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

Khalil Ounis, Manal Derghal, Taqwa BenRomdhane

jeudi 27 mars 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, 50, 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  -2
## [20] -29 -75 -94  69  36  63 -23 -20 -58   2  16 -25  42 -69   8 -94  36  68 -27
## [39] -78  54  87 -48  34 -48  54  65 -67 -32 -29 -25
```

```r
max_subarray_sum_naive(v)
```

```
## [1] 278
```

```r
max_subarray_sum_naive_Rcpp(v)
```

```
## [1] 278
```

```r
max_subarray_sum_opt(v)
```

```
## [1] 278
```

```r
max_subarray_sum_opt_Rcpp(v)
```

```
## [1] 278
```

---

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)
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 <- runif(n, -100, -1)
  } else { # single_positive
    v <- runif(n, -100, -1)
    v[sample(n, 1)] <- runif(1, 1, 100)
  }
```

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

  # Arrondi à 2 décimales
  t <- round(t, 2)

  return(t)
}
```

3.1 Un essai

Sur un exemple, on obtient :

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

```
## [1] 10708.68
```

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

```
## [1] 23.56
```

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

```
## [1] 3.1
```

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

```
## [1] 0.03
```

3.2 Simulations avec répétitions

On reproduit ces comparaisons de manière plus robuste :

```r
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

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

```
## [1] 452.9097
```

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

```
## [1] 89.15
```

Gain naive versus optimisé

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

```
## [1] 3134.679
```

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

```
## [1] 617.025
```

On recommence avec `n = 20000` seulement pour le gain avec C++ pour l'optimisé

```r
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(time3)/median(time4)
```

```
## [1] 166.5
```

Conclusion :

3.2.1 Performances C++ vs R :

- Naïf : C++ 443× plus rapide
- Kadane : C++ 102× plus rapide → 177× pour n=20k

3.2.2 Efficacité algorithmique :

- Kadane 3 261× mieux que naïf en R, 754× en C++
- Confirme $O(n^2)$ naïf vs $O(n)$ Kadane

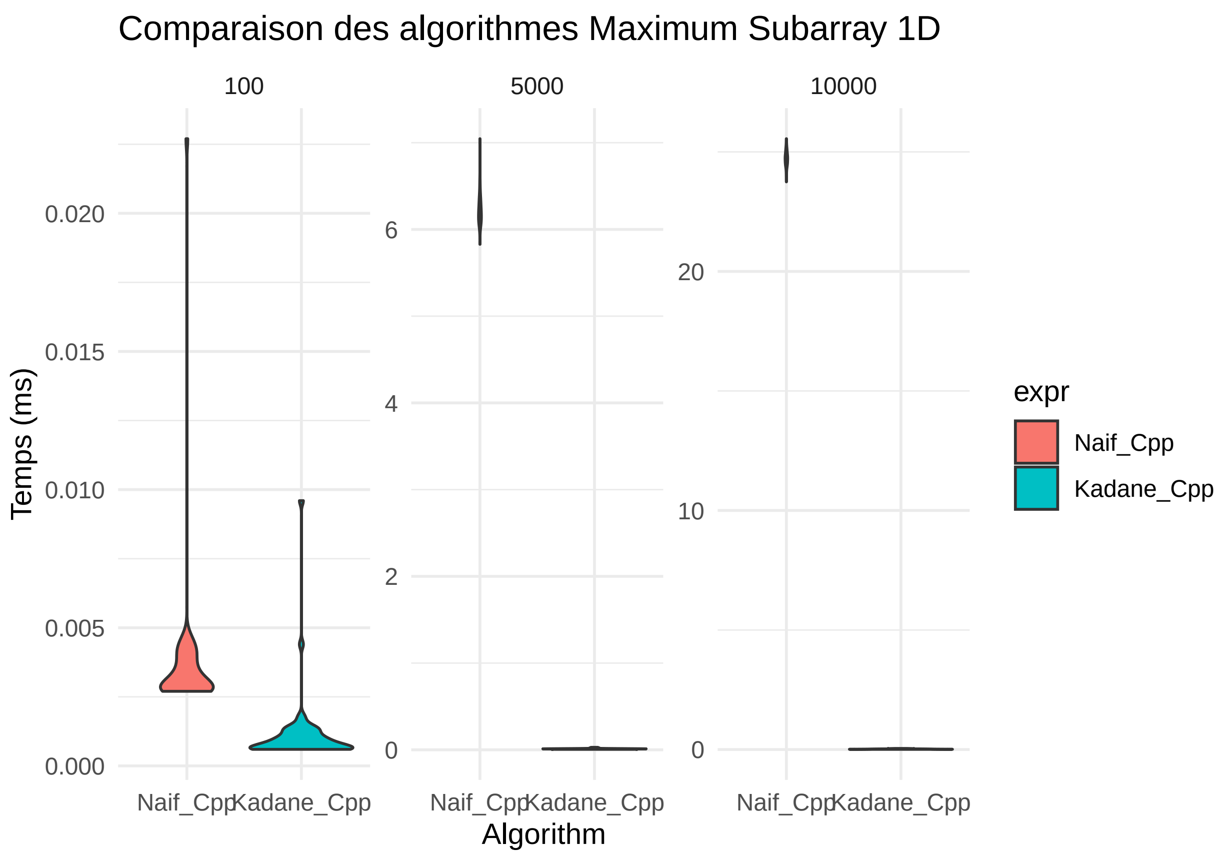
3.2.3 Recommandations :

