Algorithme d'optimisation de stockage

Yoann Bonnet, Victorien Leconte, Hugo Picard

M2 Data Science, 2023 - 2024

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

Algorithme *First-fit-decreasing bin packing*

L'algorithme **First-Fit Decreasing Bin Packing (FFD)** est une méthode d'optimisation utilisée en informatique et en recherche opérationnelle pour résoudre le problème de **bin packing**. Ce problème classique consiste à répartir un ensemble d'objets de tailles différentes, ici, il s'agit de jeux, dans le plus petit nombre possible de bacs de capacité fixe, ici espaces mémoires, par exemple, d'une console de jeux.

Implémentation de l'algorithme à l'aide du langage `R`

```r
ffd_bin_packing <- function(games, storage) {
  sorted_games <- sort(games, decreasing = TRUE)
  bins <- list()

  for (game in sorted_games) {
    fitted <- FALSE
    for (i in seq_along(bins)) {
      if (sum(bins[[i]]) + game <= storage) {
        bins[[i]] <- c(bins[[i]], game)
        fitted <- TRUE
        break
      }
    }

        if (!fitted) {
      bins <- c(bins, list(game))
    }
  }

  return(bins)
}
```

Nous pouvons tester cet algorithme sur plusieurs simulations :

```r
games <- sample(1:50, 15, replace = TRUE)
storage <- 100

bins <- ffd_bin_packing(games, storage)
```

```r
cat("Jeux à stocker:", games, "\n")
```

```
## Jeux à stocker: 38 30 10 7 41 30 17 43 28 49 17 12 1 27 16
```

```r
cat("Capacité de stockage de chaque mémoire:", storage, "\n\n")
```

```
## Capacité de stockage de chaque mémoire: 100
```

```r
cat("Répartition des jeux dans les espaces mémoire:\n")
```

```
## Répartition des jeux dans les espaces mémoire:
```

```r
for (i in seq_along(bins)) {
  cat("Mémoire", i, ":", bins[[i]], "\n")
}
```

```
## Mémoire 1 : 49 43 7 1
## Mémoire 2 : 41 38 17
## Mémoire 3 : 30 30 28 12
## Mémoire 4 : 27 17 16 10
```

Explication de l'algorithme

- La première étape, utilisant la fonction `sort()`, ligne trie les jeux par ordre décroissant de taille. Ainsi, les jeux les plus grands seront placés en premier dans les bacs. Cette étape améliore l'efficacité de l'algorithme en réduisant le nombre de mémoires nécessaires.
- On initialise ensuite une liste vide qui sera utilisée pour stocker les différentes mémoires. Chaque élément de la liste représentera un espace mémoire, et la somme des jeux dans chaque mémoire ne dépassera pas la capacité de stockag totale de la mémoire.
- Par la suite, on itère sur chacun des jeux dans la liste triée. L'objectif est de trouver un espace mémoire où le jeu peut être placé. Pour cela, on itère une deuxième fois, cette fois-ci sur la liste des espaces mémoire : pour chaque mémoire, l'algorithme vérifie sur le jeu sélectionné peut être placé dans ce bac, sans dépasser la capacité de stockage. Pour cela, on vérifie si la somme des tailles des jeux déjà dans le bac courant additionné à la taille du jeu courant est inférieure ou égale à la capacité de stockage : si tel est le cas, le jeu courant peut être placé dans cet espace mémoire.
- Une fois que le jeu a été placé dans un espace mémoire, on met à jour la variable `fitted` en lui attribuant la valeur `TRUE` pour indiquer que le jeu courant a trouvé son espace mémoire. Puis, on sort de la boucle sur les espaces mémoire avec l'instruction `break`.
- Enfin, nous vérifions si le jeu courant n'a pas été placé dans un espace mémoire existant. Si tel est le cas, nous créons un nouveau espace pour y placer le jeu courant, avec l'instruction `bins <- c(bins, list(game))`.

