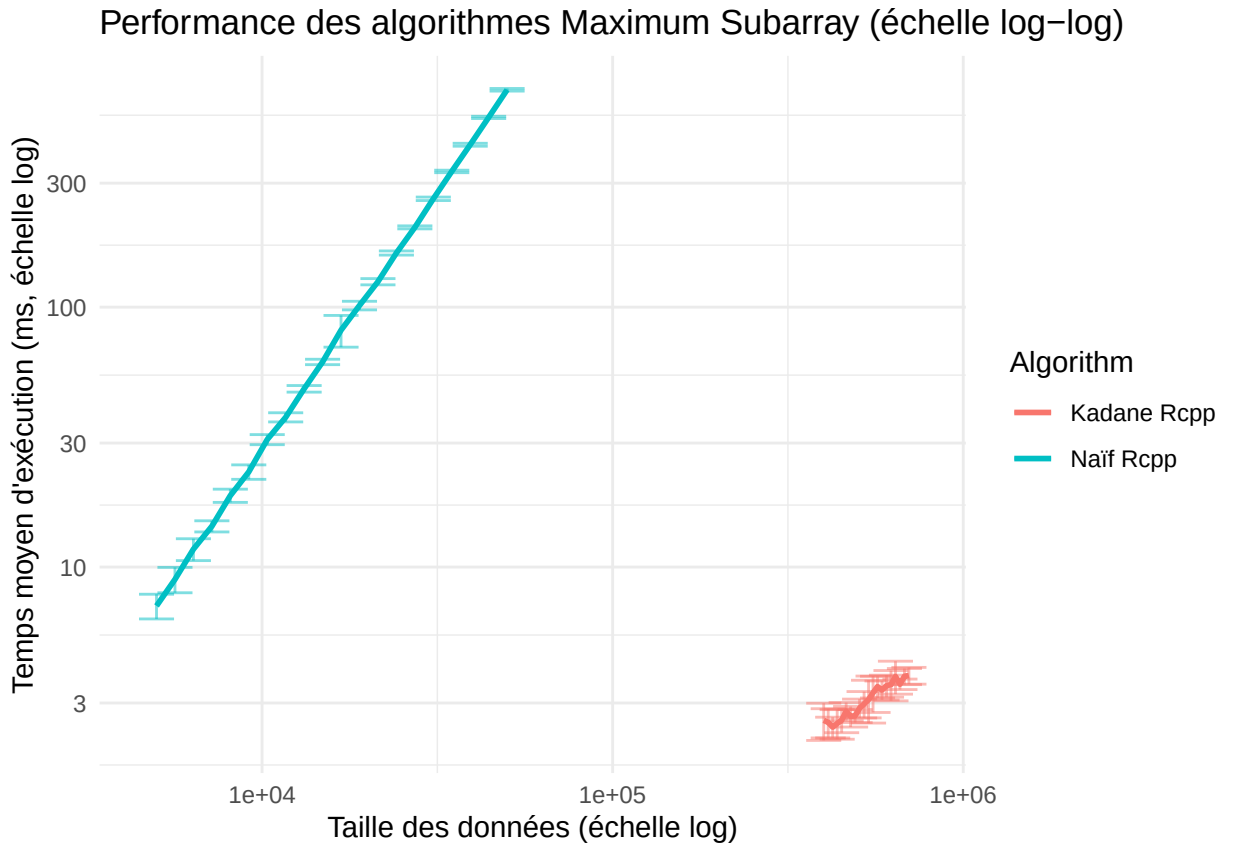
```
## 5 10000 Naif_Cpp    25.7    26.4        27.2     27.3     27.8     30.5
## 6 10000 Kadane_Cpp   0.0441  0.0568      0.0604   0.0642   0.0673   0.118
```

---

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
res_Naive
```

```
##        n mean_time    sd_time
## 1   5000     7.089  0.7716713
## 2   5644     8.969  1.0040081
## 3   6371    11.722  1.1311626
## 4   7192    14.360  0.7093189
## 5   8119    18.839  1.1019019
## 6   9165    23.243  1.4850518
## 7  10346    30.953  1.3783569
## 8  11679    37.709  1.5282557
## 9  13183    48.524  1.4181381
## 10 14882    61.565  1.4698772
```

```
## 11 16799    81.555 11.3921506
## 12 18963   101.494  3.8724359
## 13 21407   125.332  3.4540244
## 14 24165   161.602  2.9803311
## 15 27278   202.783  2.7031589
## 16 30792   261.185  4.0527748
## 17 34760   332.876  3.7155595
## 18 39238   421.695  5.4820439
## 19 44293   537.211  5.3039827
## 20 50000   686.189  7.9705673
```