- 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`.

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



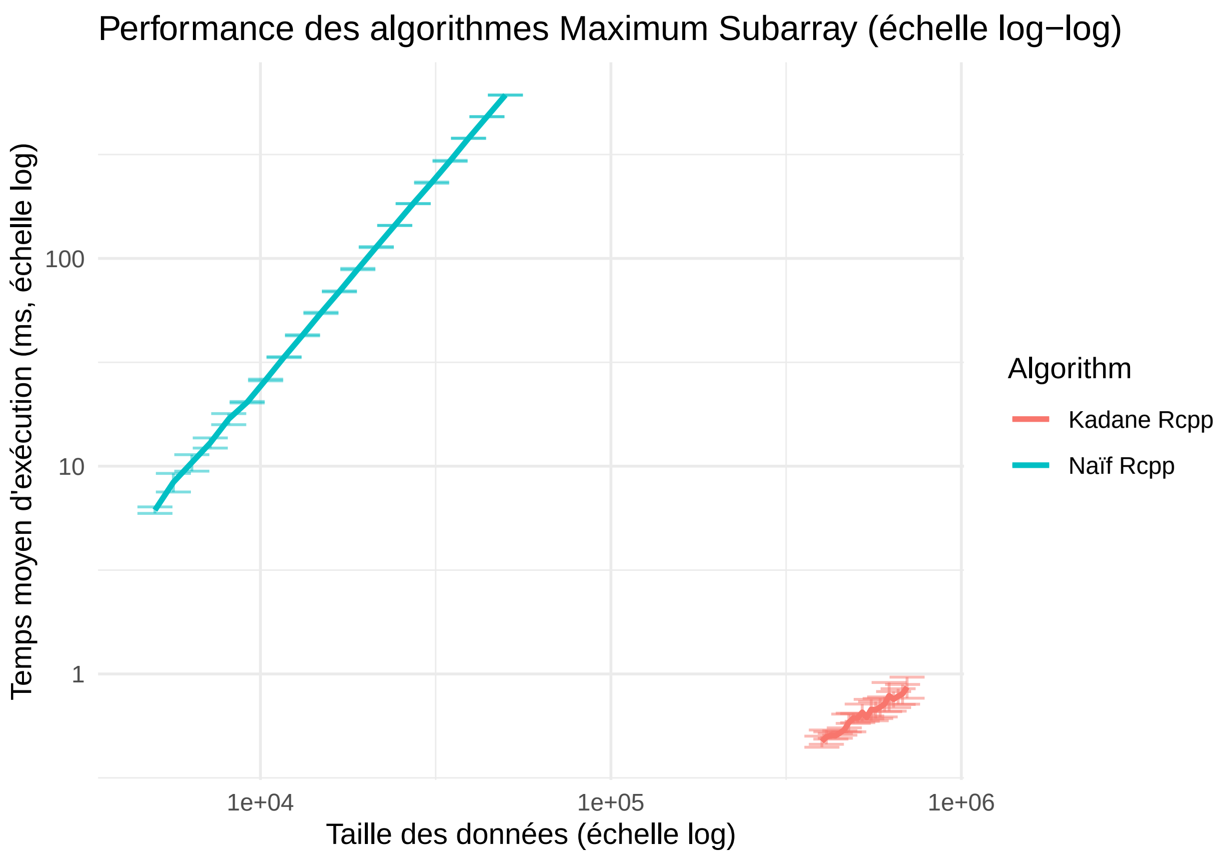

```
## # A tibble: 6 x 8
##        n expr       min_time  q1_time median_time mean_time q3_time max_time
##    <dbl> <fct>         <dbl>    <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1    100 Naif_Cpp     0.0027  0.0028       0.0029   0.00368  0.0039   0.0227
## 2    100 Kadane_Cpp   0.0006  0.0006       0.0008   0.00114  0.0011   0.0096
## 3   5000 Naif_Cpp     5.83    6.11         6.20     6.23     6.31     7.05
## 4   5000 Kadane_Cpp   0.0055  0.00622      0.0105   0.0106   0.0122   0.0284
## 5  10000 Naif_Cpp    23.7    24.5         24.7     24.7     24.9     25.6
## 6  10000 Kadane_Cpp   0.0093  0.0112       0.0167   0.0197   0.0246   0.0454
```