Analyse de la complexité

La complexité de notre algorithme dépend de plusieurs facteurs :

- Dans un premier temps, le tri des jeux, avec `sort()`, a une complexité temporelle de $O(n \log(n))$, dans le pire des cas, car c'est la complexité du tri en général, n étant la taille du vecteur des jeux.
- L'initialisation de la liste des bacs, avec `bins <- list()`, est une opération constante, donc sa complexité est $O(1)$.
- La boucle `for (game in sorted_games)` itère sur chaque jeu, donc elle a une complexité de $O(n)$. Cependant, pour chaque jeu, nous avons une autre boucle qui itère sur les espaces mémoire. La boucle `for (i in seq_along(bins))` itère sur chaque espace mémoire. Dans le pire des cas, chaque jeu est

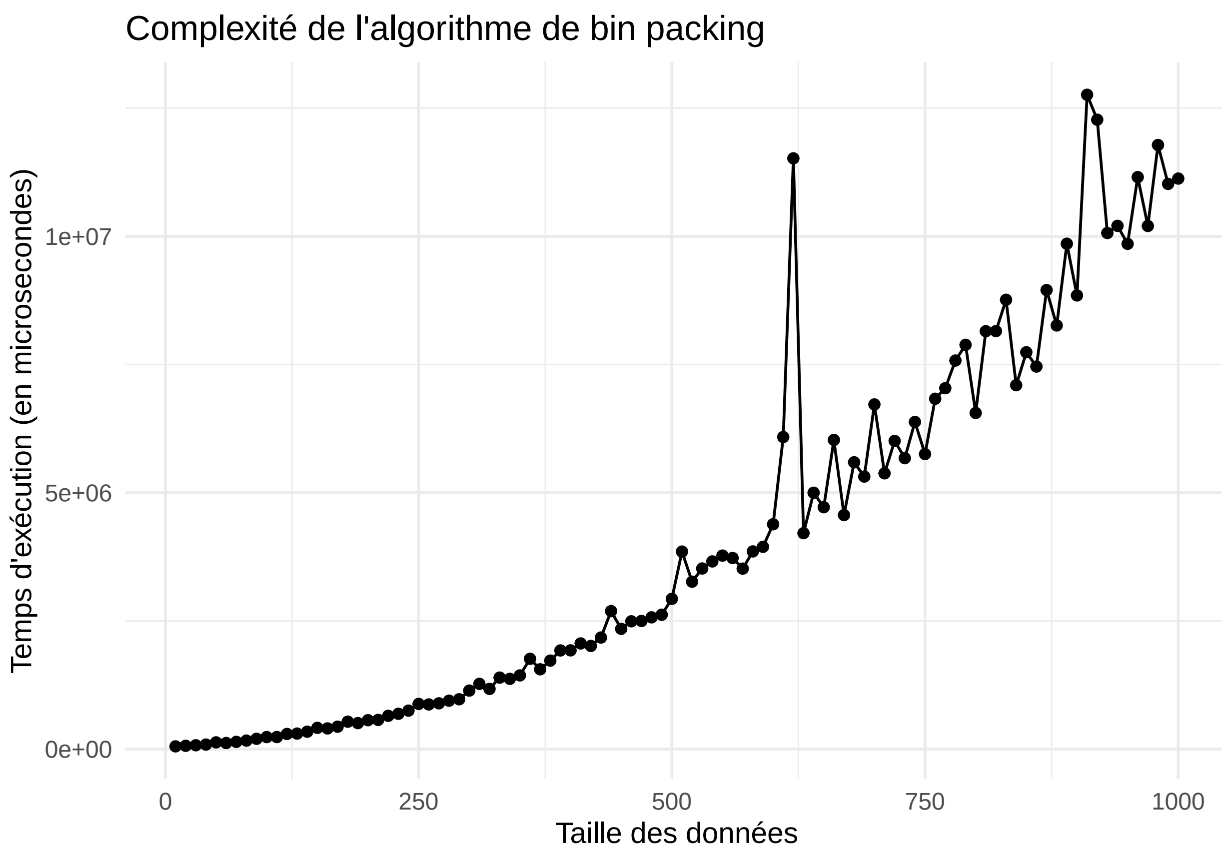
placé dans une mémoire différente, donc il y a autant de mémoires que de jeux. Cela signifie que cette boucle a aussi une complexité de O(n). Comme cette boucle est à l'intérieur de la boucle sur les jeux, la complexité totale de ces deux boucles imbriquées est O(n^2).