```
res_Kadane
```

```
##         n mean_time   sd_time
## 1  400000     2.573 0.4202658
## 2  411957     2.526 0.3252076
## 3  424271     2.423 0.2216629
## 4  436953     2.502 0.3260811
## 5  450014     2.573 0.2672514
## 6  463465     2.788 0.2238948
## 7  477319     2.666 0.2424046
## 8  491587     2.676 0.1539264
## 9  506281     2.866 0.2386629
## 10 521415     2.974 0.3452600
## 11 537001     3.090 0.5790605
## 12 553052     3.283 0.5251677
## 13 569584     3.468 0.3315888
## 14 586610     3.363 0.2810318
## 15 604144     3.498 0.3401895
## 16 622203     3.528 0.4707394
## 17 640802     3.794 0.5501555
## 18 659956     3.543 0.1643878
## 19 679683     3.800 0.2701851
## 20 700000     3.828 0.2839718
```

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
## -0.019263 -0.010687 -0.002297  0.005399  0.035703
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -14.922447   0.047727  -312.7   <2e-16 ***
## log(res_Naive$n)   1.982513   0.004924   402.7   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01539 on 18 degrees of freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:  0.9999
## F-statistic: 1.621e+05 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 1.982513
```

```
##
## Call:
## lm(formula = log(res_Kadane$mean_time) ~ log(res_Kadane$n))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.066554 -0.028535 -0.000786  0.027560  0.075691
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -10.4858     0.6841  -15.33 8.96e-12 ***
## log(res_Kadane$n)   0.8803     0.0519   16.96 1.62e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.03942 on 18 degrees of freedom
## Multiple R-squared:  0.9411, Adjusted R-squared:  0.9378
## F-statistic: 287.7 on 1 and 18 DF,  p-value: 1.622e-12
```

```
## Exposant estimé (Kadane): 0.8802988
```

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 :

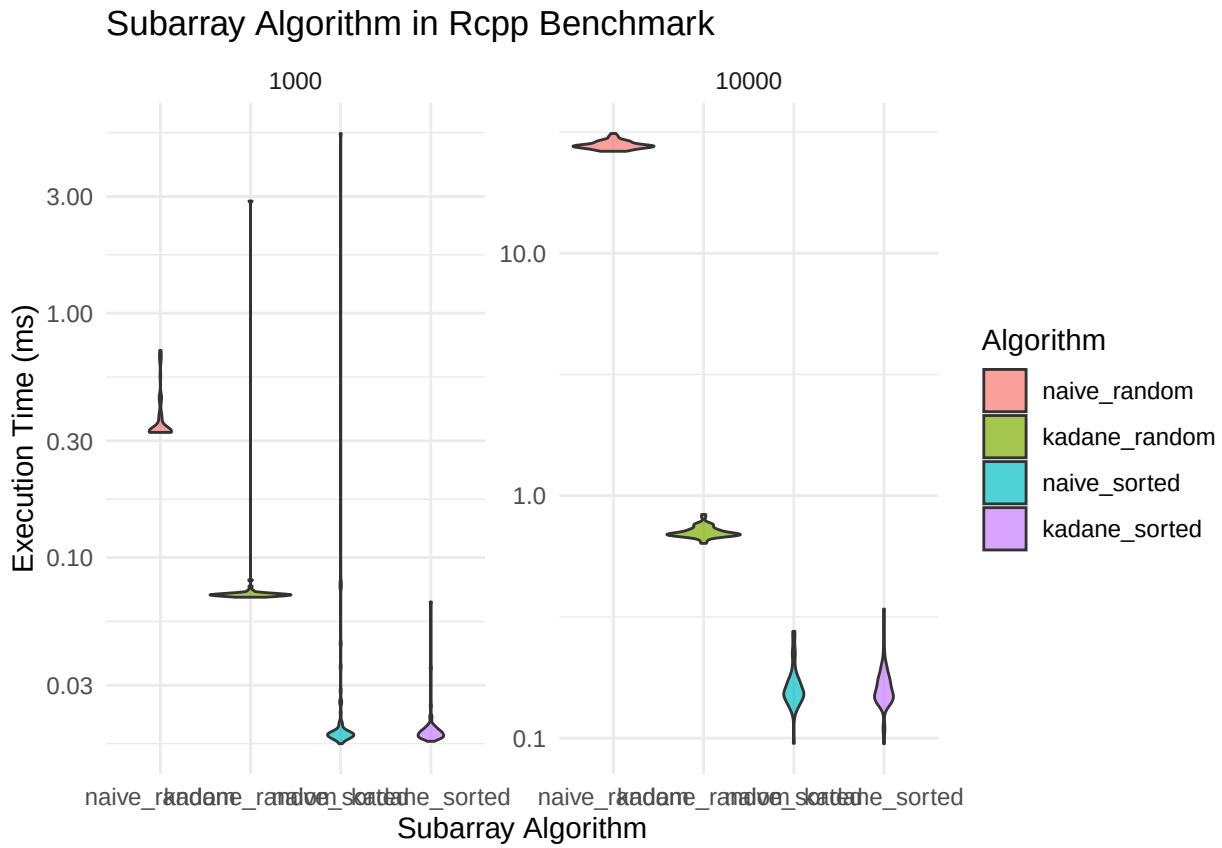
```r
v <- 1:100
n_swap <- floor(0.05 * length(v))
swap_indices <- sample(length(v), n_swap)
v[swap_indices] <- sample(v[swap_indices])
v
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  85  67  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  31  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  32  86  87  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 == "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~   0.325   0.327       0.329     0.376   0.374    0.704  rand~
## 2  1000 kadane_ra~   0.0688  0.0698      0.0704    0.127   0.0713   2.88   rand~
## 3  1000 naive_sor~   0.0173  0.0185      0.0191    0.131   0.0200   5.43   sort~
## 4  1000 kadane_so~   0.0177  0.0185      0.0191    0.0206  0.0198   0.0658 sort~
## 5 10000 naive_ran~  26.4    27.3        27.8      27.9    28.6     31.2    rand~
## 6 10000 kadane_ra~   0.636   0.685       0.697     0.709   0.724    0.837  rand~
## 7 10000 naive_sor~   0.0951  0.146       0.155     0.165   0.173    0.276  sort~
## 8 10000 kadane_so~   0.0948  0.145       0.156     0.163   0.173    0.341  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.