Algorithme d'optimisation de rangement de jeux dans des espaces mémoire

Yoann Bonnet, Victorien Leconte, Hugo Picard

M2 Data Science, 2023 - 2024

Introduction au problème

Le problème du Bin Packing est un **défi classique de l'optimisation combinatoire** qui consiste à **placer des objets de différentes tailles dans un nombre minimal de conteneurs de capacité fixe**. Cette tâche, qui semble simple au premier abord, présente des implications profondes dans divers domaines, et est classé parmi les problèmes NP-complets.

Une application pratique importante du problème du Bin Packing se trouve dans le domaine du stockage de données, qui peuvent être des fichiers de n'importe quel type, notamment dans le contexte des cartes mémoires. Dans ce scénario, les données doivent être stockées de manière efficace sur des dispositifs de stockage à capacité limitée, tels que des cartes mémoires. Le défi consiste à organiser ces données de manière à minimiser l'utilisation de l'espace de stockage tout en garantissant un accès rapide et efficace aux données. Dans notre cas, nous traiterons l'optimisation du rangement de jeux vidéos dans des carte mémoire.

Notons n le nombre de jeux à ranger, M la taille des cartes mémoire à notre disposition, g_i la taille mémoire du jeu i à ranger ($i = 1, \ldots, n$), $x_{i,j}$ une variable binaire valant 1 si le jeu i est ajouté à la carte mémoire j, sinon 0, et y_j une variable binaire valant 1 si la carte mémoire est utilisée, et 0 sinon.

Le problème se modélise mathématiquement de la façon suivante :

$$\text{Minimiser } z = \sum_j y_j$$

Sous contraintes :

- Chaque jeu doit être placé dans exactement une carte mémoire : $\sum_j x_{ij} = 1 \quad \forall i \in (1, ..., n)$
- La capacité des cartes mémoire ne doit pas être dépassée : $\sum_i g_i \cdot x_{ij} \leq M \quad \forall j$

Pour résoudre ce problème, nous utiliserons trois approches différentes, développées dans la suite, chacune implémentées à l'aide des langages `R` et `C++`.

Algorithme *First-fit-decreasing bin packing*

L'algorithme **First-Fit Decreasing Bin Packing (FFD)** est une méthode heuristique d'optimisation utilisée en informatique et en recherche opérationnelle pour résoudre le problème de **bin packing**. S'agissant de techniques heuristiques, nous ne pouvons pas être certains que la solution obtenue sera une solution optimale ; cependant, elle devra s'en rapprocher avec des délais raisonnables.

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)
}
```

```
#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: 45 49 17 12 7 19 7 13 41 15 41 35 38 38 39
```

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

```
## Version R
```

```
## Mémoire 1 : 49 45
## Mémoire 2 : 41 41 17
## Mémoire 3 : 39 38 19
## Mémoire 4 : 38 35 15 12
## Mémoire 5 : 13 7 7
```