- Les opérations à l'intérieur de la boucle sur les espaces mémoire sont toutes des opérations constantes, donc ayant une complexité en O(1). Cependant, étant à l'intérieur de la boucle sur les espaces mémoire, elles ont une complexité totale en O(n).
- Enfin, l'opération `bins <- c(bins, list(game))` est à l'intérieur de la boucle sur les jeux, elle a donc une complexité totale en O(n). En retournant la liste des mémoires utilisées avec `return`, on réalise une opération constante.

Ainsi, en combinant toutes ces complexités, on obtient une complexité totale pour l'algorithme en O(n^2).

Vérification de la complexité sur des exemples

```r
measure_execution_time <- function(sizes, storage) {
  execution_times <- numeric(length(sizes))
  for (i in seq_along(sizes)) {
    games <- runif(sizes[i], min = 1, max = 100)
    execution_times[i] <- median(microbenchmark(ffd_bin_packing(games, storage),
                                                times = 10)$time)
  }
  return(data.frame(Size = sizes, ExecutionTime = execution_times))
}

sizes <- seq(10, 1000, by = 10)

storage <- 1000

execution_data <- measure_execution_time(sizes, storage)

ggplot(execution_data, aes(x = Size, y = ExecutionTime)) +
  geom_line() +
  geom_point() +
  labs(x = "Taille des données", y = "Temps d'exécution (en microsecondes)",
       title = "Complexité de l'algorithme de bin packing") +
  theme_minimal()
```



On reconnaît aisément une fonction sensiblement proche de la fonction $n \mapsto n^2$.

Implémentation de l'algorithme à l'aide du langage C++

```
#include <Rcpp.h>
using namespace Rcpp;
using namespace std;

#include<vector>
#include <iostream>
#include <algorithm>
#include <numeric>

// [[Rcpp::export]]
std::vector<std::vector<int>> ffd_bin_packing_Rcpp(std::vector<int>& games, int storage)
  {
    sort(games.begin(), games.end(), greater<int>());

    vector<vector<int>> bins;

    for (int game : games) {
        bool fitted = false;
        for (vector<int>& bin : bins) {
            if (accumulate(bin.begin(), bin.end(), 0) + game <= storage) {
                bin.push_back(game);
                fitted = true;
```

```
                break;
            }
        }

        if (!fitted) {
            bins.push_back({game});
        }
    }

    return bins;
}
```

Nous pouvons tester cet algorithme sur plusieurs simulations :

```
games <- sample(1:50, 15, replace = TRUE)
storage <- 100

bins <- ffd_bin_packing_Rcpp(games, storage)

cat("Jeux à stocker:", games, "\n")
```

```
## Jeux à stocker: 38 19 28 3 40 4 23 36 31 45 32 34 33 44 33
```

```
cat("Capacité de stockage de chaque mémoire:", storage, "\n\n")
```

```
## Capacité de stockage de chaque mémoire: 100
```

```
cat("Répartition des jeux dans les espaces mémoire:\n")
```

```
## Répartition des jeux dans les espaces mémoire:
```

```
for (i in seq_along(bins)) {
  cat("Mémoire", i, ":", unlist(bins[[i]]), "\n")
}
```

```
## Mémoire 1 : 45 44 4 3
## Mémoire 2 : 40 38 19
## Mémoire 3 : 36 34 28
## Mémoire 4 : 33 33 32
## Mémoire 5 : 31 23
```

Explication de l'algorithme

Le fonctionnement de cet algorithme en `C++` est assez similaire au fonctionnement de la version `R`. La fonction prend en arguments un vecteur d'entiers `games`, représentant les tailles de chacun des jeux, et un entier `storage`, représentant la taille de chaque mémoire. La fonction renvoit un vecteur de vecteurs d'enteirs, représentant les jeux placés dans chaque mémoire.