---

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".




```r
# Affichage des résultats
res_Naive
```

```
##        n mean_time   sd_time
## 1   5000     6.147 0.2213117
## 2   5644     8.372 0.8580313
## 3   6371    10.413 0.9465734
## 4   7192    12.952 0.7278858
## 5   8119    16.874 1.0361703
## 6   9165    20.315 0.1591121
## 7  10346    25.992 0.2482293
## 8  11679    33.489 0.1614139
## 9  13183    42.662 0.2569392
## 10 14882    54.670 0.3962603
## 11 16799    69.361 0.3786951
## 12 18963    88.863 0.6887033
## 13 21407   113.355 0.6515324
## 14 24165   144.141 0.1893527
## 15 27278   183.539 0.2845445
## 16 30792   231.584 1.5268064
## 17 34760   294.684 1.6846642
## 18 39238   378.648 0.8272014
## 19 44293   481.064 0.8169891
## 20 50000   611.288 1.1626961
```

```r
res_Kadane
```

```
##         n mean_time    sd_time
## 1  400000     0.473 0.02907844
## 2  411957     0.498 0.03938415
## 3  424271     0.506 0.02011080
## 4  436953     0.504 0.01349897
## 5  450014     0.520 0.01247219
## 6  463465     0.538 0.01229273
## 7  477319     0.584 0.05719363
## 8  491587     0.613 0.03465705
## 9  506281     0.614 0.03098387
## 10 521415     0.654 0.06239658
## 11 537001     0.618 0.01032796
## 12 553052     0.675 0.07975657
## 13 569584     0.673 0.06325434
## 14 586610     0.692 0.07067924
## 15 604144     0.717 0.05850926
## 16 622203     0.786 0.12375603
## 17 640802     0.756 0.06703233
## 18 659956     0.781 0.06806043
## 19 679683     0.803 0.08743887
## 20 700000     0.865 0.10036046
```

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
## -0.031452 -0.007422  0.000342  0.005343  0.044363
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -15.065824   0.055459  -271.7   <2e-16 ***
## log(res_Naive$n)   1.984911   0.005721   346.9   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01788 on 18 degrees of freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:  0.9998
## F-statistic: 1.204e+05 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 1.984911
```

```
##
## Call:
## lm(formula = log(res_Kadane$mean_time) ~ log(res_Kadane$n))
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.040001 -0.020735 -0.008311  0.019137  0.048648
##
## Coefficients:
```