```
## Version C++
```

```
## Mémoire 1 : 49 45
## Mémoire 2 : 41 41 17
## Mémoire 3 : 39 38 19
## Mémoire 4 : 38 35 15 12
## Mémoire 5 : 13 7 7
```

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. On sort de la boucle avec l'instruction `break`.
- Enfin, si le jeu courant n'a pas été placé dans un espace mémoire existant, créons un nouveau espace pour y placer le jeu courant.

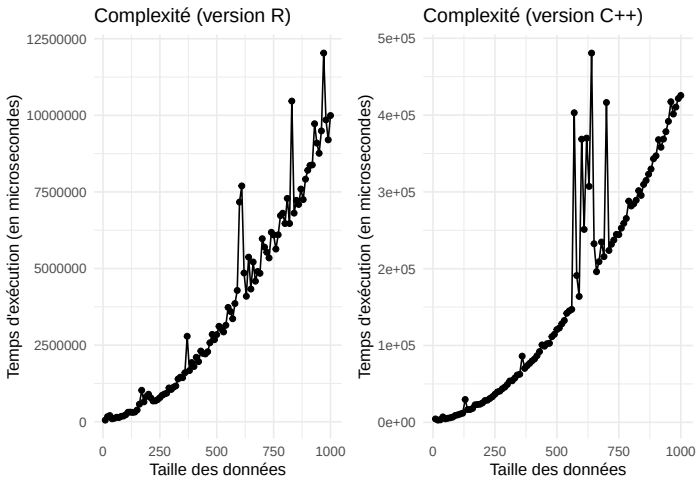
Analyse théorique de la complexité

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 $O(n \log(n))$, 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 $O(1)$.
- La boucle qui itère sur chaque jeu a une complexité de $O(n)$. 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 $O(n)$. 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 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 $O(n)$.

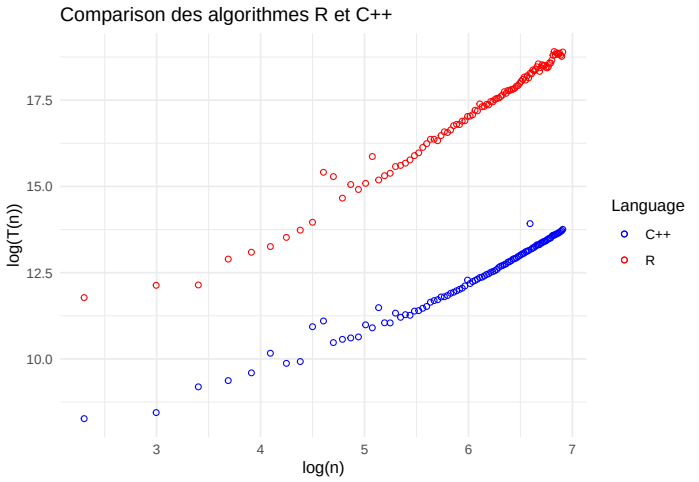
Ainsi, en combinant toutes ces complexités, on obtient une complexité totale pour l'algorithme en $O(n^2)$.

Vérification de la complexité sur des exemples



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.

Algorithme exact naïf

Dans ce modèle, nous utiliserons une approche naïve énumérant toutes les combinaisons possibles afin de sélectionner la meilleure solution. Cette approche est garantie de trouver la solution optimale, mais elle peut être inefficace en termes de temps de calcul, surtout pour des instances de grande taille.

Implémentations en `R` et en `C++`

```
naive_storage <- function(jeux, taille_memoire) {

  generate_permutations <- function(elements) {
    if (length(elements) <= 1) {
      return(list(elements))
    } else {
```

```
    perms <- list()
    for (i in 1:length(elements)) {
      current_element <- elements[i]
      remaining_elements <- elements[-i]
      sub_perms <- generate_permutations(remaining_elements)
      for (perm in sub_perms) {
        perms <- c(perms, list(c(current_element, perm)))
      }
    }
    return(perms)
  }
}

permutations <- generate_permutations(jeux)
best_bins <- list()

memoire_minimale <- Inf

for (permutation in permutations) {
  bins <- list()
  memoires <- rep(taille_memoire, length(jeux))
  nombre_memoires <- 0

  for (i in seq_along(permutation)) {
    jeu <- permutation[i]
    fitted <- FALSE
    for (j in seq_along(bins)) {
      if (sum(bins[[j]]) + jeu <= taille_memoire) {
        bins[[j]] <- c(bins[[j]], jeu)
        memoires[j] <- memoires[j] - jeu
        fitted <- TRUE
        break
      }
    }

    if (!fitted) {
      bins <- c(bins, list(jeu))
      memoires <- c(memoires, taille_memoire - jeu)
      nombre_memoires <- nombre_memoires + 1
    }
  }

  nombre_memoires <- sum(taille_memoire - memoires > 0)

  if (nombre_memoires < memoire_minimale){
    memoire_minimale <- nombre_memoires
    best_bins <- bins
  }

}

jeux_dans_stockages <- list()
for (bin in best_bins) {
  jeux_dans_stockages <- c(jeux_dans_stockages, list(bin))
}

return(list(memoire_minimale = memoire_minimale, jeux_dans_stockages = jeux_dans_stockages))
```