On trie les objets à l'aide de la fonction `sort(games.begin(), games.end(), greater<int>())`, puis on initialise un vecteur d'entiers pour stocker les objets placés dans chaque mémoire.

On rentre, par la suite, dans deux boucles : tout d'abord, on itère sur chaque jeu, puis on créée une variable booléenne pour indiquer si le jeu courant a été placé dans un espace mémoire. Ensuite, on itère sur chacun des espaces mémoire.

Une fois dans la deuxième boucle, on vérifie, à l'aide de la fonction `accumulate()`, si la somme des tailles des jeux déjà placés dans la mémoire courante additionnées à la taille du jeu courant, ne dépasse pas la taille de la mémoire. Si tel est le cas, le jeu courant est placé dans la mémoire courante et `fitted` est actualisé avec `TRUE`.

Si le jeu courant n'a pas été placé dans une mémoire existante, un nouvel espace mémoire est créé et le jeu courant est placé dedans, à l'aide de la fonction `bins.push_back({game})`.

Analyse de la complexité

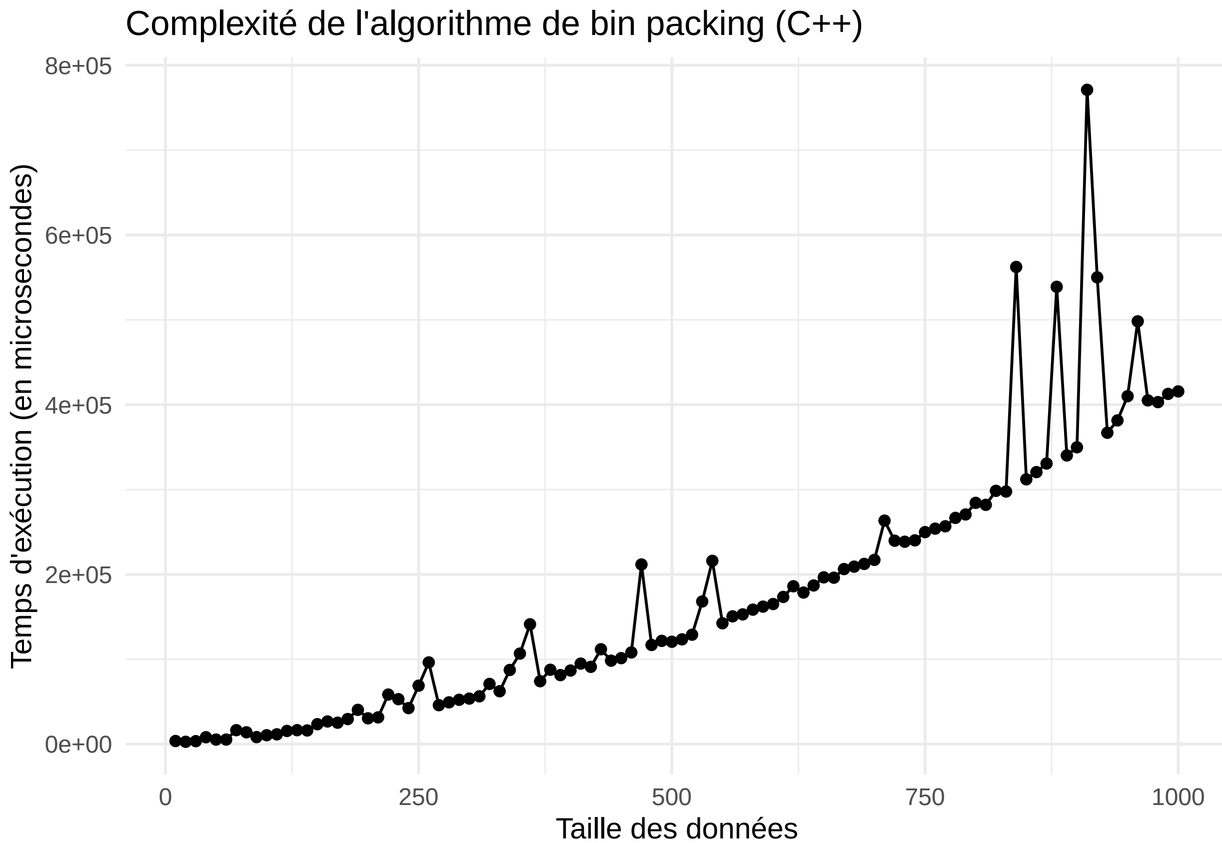
De manière similaire à notre algorithme implémenté en `R`, l'algorithme tri, dans un premier temps, le vecteur des jeux à l'aide de la fonction `std::sort()`, qui a une complexité temporelle en $O(n \log(n))$, où n est la taille du vecteur des jeux.

