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

Khalil Ounis, Manal Derghal, Taqwa BenRomdhane

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 :

```r
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 <- sample(-100:-1, n, replace = TRUE)
  } else if (data_type == "all_positive") {
    v <- sample(0:100, n, replace = TRUE)
  } else {
    stop("data_type inconnu")
```

```r
  }

  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 :

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

```
## [1] 13067.59
```

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

```
## [1] 26.43
```

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

```
## [1] 4.49
```

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

```
## [1] 0.08
```

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] 540.614
```

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

```
## [1] 43.1875
```

Gain naive versus optimisé

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

```
## [1] 3579.84
```

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

```
## [1] 285.9792
```

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] 71.08333
```

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`

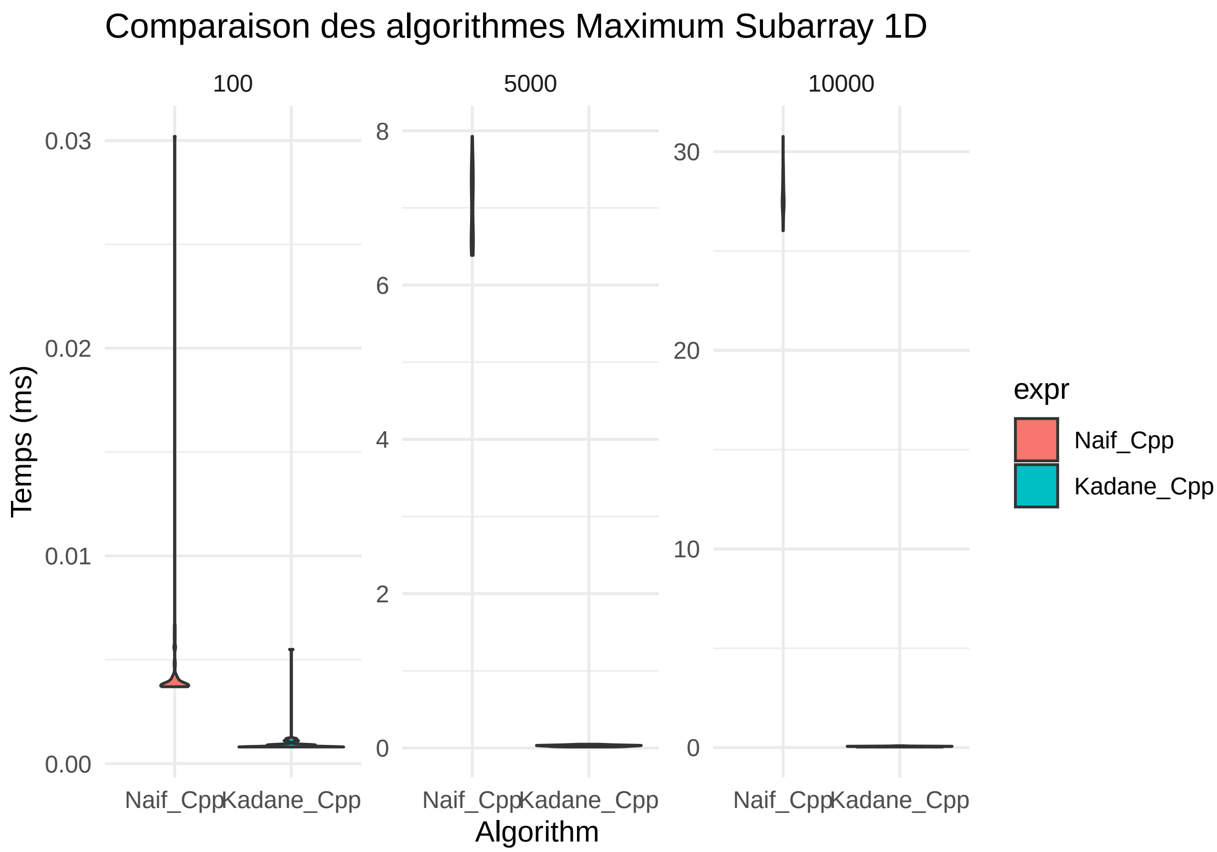
Vous avez besoin des packages `microbenchmark` et `ggplot2` pour exécuter les simulations et afficher les résultats (sous forme de diagrammes en violon). Nous comparons `naive_Rcpp` avec `opt_Rcpp` pour des tailles de données `n = 1000` et `n = 10000`.

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

