



# Algorithmique et Complexité

## TD 6/7 – Problème d'emballage

CentraleSupélec – Gif

ST2 – Gif



## Introduction

L'objectif de ce TD est de comparer différentes heuristiques d'une même famille pour résoudre/approcher un problème NP-complet.



# Plan

- 1 Analyse d'un problème
- 2 Algorithmes de résolution



# Plan

- 1 Analyse d'un problème
- 2 Algorithmes de résolution



## Présentation

Une entreprise met à disposition de ses usagers des services de *Cloud Computing*. Ce sont de puissants ordinateurs accessibles sur réservation pour réaliser des calculs très consommateurs en CPU. Chaque client a la possibilité de réserver du temps de calcul pour des tâches. Chaque tâche sera traitée sur une machine unique.

La société dispose de  $M$  serveurs principaux disponibles  $B$  minutes par jour. Elle peut au besoin louer des serveurs complémentaires mais souhaite, dans la mesure du possible, ne pas y avoir recours. Tous les soirs, elle doit affecter chacune des  $N$  tâches de ses clients à une machine. Les tâches seront ensuite exécutées le lendemain.

L'entreprise souhaite concevoir un logiciel permettant de déterminer s'il est possible d'affecter chaque tâche à une machine sans avoir recours à la location de serveurs de calcul supplémentaires.



## Question 1

Donnez une définition formelle de ce problème de décision (donnez les entrées et la question du problème).



## Question 1 : correction

Le problème que l'on souhaite résoudre est le problème du Bin Packing ( le problème de **décision**).

### **BIN PACKING**

#### **Entrée :**

- un ensemble  $O$  de  $N$  objets de taille  $o_i \in \mathbb{N}$
- $B \in \mathbb{N}$  une taille de sac
- $M$  un nombre de sacs disponibles

**Question :** Existe-t-il une affectation des  $N$  objets dans les  $M$  sacs, telle que la somme des tailles des éléments de chaque sac est inférieur ou égale à  $B$ .



## Question 1 : correction

Formellement, on peut voir l'affectation de plusieurs manières :

- une fonction  $f$  qui associe à chaque numéro d'objet un numéro de sac ( $f : [1, M] \rightarrow [1, M]$ ) vérifiant :  
$$\forall j \in [1, M]. \sum_{i \in f^{-1}(\{j\})} o_i \leq B ;$$
- ...



## Question 2

Montrez que le problème formalisé précédemment est *NP*-Complet. Pour cela, on pourra s'appuyer sur le problème de Partition qui est également *NP*-Complet.

### **PARTITION**

**Entrée** : Soit  $E$  un ensemble d'entiers positifs.

**Question** : Existe-t-il une partition de  $E$  en deux sous-ensembles  $E_1$  et  $E_2$  telle que  $\sum_{e_i \in E_1} e_i = \sum_{e_j \in E_2} e_j$



## Question 2 : correction

La classe P est la classe des problèmes **de décision** (auxquels on répond par oui ou non) pour lesquels il existe un algorithme de résolution ayant une complexité en temps polynomiale.

La classe NP est la classe des problèmes de décision pour lesquels il existe un algorithme de **vérification d'une solution** de complexité polynomiale

Pour monter qu'un problème est NP-complet, il suffit de montrer qu'il est NP et qu'il existe une réduction polynomiale d'un problème NP-complet vers ce problème.



## Question 2 : correction

On commence donc par montrer que le problème est dans  $NP$ .  
Pour cela on donne un algorithme qui vérifie en temps polynomial qu'une affectation est bien une solution de l'instance.

```
# un objet est un entier
O = [o1, o2, ..., oN]
M =
B =

Aff = bin_packing()
# Aff est une liste de listes :
# Aff est une liste de sacs, un sac est une liste d'objets

def verify(O, Aff, B, M):
```



## Question 2 : correction

En Python et en  $\mathcal{O}(n \log n)$  :

```
def verify(O, Aff, B, M):
    if len(Aff) > M: # M sacs suffisent
        return False
    # Tous les objets sont ranges
    Aff_flattened = [val for bin in Aff for val in bin]
    if sorted(O) != sorted(Aff_flattened):
        return False
    # Aucun sac ne déborde
    for bin in Aff:
        if sum(bin) > B:
            return False
    return True
```



## Question 2 : correction

Ensuite on montre qu'on peut réduire un problème *NP*-Complet (ici Partition) à un problème de Bin Packing.



## Question 2 : correction

Ensuite on montre qu'on peut réduire un problème *NP*-Complet (ici Partition) à un problème de Bin Packing.

A partir de Partition, on construit l'instance du problème de Bin Packing suivante :

- $O = E$
- $M = 2$
- $B = \frac{1}{2} \times \sum_{e \in E} e$



## Question 3

Définissez le problème sous forme de problème d'optimisation (donnez les entrées et la question du problème).