```
}
```

```
#include <Rcpp.h> //to use the NumericVector object
using namespace Rcpp; //to use the NumericVector object
using namespace std;

#include <vector>
#include <algorithm>
#include <numeric>
#include <limits>
#include <iterator>

std::vector<std::vector<int>> generate_permutations(std::vector<int> elements) {
  if (elements.size() <= 1) {
    return {elements};
  } else {
    std::vector<std::vector<int>> perms;
    for (int i = 0; i < elements.size(); i++) {
      int current_element = elements[i];
      vector<int> remaining_elements;
      remaining_elements.reserve(elements.size() - 1);
      for (int j = 0; j < elements.size(); j++) {
        if (j != i) {
          remaining_elements.push_back(elements[j]);
        }
      }
      auto sub_perms = generate_permutations(remaining_elements);
      for (auto perm : sub_perms) {
        perm.insert(perm.begin(), current_element);
        perms.push_back(perm);
      }
    }
    return perms;
  }
}



//' Insertion sort algorithm using C++
//'
//' @param j an unsorted vector of numeric data
//' @param mem an integer corresponding to the memory size
//' @return a sorted vector
//' @export
// [[Rcpp::export]] //mandatory to export the function
std::vector<std::vector<int>> naive_storage_Rcpp(std::vector<int> j, int mem) {
  std::vector<std::vector<int>> permutations = generate_permutations(j);

  int memoire_minimale = numeric_limits<int>::max();
  std::vector<std::vector<int>> best_bins;

  for (const auto& permutation : permutations) {
    std::vector<int> memoires(j.size(), mem);
    int nombre_memoires = 0;
    std::vector<vector<int>> bins(j.size());

    for (int i = 0; i < permutation.size(); i++) {
```

```
      int jeu = permutation[i];
      bool fitted = false;
      for (int k = 0; k < bins.size(); k++) {
        if (accumulate(bins[k].begin(), bins[k].end(), 0) + jeu <= mem) {
          bins[k].push_back(jeu);
          memoires[k] -= jeu;
          fitted = true;
          break;
      }
    }
      if (!fitted) {
        bins[nombre_memoires].push_back(jeu);
        memoires[nombre_memoires] -= jeu;
        nombre_memoires++;
    }
  }

    nombre_memoires = count_if(memoires.begin(), memoires.end(), [mem](int m){
      return m < mem; });

    if (nombre_memoires < memoire_minimale) {
      memoire_minimale = nombre_memoires;
      best_bins = bins;
    }
  }

  best_bins.erase(remove_if(best_bins.begin(),
                            best_bins.end(),
                            [](const vector<int>& v) { return v.empty(); }),
                            best_bins.end());

  return best_bins;
}
```

Nous pouvons tester ces deux algorithmes sur plusieurs simulations, ce qui nous renvoie :