```
##
## Attachement du package : 'dplyr'
```

```
## Les objets suivants sont masqués depuis 'package:stats':
##
```

```
##     filter, lag
## Les objets suivants sont masqués depuis 'package:base':
##
##     intersect, setdiff, setequal, union
```



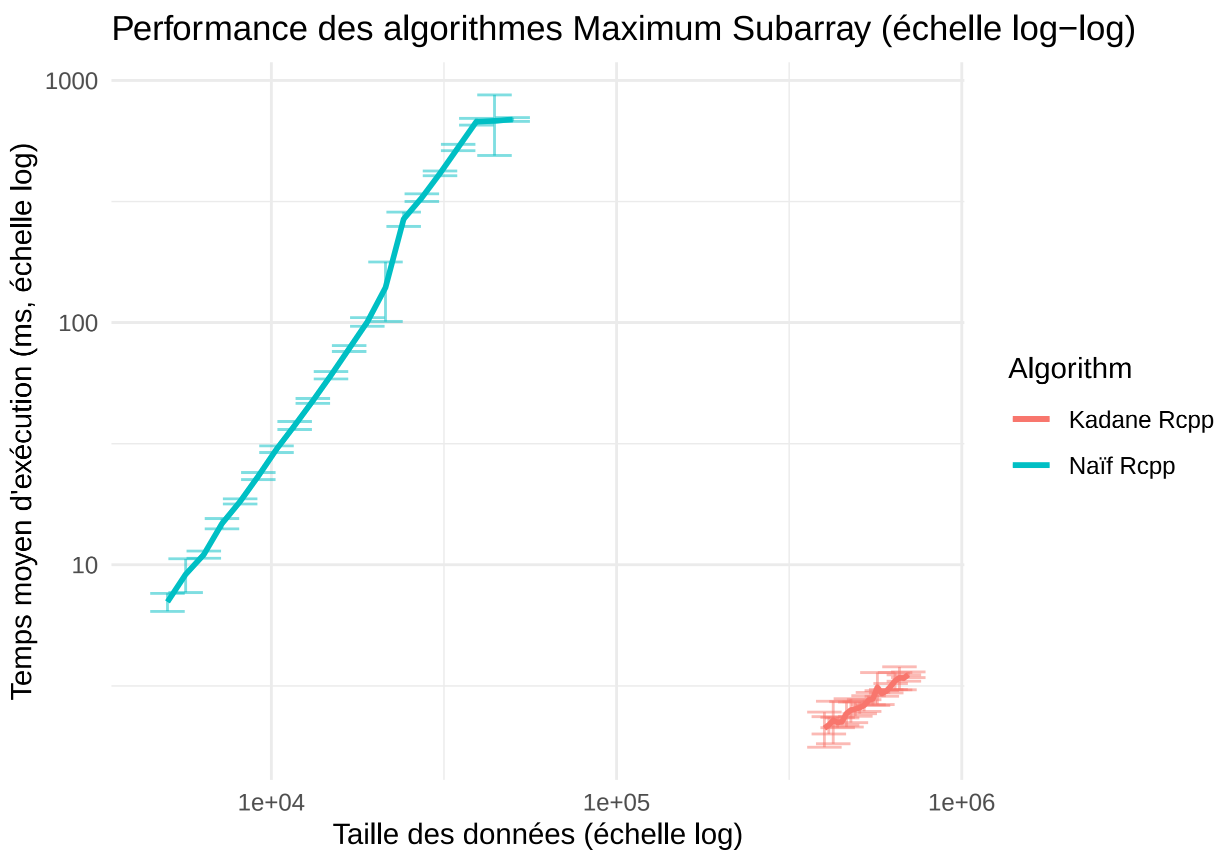