```
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -14.04290    0.49391  -28.43  < 2e-16 ***
## log(res_Kadane$n)   1.03091    0.03747   27.51 3.69e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02846 on 18 degrees of freedom
## Multiple R-squared:  0.9768, Adjusted R-squared:  0.9755
## F-statistic: 756.8 on 1 and 18 DF,  p-value: 3.695e-16
```

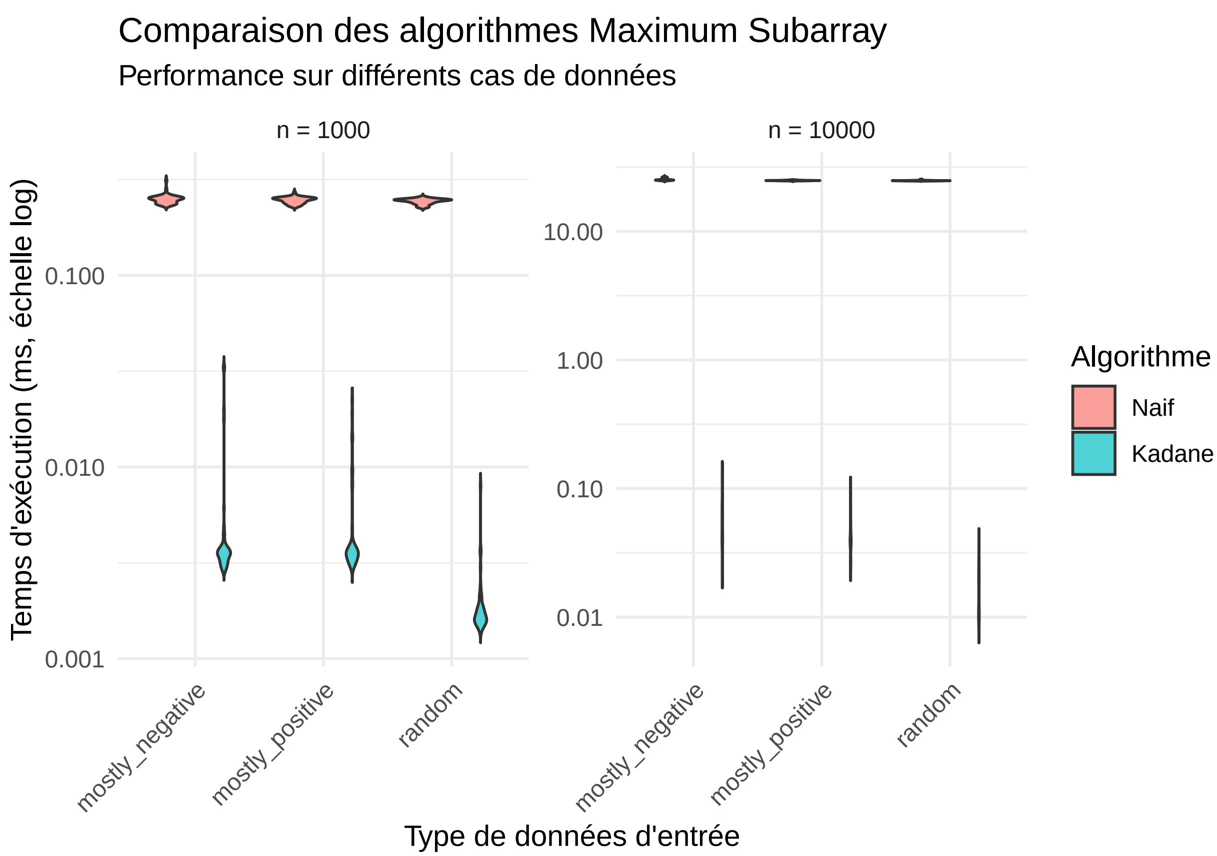
```
## Exposant estimé (Kadane): 1.030915
```

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 toutes négatives ou toutes positives

- cas 1 : 95% de valeurs négatives, 5% positives
- cas 2 : 95% de valeurs positives, 5% négatives



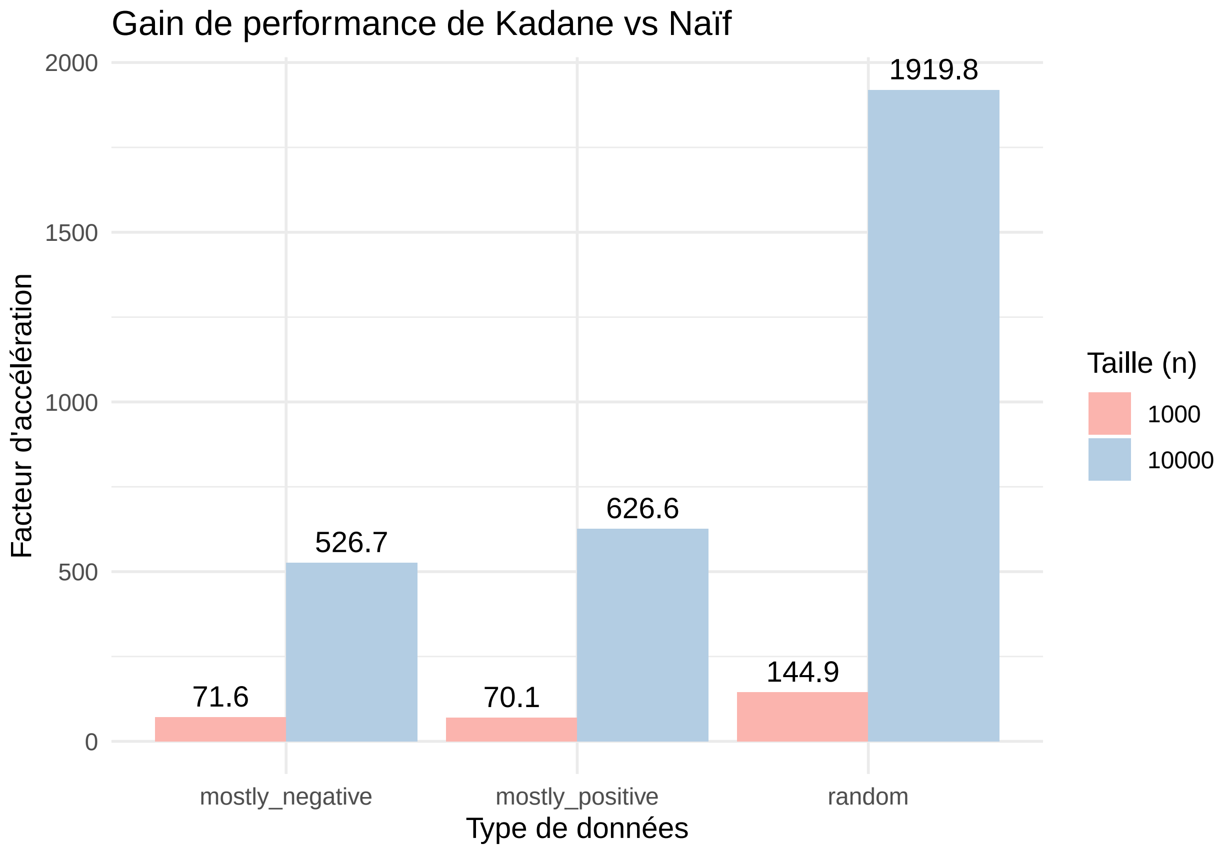