```
## Jeux à stocker: 25 12 44 8 16 50 37 32

## Capacité de stockage de chaque mémoire: 100

## Version R

## Mémoire 1 : 25 12 44 8
## Mémoire 2 : 16 50 32
## Mémoire 3 : 37

## Version C++

## Mémoire 1 : 25 12 44 8
## Mémoire 2 : 16 50 32
## Mémoire 3 : 37
```

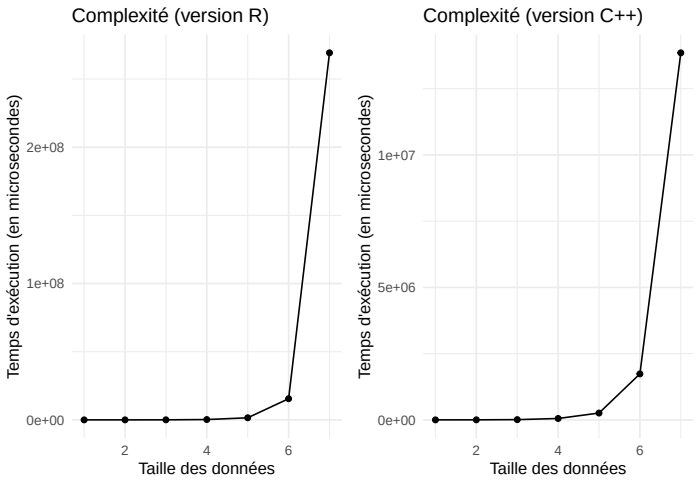
Explication de l'algorithme

- Dans un premier temps, l'algorithme commence par générer toutes les permutations possibles des jeux donnés en entrée à l'aide d'une fonction récursive appelée `generate_permutations`. La complexité associée est en $O(n!)$.
- Ensuite, pour chaque permutation générée, l'algorithme itère pour placer chaque jeu dans une carte mémoire. Il utilise une approche gloutonne en essayant de placer chaque jeu dans la première carte mémoire disponible qui a suffisamment d'espace. Cette étape a une complexité en $O(n.m)$, où m représente le nombre de cartes mémoire utilisé.

- Une fois tous les jeux placés, l'algorithme calcule le nombre de cartes mémoire utilisées en comptant le nombre de cartes restantes avec de l'espace libre, ce qui nous donne une complexité en $O(m)$.
- Pendant le processus, l'algorithme maintient une variable `memoire_minimale` qui stocke le nombre minimal de cartes mémoire nécessaires pour stocker tous les jeux. Cette variable est initialisée à l'infini et est mise à jour si le nouveau nombre de cartes mémoires à utilisé est plus faible que `memoire_minimale`, et cela pour chaque chaque itération sur la permutaion. On stocke également la répartition des jeux dans les cartes mémoires grace à une variable `best_placement` qui est mis à jour en meme temps que `memoire_minimale`. Cette opération est réalisé en temps constant.
- Enfin, l'algorithme retourne `memoire_minimale`, et `best_placement` représentant le nombre minimal de cartes mémoire nécessaires pour stocker tous les jeux, et la répartition de chaque jeux dans les cartes mémoires. Cette opération est réalisé en temps constant.

Ainsi, la complexité temporelle finale de notre algorithme est en $O(m.n!)$, ce qui s'explique par la génération de toutes les permutations possibles des jeux. Cela rend, en pratique, l'algorithme impraticable pour un grand nombre de jeux.

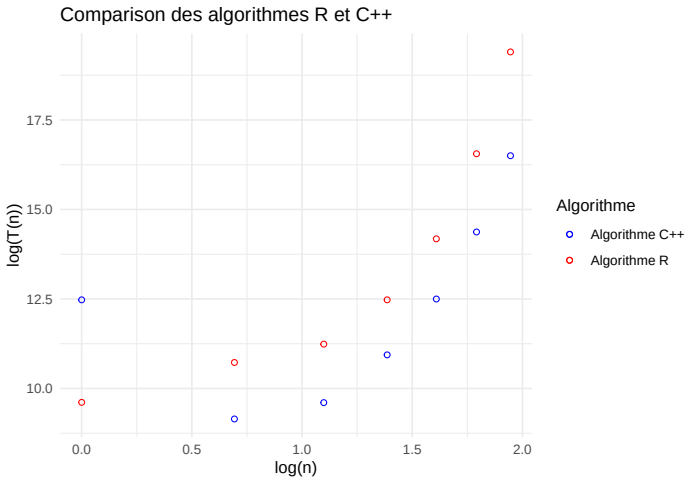
Vérification de la complexité sur des exemples



Nous remarquons ici que, plus le nombre de jeux à ranger dans les cartes mémoires est grand, plus le temps augmente, et cela de manière **exponentielle**. En effet, à partir de 7 jeux à ranger, la complexité en temps explose, et il n'est plus possible d'obtenir un résultat dans un délai raisonnable en ayant 8 jeux ou plus à ranger. C'est la raison pour laquelle nous n'avons pas réalisé de graphiques pour une taille supérieure ou égale à 8.

Nous voyons, toutefois, que le temps d'exécution est plus faible pour la version `C++`, ce qui confirme nos attentes.

Comparaison des temps d'exécution pour l'algorithme naïf



Nous pouvons voir que les deux versions des algorithmes sont assez semblables en termes de temps. Cependant, à taille égale, l'algorithme R est plus gourmand, ce qui était attendu.

Algorithme optimisé

Algorithme optimisé

Dans ce modèle, nous utiliserons une approche exacte optimisée. En effet, on utilise l'algorithme branch and bound qui explore un arbre de manière sélective pour trouver la solution optimale. Cet algorithme utilise une borne inf permettant d'élaguer les branches qui ne peuvent pas contenir la solution optimale.

Implémentations en R et en C++