```
## # A tibble: 6 x 8
##       n expr       min_time q1_time median_time mean_time q3_time max_time
##   <dbl> <fct>         <dbl>   <dbl>       <dbl>     <dbl>   <dbl>    <dbl>
## 1   100 Naif_Cpp     0.0037  0.0037      0.0038  0.00461   0.0041   0.0302
## 2   100 Kadane_Cpp   0.0008  0.0008      0.0008  0.000968  0.0009   0.0055
## 3  5000 Naif_Cpp     6.38    6.60        7.03    7.03      7.40     7.93
## 4  5000 Kadane_Cpp   0.0136  0.0226      0.0294  0.0295    0.0351   0.0494
## 5 10000 Naif_Cpp    26.0    27.2        27.6    27.8      28.4     30.8
## 6 10000 Kadane_Cpp   0.0297  0.0422      0.0598  0.0542    0.0633   0.0929
```

---

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     7.030  0.6015720
## 2   5644     9.137  1.4486626
## 3   6371    11.029  0.3733765
## 4   7192    14.803  0.7333341
## 5   8119    18.275  0.4438280
## 6   9165    23.261  0.7987692
## 7  10346    30.011  0.9560271
## 8  11679    37.642  1.4872480
## 9  13183    47.568  1.0732070
## 10 14882    60.649  2.0885426
## 11 16799    78.128  2.1889409
## 12 18963   100.763  4.0222963
## 13 21407   139.588 38.5011812
## 14 24165   267.876 18.3665670
## 15 27278   328.288 12.0094646
## 16 30792   413.570  9.4930735
## 17 34760   528.812 16.0630112
## 18 39238   675.797 21.3498561
## 19 44293   680.835 191.0169773
## 20 50000   689.922 12.4544743
```

```
res_Kadane
```

```
##         n mean_time    sd_time
## 1  400000     2.116 0.35011744
## 2  411957     2.182 0.17980236
## 3  424271     2.280 0.45509462
## 4  436953     2.236 0.10966616
## 5  450014     2.255 0.07706419
## 6  463465     2.432 0.29275701
## 7  477319     2.515 0.28402856
## 8  491587     2.541 0.17084431
## 9  506281     2.573 0.14414884
## 10 521415     2.629 0.14594139
## 11 537001     2.763 0.11851395
## 12 553052     2.801 0.17381024
## 13 569584     3.122 0.47121121
## 14 586610     2.945 0.07706419
## 15 604144     3.007 0.04967673
## 16 622203     3.149 0.08937437
## 17 640802     3.319 0.27497273
## 18 659956     3.419 0.37173617
## 19 679683     3.410 0.10110501
## 20 700000     3.518 0.09319037
```

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
## -0.34533 -0.10013 -0.02736  0.10413  0.29276
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      -16.69069    0.50682  -32.93   <2e-16 ***
## log(res_Naive$n)   2.17866    0.05228   41.67   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1634 on 18 degrees of freedom
## Multiple R-squared:  0.9897, Adjusted R-squared:  0.9892
## F-statistic:  1736 on 1 and 18 DF,  p-value: < 2.2e-16
```

```
## Exposant estimé (naïf): 2.178658
```

```
##
## Call:
## lm(formula = log(res_Kadane$mean_time) ~ log(res_Kadane$n))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.03866 -0.01699 -0.00346  0.01195  0.06722
##
## Coefficients:
```