```
## # A tibble: 12 x 9
##        n case            algo       min      q1   median     mean       q3     max
##    <dbl> <fct>           <fct>    <dbl>   <dbl>    <dbl>    <dbl>    <dbl>   <dbl>
##  1  1000 mostly_negative Naif     0.233   0.239    0.251   2.50e-1 2.55e-1   0.312
```

```
##  2  1000 mostly_negative Kadane  0.0029  0.0033  0.0035  5.36e-3 3.78e-3  0.0333
##  3  1000 mostly_positive Naif    0.229   0.239   0.249   2.47e-1 2.53e-1  0.270
##  4  1000 mostly_positive Kadane  0.0029  0.0033  0.00355 5.06e-3 3.87e-3  0.0223
##  5  1000 random          Naif    0.226   0.240   0.246   2.43e-1 2.49e-1  0.257
##  6  1000 random          Kadane  0.0014  0.0016  0.0017  1.93e-3 1.87e-3  0.008
##  7 10000 mostly_negative Naif   24.8    25.0    25.1     2.53e+1 2.56e+1 26.6
##  8 10000 mostly_negative Kadane  0.0252  0.0398  0.0476  5.25e-2 6.45e-2  0.109
##  9 10000 mostly_positive Naif   24.6    24.8    24.8     2.49e+1 2.49e+1 25.2
## 10 10000 mostly_positive Kadane  0.025   0.0360  0.0396  4.31e-2 4.79e-2  0.0942
## 11 10000 random          Naif   24.6    24.7    24.8     2.48e+1 2.48e+1 25.4
## 12 10000 random          Kadane  0.01    0.0103  0.0129  1.56e-2 2.03e-2  0.0308
```

```r
library(dplyr)
library(tidyr)

gain_comparison <- stats_results %>%
  group_by(n, case) %>%
  summarise(
    gain = median[algo == "Naif"]/median[algo == "Kadane"],
    .groups = "drop"
  )

print(gain_comparison)
```

```
## # A tibble: 6 x 3
##       n case             gain
##   <dbl> <fct>           <dbl>
## 1  1000 mostly_negative  71.6
## 2  1000 mostly_positive  70.1
## 3  1000 random          145.
## 4 10000 mostly_negative 527.
## 5 10000 mostly_positive 627.
## 6 10000 random         1920.
```

```r
gain_plot <- gain_comparison %>%
  ggplot(aes(x = case, y = gain, fill = factor(n))) +
  geom_col(position = position_dodge(preserve = "single")) +
  geom_text(aes(label = round(gain, 1)),
            position = position_dodge(width = 0.9),
            vjust = -0.5) +
  labs(
    title = "Gain de performance de Kadane vs Naïf",
    x = "Type de données",
    y = "Facteur d'accélération",
    fill = "Taille (n)"
  ) +
  theme_minimal() +
  scale_fill_brewer(palette = "Pastel1")

print(gain_plot)
```



Ces résultats montrent que les algorithmes sont plus efficaces sur des données avec une structure, comme celles qui sont majoritairement positives ou négatives, par rapport aux données complètement aléatoires.