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)
}
```

Explication de l'algorithme

- **Première étape.** L'algorithme FFD trie d'abord les objets par ordre décroissant de taille, puis il place chaque objet dans le premier bac qui a suffisamment d'espace pour l'accueillir, d'où le terme *First-Fit*.
- **Deuxième étape.** Pour chaque jeu trié, l'algorithme parcourt les bacs existants pour voir s'il peut être placé dans l'un d'entre eux sans dépasser la capacité maximale. S'il trouve un bac où le jeu peut

être placé, il l'ajoute à ce bac. Sinon, il crée un nouveau bac contenant uniquement ce jeu.

- **Troisième étape.** L'algorithme retourne une liste de bacs, où chaque bac contient les jeux qui y ont été placés.

Analyse de la complexité

La complexité de l'algorithme de bin packing dépend de plusieurs facteurs :

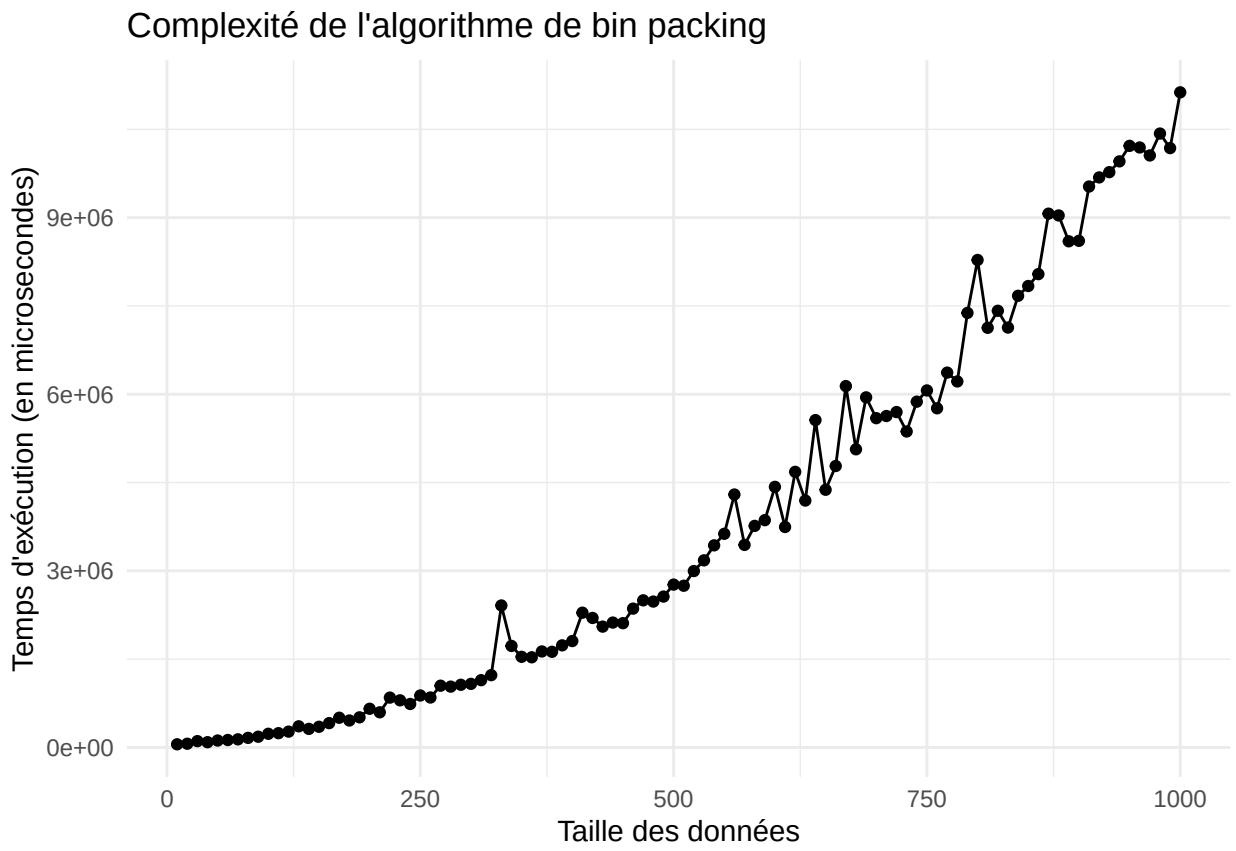
- La phase de tri des jeux peut être réalisé en $O(n \log(n))$ dans le pire des cas, où n est le nombre de jeux.
- Pour chaque jeu, l'algorithme parcourt les bacs existants, ce qui peut prendre jusqu'à $O(m)$ où m est le nombre de bacs déjà créés. Dans le pire des cas, chaque jeu doit être placé dans un nouveau bac, ce qui donne une complexité en $O(n^2)$.
- Enfin, retourner la liste des bacs a une complexité linéaire par rapport au nombre total de jeux, donc $O(n)$.

Ainsi, on peut combiner ces éléments est dire que la complexité totale de cet algorithme est

En combinant ces éléments, on peut dire que la complexité totale de cet algorithme est $O(n \log n + n^2 + m)$. La contribution dominante est $O(n^2)$. Donc, la complexité de cet algorithme peut être approximativement considérée comme **quadratique**.

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)  # Génère des jeux de taille aléatoire
    execution_times[i] <- median(microbenchmark::microbenchmark(ffd_bin_packing(games, storage), times =
  }
  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++