```r
branch_and_bound <- function(jeux, m) {
  n <- length(jeux)
  jeux <- sort(jeux, decreasing = TRUE)
  best_sol <- n
  bfd_sol <- length(bfd(jeux, m))
  if (bfd_sol < best_sol) {
    best_sol <- bfd_sol
  }
  stack <- list()
  stack[[1]] <- list(level = 0, items = jeux, bins = numeric(n), num_bins = 0, bound = borne_inf(jeux, m))
  while (length(stack) > 0) {
    node <- stack[[length(stack)]]
    stack <- stack[-length(stack)]

    if (length(node$items) == 0) {
      next
    }
    # Exécuter l'algorithme BFD
    bfd_node <- bfd(node$items, m)
    if (length(bfd_node) < best_sol) {
      best_sol <- length(bfd_node)
    }
    new_bins <- node$bins
    new_num_bins <- node$num_bins
    for (i in 1:length(node$items)) {
      new_bins[i] <- new_bins[i] + node$items[1]
      new_num_bins <- new_num_bins + 1
      if (new_num_bins + borne_inf(node$items[-1], m) >= best_sol) {
        new_bins[i] <- new_bins[i] - node$items[1]
        new_num_bins <- new_num_bins - 1
        break
      }
      new_items <- node$items[-1]
      new_bound <- borne_inf(new_items, m)
      new_node <- list(level = node$level + 1, items = new_items, bins = new_bins, num_bins = new_num_bins, b
      stack <- c(stack, list(new_node))
      new_bins[i] <- new_bins[i] - node$items[1]
      new_num_bins <- new_num_bins - 1

      if (node$bound >= best_sol) {
        next
      }
      if (node$num_bins + node$bound >= best_sol) {
```