## Question 3 : correction

Pour formaliser un problème d'optimisation, on doit expliciter une fonction *score* à optimiser (minimiser ou maximiser) :

### **BIN PACKING**

**Entrée :**

- un ensemble  $O$  de  $N$  objets de taille  $o_i \in \mathbb{N}$
- $B \in \mathbb{N}$  une taille de sac

**Question :** trouver une affectation  $f : [1, N] \rightarrow \mathbb{N}$  vérifiant (on note  $img(f)$  l'ensemble image de  $f$ ) :

$$\forall j \in img(f), \sum_{i \in f^{-1}(j)} o_i \leq B$$

minimisant le nombre de sacs c'est-à-dire :

$$score(f) = card(img(f))$$



## Question 4

Pourquoi pense-t-on qu'il n'existe pas d'algorithme en temps polynomial pour résoudre ce problème d'optimisation ?



## Question 4 : correction

Si on résout le problème d'optimisation en temps polynomial, on répond au problème de décision en temps polynomial également. Il suffit de comparer le nombre de sacs optimal  $M^*$  en sortie du problème d'optimisation avec le nombre de sacs  $M$  en entrée du problème de décision. On répond "oui" au problème de décision si et seulement si  $M \geq M^*$ .

Trouver un algorithme polynomial à un problème d'optimisation associé à un problème de décision  $NP$ -Complet est impossible sous le postulat de  $\mathcal{P} \neq \mathcal{NP}$ .



# Plan

- 1 Analyse d'un problème
- 2 Algorithmes de résolution**



## Présentation

Le problème d'optimisation que l'on souhaite résoudre est le problème du Bin-Packing, vu à l'exercice précédent :

### **BIN PACKING**

#### **Entrée :**

- un ensemble  $O$  de  $N$  objets de taille  $o_i \in \mathbb{N}$
- $B \in \mathbb{N}$  une taille de sac

**Question :** trouver une affectation  $f : [1, N] \rightarrow \mathbb{N}$  vérifiant (on note  $img(f)$  l'ensemble image de  $f$ ) :

$$\forall j \in img(f), \sum_{i \in f^{-1}(j)} o_i \leq B$$

minimisant le nombre de sacs c'est-à-dire :

$$score(f) = card(img(f))$$



## Question 1-2-3

- Proposez un algorithme **glouton** pour résoudre ce problème. Il en existe plusieurs.
- Écrivez cet algorithme (en pseudo-code ou en python) et donnez sa complexité.
- Testez cet algorithme sur l'instance suivante où la taille des sacs est 10 et la taille des éléments est :

4, 4, 5, 5, 5, 4, 4, 6, 6, 2, 2, 3, 3, 7, 7, 2, 2, 5, 5, 8, 8, 4, 4, 5



## Question 1-2-3 : correction

Rappel : avec un algorithme **glouton**, on ...



## Question 1-2-3 : correction

Rappel : avec un algorithme **glouton**, on ...

... construit la solution petit à petit sans jamais remettre en cause un choix qui a été fait : une fois un objet ajouté à un sac on ne le changera jamais de sac.



## Question 1-2-3 : correction

*First Fit (FF)* : ajouter les objets, les uns après les autres, dans le premier sac possible (celui qui porte le plus petit numéro et dans lequel il y a suffisamment de place). S'il n'existe aucun sac avec suffisamment de place disponible, en ouvrir un nouveau.

```
def FirstFit(O, B):
```



## Question 1-2-3 : correction

```
def FirstFit(O, B):
    Aff = []
    for o in O:
        for bin in Aff:
            if sum(bin) + o <= B :
                bin.append(o)    # si on trouve un sac ou ca rentre
                break
        else:
            Aff.append([o])    # sinon on cree un nouveau sac

    return Aff
```



## Question 1-2-3 : application FF

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5



## Question 1-2-3 : application FF

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5

FF : 13 sacs



## Question 1-2-3 : correction

*BestFit (BF)* : on met un élément dans le sac qui est le plus rempli et dans lequel il y a suffisamment de place ; on continue jusqu'à l'épuisement des éléments.

```
def BestFit(O, B):
```



## Question 1-2-3 : correction

```
def BestFit(O, B):
    Aff = []
    for o in O:
        idx_bin = None
        min_space = B
        for idx in range(len(Aff)):
            space = B - sum(Aff[idx])
            # it may become the new best fit
            if o <= space and space < min_space :
                idx_bin = idx
                min_space = space
        if idx_bin == None:
            # if we found no bin for our object, use a new bin
            Aff.append([o])
        else:
            Aff[idx_bin].append(o)

    return Aff
```



## Question 1-2-3 : application BF

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5



## Question 1-2-3 : application BF

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5

FF : 13 sacs

BF : 12 sacs



## Question 1-2-3 : correction

La complexité des algorithmes FF et BF est  $\mathcal{O}(N^2)$ . En effet, la boucle extérieure est exécutée exactement  $N$  fois et, dans le pire des cas, on utilisera/parcourra les  $N$  sacs (boucle intérieure). C'est le cas lorsque l'on a essentiellement de très gros objets.