```cpp
#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>& items, int bin_size)
  {
    sort(items.begin(), items.end(), greater<int>());

    vector<vector<int>> bins;

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

```
                break;
            }
        }

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

    return bins;
}
```

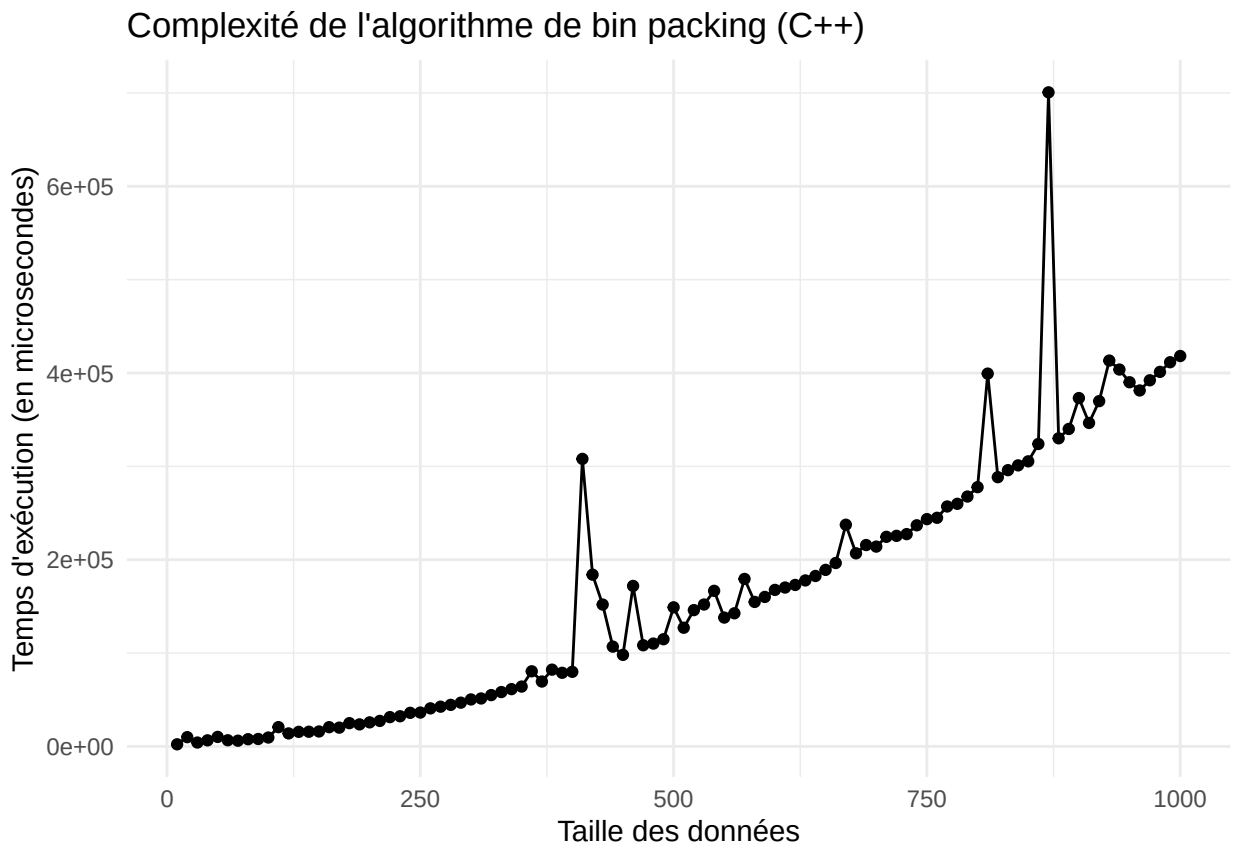
La complexité temporelle de l'étape de tri est de $O(n \log n)$ car le tri d'une liste de n éléments à l'aide d'un algorithme de tri basé sur la comparaison prend $O(n \log n)$ de temps dans le pire des cas. Le tri est nécessaire pour garantir que les éléments sont considérés par ordre décroissant de taille, ce qui est important pour que l'heuristique FFD fonctionne efficacement.