```
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)        -11.27146    0.41973  -26.85 5.65e-16 ***
## log(res_Kadane$n)    0.93134    0.03185   29.25  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02419 on 18 degrees of freedom
## Multiple R-squared:  0.9794, Adjusted R-squared:  0.9782
## F-statistic: 855.3 on 1 and 18 DF,  p-value: < 2.2e-16
```

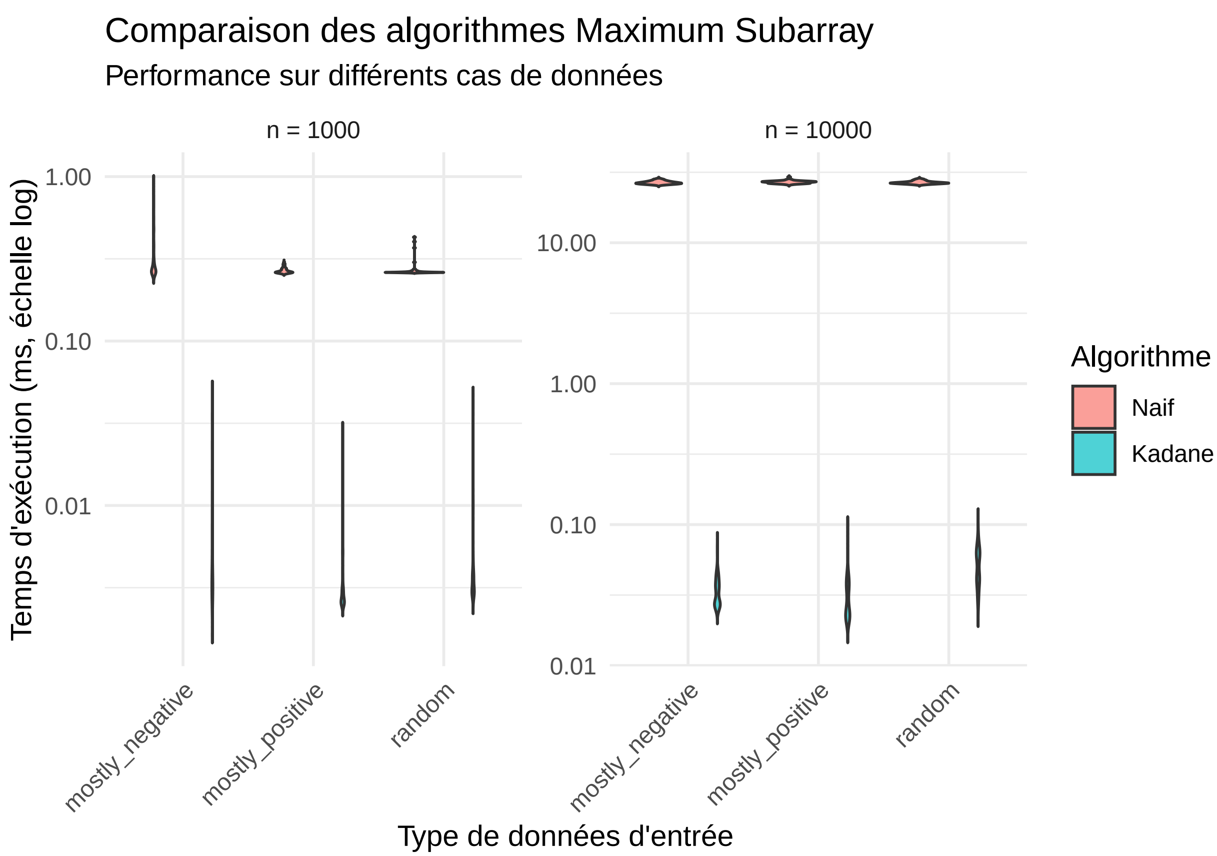
```
## Exposant estimé (Kadane): 0.9313389
```

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



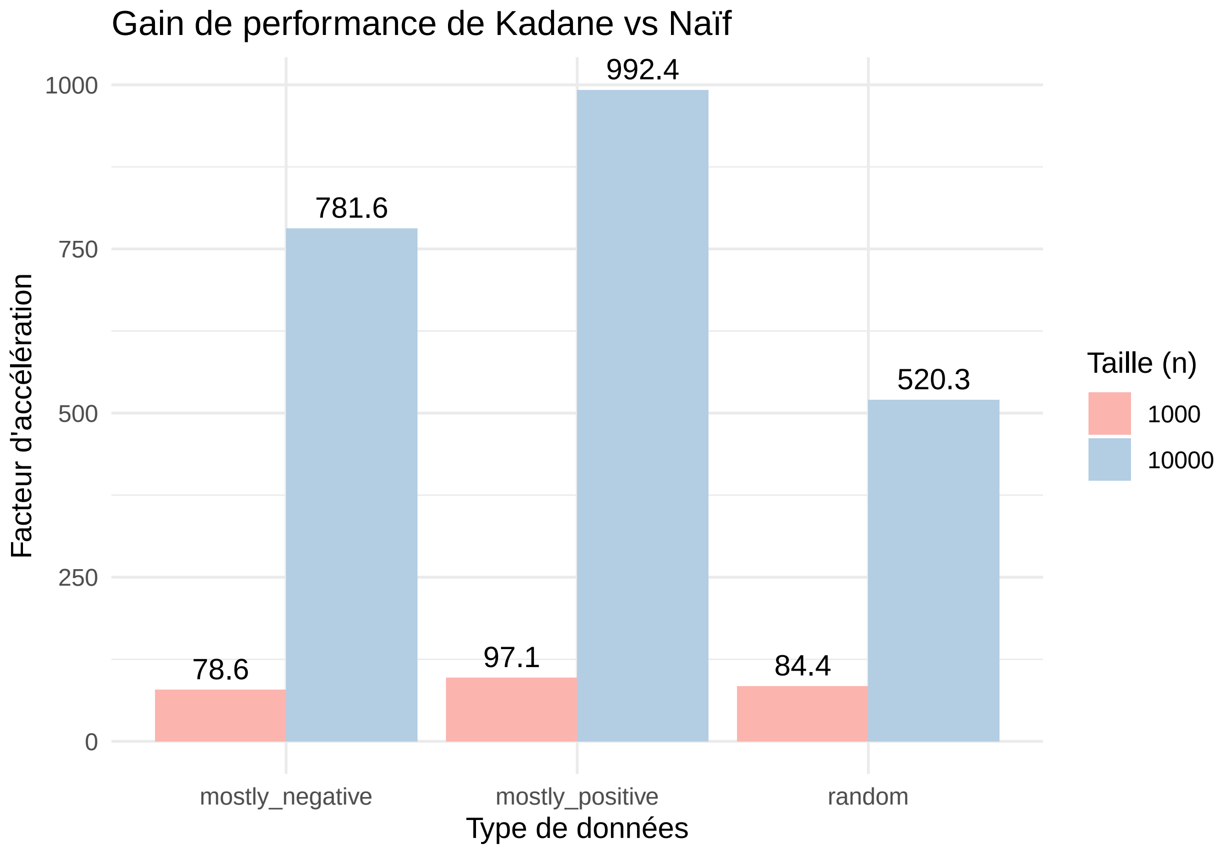