Si en pratique les objets sont petits devant la taille du sac, la complexité au pire sera une borne relativement loin de la réalité.



## Question 1-2-3 : correction

Les deux algorithmes ci-dessus peuvent être améliorés en ayant préalablement trié les objets par ordre décroissant de taille. On obtient les algorithmes *First Fit Decreasing (FFD)* et *Best Fit Decreasing (BFD)*.

La complexité au pire de ces algorithmes ne change pas car le coût d'un tri  $\mathcal{O}(N \log(N))$  est inférieur à la complexité du placements  $\mathcal{O}(N^2)$ .



## Question 1-2-3 : application FFD et BFD

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5  
8 8 7 7 6 6 5 5 5 5 5 5 4 4 4 4 4 4 3 3 2 2 2 2



## Question 1-2-3 : application FFD et BFD

$B = 10$

4 4 5 5 5 4 4 6 6 2 2 3 3 7 7 2 2 5 5 8 8 4 4 5  
8 8 7 7 6 6 5 5 5 5 5 5 4 4 4 4 4 4 3 3 2 2 2 2

FF : 13 sacs

BF : 12 sacs

FFD : 11 sacs

BFD : 11 sacs



## Question 1-2-3 : correction

Approche en  $\mathcal{O}(N)$  :

*NextFit (NF)* : on met l'élément dans le dernier sac ouvert quand il y reste suffisamment de place ; sinon on ouvre un nouveau sac.

```
def NextFit(O, B):
```



## Question 1-2-3 : correction

```
def NextFit(O, B):
    Aff = [[]]
    for o in O:
        # verifier s'il reste de la place
        # dans le dernier sac
        if sum(Aff[-1]) + o > B :
            # ouvrir un nouveau sac
            # et y ajouter l'objet
            Aff.append([o])
        else :
            # ajouter l'objet au dernier sac
            Aff[-1].append(o)

    return Aff
```



## Question 1-2-3 : correction

Démo. Les résultats :

- NF : 14 sacs
- FF : 13
- FFD : 11 ! optimal (car la somme des objets est égale à 110)
- BF : 12
- BFD : 11 ! optimal



## Question 1-2-3 : correction

Démo. Les résultats :

- NF : 14 sacs
- FF : 13
- FFD : 11 ! optimal (car la somme des objets est égale à 110)
- BF : 12
- BFD : 11 ! optimal

À titre indicatif, voici une instance sur laquelle le BFD et FFD en trouve pas l'optimal. Sac de taille 13, et les éléments (regroupés optimalement en 2 sacs) :  $[3,5,5]$   $[2,2,2,7]$ .



## Question 4

Essayez d'estimer les performances de votre algorithme : de combien de fois au pire une solution renvoyée par votre algorithme dépasse le nombre minimal de serveurs.



## Question 4 : correction

Faire la preuve FF d'abord :

Pour cette estimation il est pratique de supposer que chaque sac peut contenir le poids 1. Le poids des objets est normalisé en référence à la capacité des sacs :  $p_i \leftarrow \frac{o_i}{B}$ .

**Combien de sacs *au moins* faudra-t-il ?**



## Question 4 : correction

Faire la preuve FF d'abord :

Pour cette estimation il est pratique de supposer que chaque sac peut contenir le poids 1. Le poids des objets est normalisé en référence à la capacité des sacs :  $p_i \leftarrow \frac{o_i}{B}$ .

**Combien de sacs *au moins* faudra-t-il ?**

La réponse est immédiate,  $\lceil \sum_{i=1}^n p_i \rceil$ . On ne pourra jamais avoir moins que  $\lceil \sum_{i=1}^n p_i \rceil$  sacs. Cette réponse est notamment vraie pour n'importe quel algorithme, pas uniquement FF.



## Question 4 : correction

**Combien de sacs *au plus* faudra-t-il ?**



## Question 4 : correction

### Combien de sacs *au plus* faudra-t-il ?

**Observation** : il y a au plus un sac non-vide dans lequel l'espace disponible est supérieur ou égal à  $\frac{1}{2}$ .

**Preuve** : Supposons que nous avons deux sacs  $i, j, i < j$  qui sont plus qu'à moitié vides. Nous constatons que les éléments dans le sac  $j$  auraient pu avoir de la place dans le sac  $i$  (dont le numéro est inférieur) et FF aurait dû les mettre dans le sac  $i$ . Notre observation est donc vraie.

Nous avons donc tous les sacs (sauf, peut-être, un) remplis au moins à moitié. Par conséquent, nous n'avons jamais besoin de plus de  $\lceil 2 \sum_{i=1}^n p_i \rceil$  sacs.



## Question 4 : correction

**Conclusion** : Nous sommes sûrs qu'une solution avec FF ne dépasse jamais deux fois la solution optimale :  $k^* \leq k < 2k^*$  où  $k$  notre solution et  $k^*$  la solution optimale.