Ensuite, pour chaque jeu dans le vecteur `games`, l'algorithme parcourt la liste de tous les espaces mémoire existant afin d'essayer de trouver un espace mémoire dans lequel le jeu courant peut être placé. Dans le pire des cas, cette opération a une complexité temporelle de $O(n)$, ce qui donne une complexité totale, pour les deux boucles imbriquées, en $O(n^2)$.

On obtient, ainsi, une complexité totale pour cet algorithme en $O(n^2)$.

Vérification de la complexité sur des exemples

```r
measure_execution_time_cpp <- function(sizes, storage) {
  execution_times <- numeric(length(sizes))
  for (i in seq_along(sizes)) {
    games <- sample(1:100, sizes[i], replace = TRUE)
    execution_times[i] <- median(microbenchmark(ffd_bin_packing_Rcpp(games, storage),
                                                times = 10)$time)
  }
  return(data.frame(Size = sizes, ExecutionTime = execution_times))
}

sizes <- seq(10, 1000, by = 10)

storage <- 1000

execution_data_cpp <- measure_execution_time_cpp(sizes, storage)

ggplot(execution_data_cpp, aes(x = Size, y = ExecutionTime)) +
  geom_line() +
  geom_point() +
  labs(x = "Taille des données", y = "Temps d'exécution (en microsecondes)",
       title = "Complexité de l'algorithme de bin packing (C++)") +
  theme_minimal()
```