```
## # A tibble: 12 x 9
##         n case            algo       min      q1  median    mean      q3     max
##     <dbl> <fct>           <fct>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1  1000 mostly_negative Naif     0.261   0.262    0.267 3.18e-1 3.08e-1   0.871
```

```
##  2  1000 mostly_negative Kadane  0.003   0.0031   0.0034 6.00e-3 6.78e-3  0.0278
##  3  1000 mostly_positive Naif    0.260   0.261    0.262  2.68e-1 2.72e-1  0.299
##  4  1000 mostly_positive Kadane  0.0025  0.0026   0.0027 3.53e-3 3.1 e-3  0.0272
##  5  1000 random          Naif    0.260   0.261    0.262  2.72e-1 2.64e-1  0.428
##  6  1000 random          Kadane  0.0028  0.00292  0.0031 4.39e-3 3.8 e-3  0.0411
##  7 10000 mostly_negative Naif   25.7    26.2     26.6    2.68e+1 2.71e+1 28.3
##  8 10000 mostly_negative Kadane  0.026   0.0273   0.0340 3.40e-2 3.83e-2  0.0667
##  9 10000 mostly_positive Naif   25.8    26.5     27.0    2.69e+1 2.72e+1 29.2
## 10 10000 mostly_positive Kadane  0.0211  0.0221   0.0272 3.07e-2 3.67e-2  0.078
## 11 10000 random          Naif   25.9    26.4     26.6    2.68e+1 2.71e+1 28.4
## 12 10000 random          Kadane  0.0274  0.0410   0.0511 5.16e-2 6.28e-2  0.0891
```

```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  78.6
## 2  1000 mostly_positive  97.1
## 3  1000 random           84.4
## 4 10000 mostly_negative 782.
## 5 10000 mostly_positive 992.
## 6 10000 random          520.
```

```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.