La complexité temporelle de l'étape 2, qui consiste à parcourir les éléments triés et à placer chacun d'entre eux dans la première case qui peut l'accueillir, est de $O(n)$ car l'algorithme parcourt les éléments une fois et effectue une quantité de travail constante pour chaque élément. Plus précisément, pour chaque élément, l'algorithme recherche la première case qui peut lui convenir, ce qui prend $O(1)$ de temps si la case est trouvée. Toutefois, dans le pire des cas, si toutes les cases sont presque pleines et que l'élément en cours est très petit, la recherche peut prendre $O(n)$ de temps parce que l'algorithme peut avoir besoin de vérifier toutes les cases.

Par conséquent, la complexité temporelle globale de l'heuristique FFD est de $O(n \log n) + O(n) = O(n \log n)$.

Vérification de la complexité sur des exemples

```
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)  # Génère des jeux de taille aléatoire
    execution_times[i] <- median(microbenchmark::microbenchmark(ffd_bin_packing_Rcpp(games, storage), ti
  }
  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'al
  theme_minimal()
```

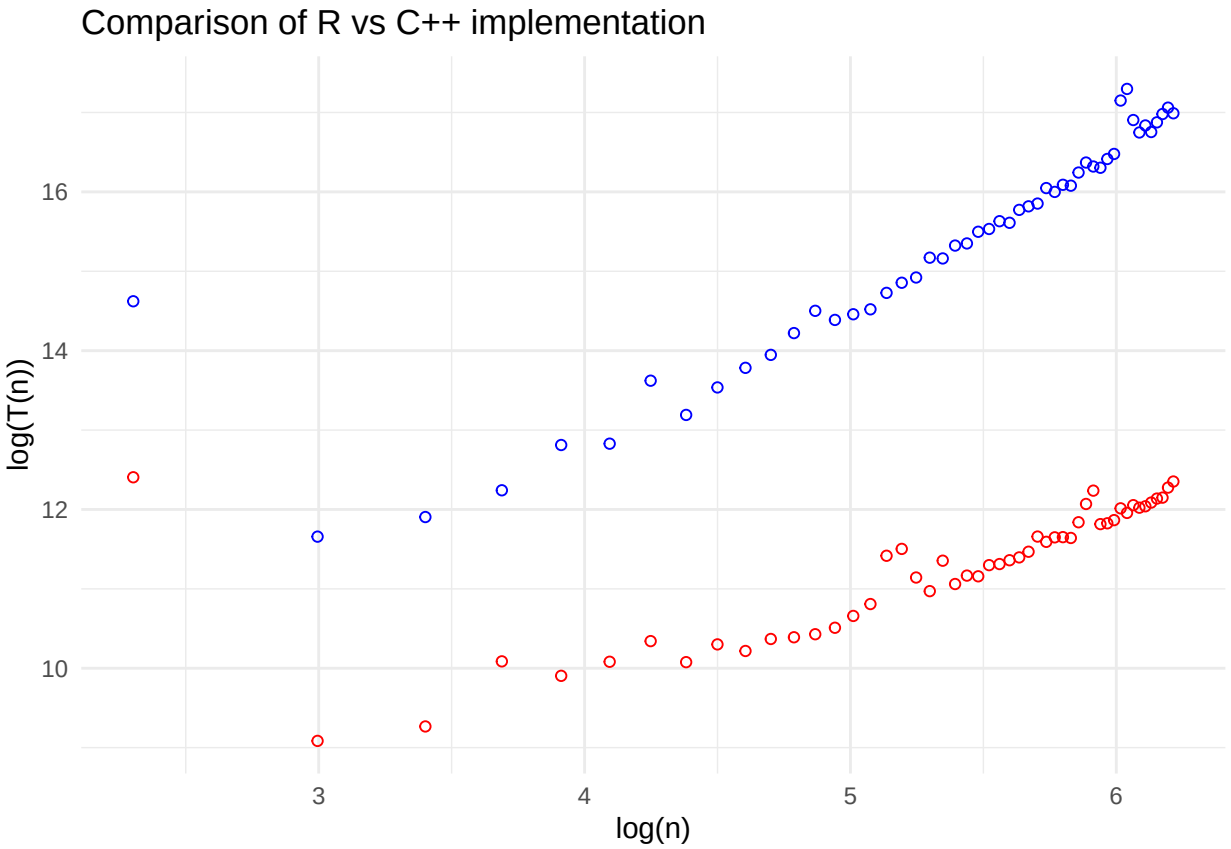


Comparaison des temps d'exécution

```
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, 500, 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 = "blue", shape = 1) +
```

```
  geom_point(aes(x = log(sizes), y = log(times_Cpp)), color = "red", shape = 1) +
  labs(x = "log(n)", y = "log(T(n))", title = "Comparison of R vs C++ implementation") +
  theme_minimal()
```



Algorithme naïf

Algorithme optimisé