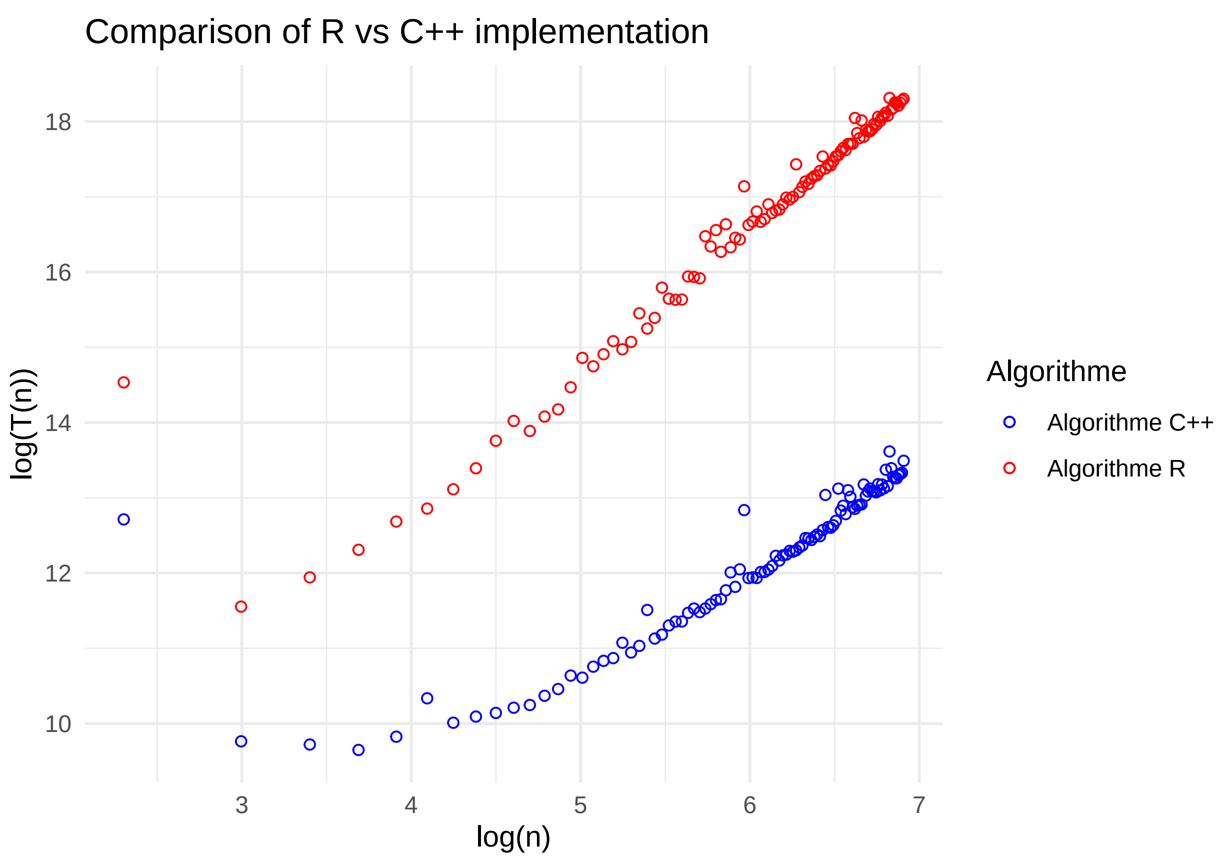


Nous pouvons également voir un tracé sensiblement proche de la fonction $n \mapsto n^2$. Cependant, en comparant les deux graphiques, nous pouvons donc voir que cette fonction est plus rapide que la fonction `R`, ce que nous pouvons vérifier, ci-dessous, en comparant les temps d'exécution sur le même graphique.

Comparaison des temps d'exécution pour le *First-fit-decreasing bin packing*

```
source("StorageOptimisation/R/ffd_bin_packing.R")
sourceCpp("StorageOptimisation/src/ffdBinPacking.cpp")

measure_time_R <- function(n) {
  items <- sample(1:10, n, replace = TRUE)
  bin_size <- 10
  microbenchmark(ffd_bin_packing(items, bin_size), times = 5)
}

measure_time_Cpp <- function(n) {
  items <- sample(1:10, n, replace = TRUE)
  bin_size <- 10
  microbenchmark(ffd_bin_packing_Rcpp(items, bin_size), times = 5)
}

sizes <- seq(10, 1000, by = 10)

times_R <- sapply(sizes, function(n) mean(measure_time_R(n)$time))

times_Cpp <- sapply(sizes, function(n) mean(measure_time_Cpp(n)$time))
```

```
ggplot() +
  geom_point(aes(x = log(sizes), y = log(times_R),
                 color = "Algorithme R"),
             shape = 1) +
  geom_point(aes(x = log(sizes), y = log(times_Cpp),
                 color = "Algorithme C++"),
             shape = 1) +
  labs(x = "log(n)", y = "log(T(n))",
       title = "Comparison of R vs C++ implementation") +
  scale_color_manual(values = c("blue", "red"), name = "Algorithme") +
  theme_minimal()
```



Nous pouvons voir, comme attente, que la complexité temporelle des deux algorithmes évolue de la même manière, *i.e.* de manière quadratique. Toutefois, l'algorithme `C++` est nettement plus rapide que sa version `R`, notamment pour les grandes valeurs de n.

Algorithme naïf

Algorithme optimisé