Algorithme d'optimisation de stockage

Yoann Bonnet, Victorien Leconte, Hugo Picard

M2 Data Science, 2023 - 2024

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.

Implémentations en `R` et en `C++`

```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)
}
```

```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>& 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 ces deux algorithmes sur plusieurs simulations, ce qui nous renvoie :

```
## Jeux à stocker: 9 6 8 47 18 16 40 8 14 30 1 1 41 12 25

## Capacité de stockage de chaque mémoire: 100

## Version R

## Mémoire 1 : 47 41 12
## Mémoire 2 : 40 30 25 1 1
## Mémoire 3 : 18 16 14 9 8 8 6

## Version C++

## Mémoire 1 : 47 41 12
## Mémoire 2 : 40 30 25 1 1
## Mémoire 3 : 18 16 14 9 8 8 6
```

Explication de l'algorithme

- Dans un premier temps, nous trions les jeux par ordre décroissant de taille. Ainsi, les jeux les plus grands seront placés en premier dans les bac, ce qui 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 stockage 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.

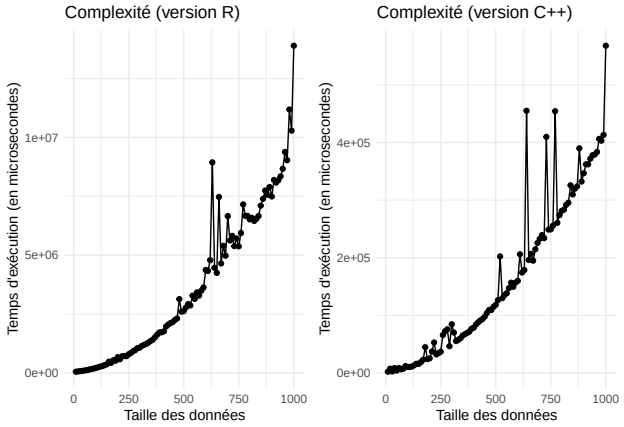
Analyse théorique de la complexité

Les algorithmes `R` et `C++` fonctionnent sensiblement de la même manière, l'analyse de leur complexité est donc équivalente.

- Dans un premier temps, le tri des jeux, avec `sort()` ou `std::sort()`, a une complexité temporelle de $\mathrm{O}(n\log(n))$, dans le pire des cas, n étant la taille du vecteur des jeux.
- L'initialisation de la liste des bacs est une opération constante, donc en $\mathrm{O}(1)$.
- La boucle qui itère sur chaque jeu a une complexité de $\mathrm{O}(n)$. Cependant, pour chaque jeu, nous avons une autre boucle qui itère sur les espaces 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 $\mathrm{O}(n)$. Comme cette boucle est à l'intérieur de la boucle sur les jeux, la complexité totale de ces deux boucles imbriquées est $\mathrm{O}(n^2)$.
- Les opérations à l'intérieur de la boucle sur les espaces mémoire sont toutes des opérations constantes, donc ayant une complexité en $\mathrm{O}(1)$. Cependant, étant à l'intérieur de la boucle sur les espaces mémoire, elles ont une complexité totale en $\mathrm{O}(n)$.
- Enfin, l'opération de création d'un nouvel espace mémoire dans le cas où cela est néccesaire est à l'intérieur de la boucle sur les jeux, elle a donc une complexité totale en $\mathrm{O}(n)$.

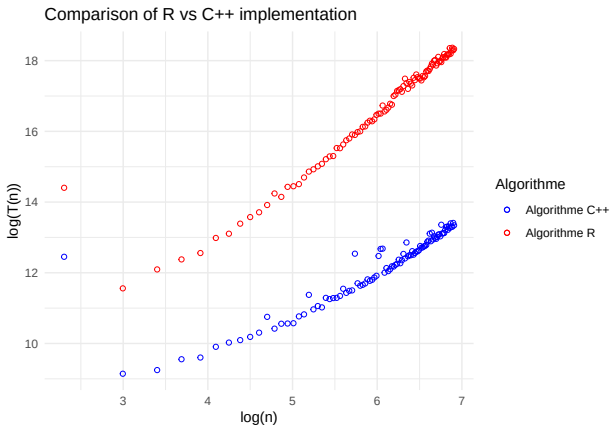
Ainsi, en combinant toutes ces complexités, on obtient une complexité totale pour l'algorithme en $\mathbf{O}(n^2)$.

Vérification de la complexité sur des exemples



On reconnaît aisément une fonction sensiblement proche de la fonction $n \mapsto n^2$. Nous pouvons voir, en comparant les temps d'exécution, que l'algorithme `C++` est plus rapide que la fonction `R`.

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



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.