```
        next
      }
      if (node$num_bins + node$bound < best_sol) {
        best_sol <- node$num_bins + node$bound
      }
    }
  }
  return(best_sol)
}
```

Explication de l'algorithme

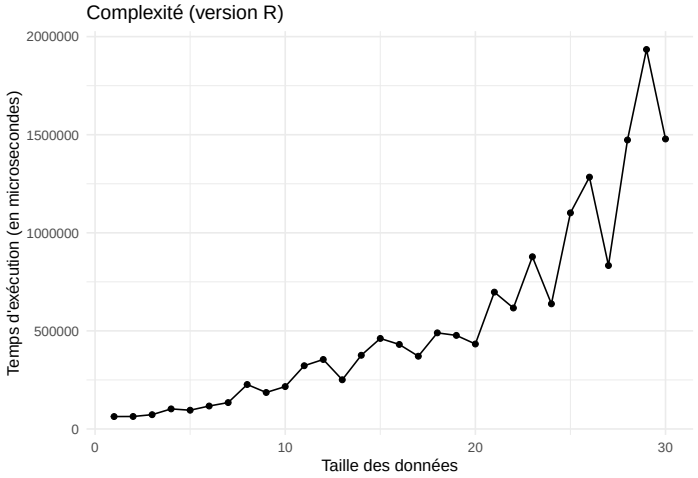
- La fonction `borne_inf` calcule la borne inférieure du nombre de bacs nécessaires pour stocker tous les éléments. Ce qui nous servira pour éliminer des solutions.
- La fonction `bfd` implémente l'algorithme glouton Best Fit Decreasing similaire au `ffd` décris précédemment.
- La fonction `branch_and_bound` implémente l'algorithme de Branch and Bound pour résoudre le problème de Bin Packing. Elle prend en entrée une liste d'éléments jeux et la taille maximale d'un bac m, et renvoie le nombre minimal de bacs nécessaires pour stocker tous les éléments. Elle commence par trier les éléments par ordre décroissant de taille, puis initialise la meilleure solution trouvée à la taille maximale de la liste d'éléments. Ensuite, elle appelle la fonction `bfd` pour obtenir une solution initiale, et met à jour la meilleure solution trouvée si la solution initiale est meilleure. Puis, elle utilise une pile pour explorer récursivement toutes les solutions possibles en utilisant l'algorithme de Branch and Bound. À chaque itération, elle calcule la borne inférieure du nombre de bacs nécessaires pour stocker les éléments restants, et utilise cette borne pour élague les solutions qui ne peuvent pas être meilleures que la meilleure solution trouvée. Enfin, elle renvoie la meilleure solution trouvée.

Analyse théorique de la complexité

- La fonction `borne_inf` parcourt tous les jeux, la complexité associée est $\mathrm{O}(n)$.
- La fonction `bfd` parcourt la liste des jeux et pour chaque jeu, elle parcourt la liste des stockages pour trouver le meilleur ajustement, donc la complexité associée est $\mathrm{O}(n^2)$.
- La fonction `branch_and_bound` a une complexité temporelle qui dépend de la taille de l'arbre de recherche. Dans le pire des cas, l'algorithme peut avoir une complexité temporelle exponentielle, car chaque nœud de l'arbre peut avoir jusqu'à n fils, où n est le nombre d'éléments dans la liste jeux. Cependant, la complexité réelle de l'algorithme dépend de la fonction de borne utilisée pour élaguer l'arbre de recherche. Si la fonction de borne est suffisamment précise, l'algorithme peut avoir une complexité temporelle polynomiale. Ce qui donne une complexité dans le pire des cas de $\mathrm{O}(n^n)$, n étant le nombre de jeux.

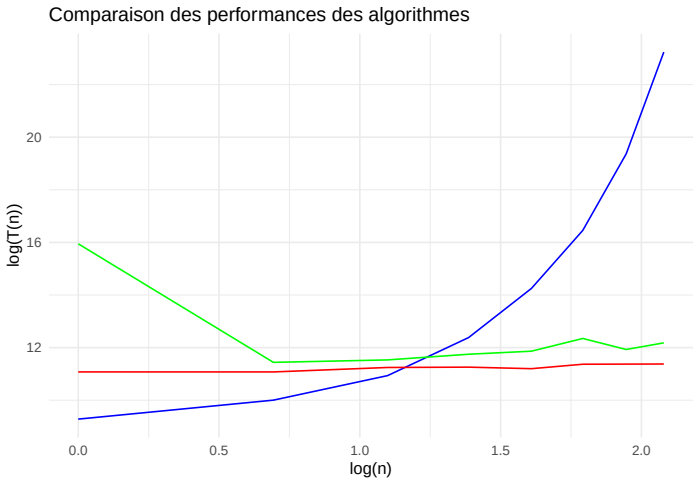
Ceci nous donne donc au total une complexité dans le pire des cas de $\mathrm{O}(n^n)$, c'est-à-dire exponentielle.

Vérification de la complexité sur des exemples



Nous remarquons ici que, plus le nombre de jeux à ranger dans les cartes mémoires est grand, plus le temps augmente, mais de manière moins rapide que pour la fonction naïve. Cependant, la complexité ne semble pas exponentielle comme vu en théorie. Cela peut s'expiquer par le fait que de nombreuses branches seront élaguées grâce à la borne minimale.

Comparaison entre les différents algorithmes



Nous pouvons voir, sur le graphique ci-dessus, que l'algorithme naïf exact est l'algorithme ayant la complexité temporelle la plus élevée, cette dernière s'envolant très rapidement avec n. Les algorithmes de *FFD Bin Packing* et *Branch & Bound* ont, quant à eux, une complexité temporelle relativement proche, ce qui est apparaît comme cohérent.