SIMA Daniel DM Info/EEA 3800992
SADYKOVA Karima DM Info/EEA 28610100

SCIENCES SORBONNE UNIVERSITÉ

Projet d'algorithmique - Confitures

Ce projet traite d'un problème d'ordonnancement de bocaux de confiture, cherchant à minimiser le nombre de bocaux à utiliser de la manière la plus optimale possible. Pour cela nous allons être confrontés à plusieurs algorithmes, dont 3 principaux. L'algorithme I est un algorithme récursif possédant la plus grande complexité dans ce projet, s'ensuit l'algorithme II qui a la particularité d'utiliser une matrice et une matrice de tableaux en mettant l'accent sur la complexité spatiale en plus de la complexité temporelle. Enfin l'algorithme III est un algorithme glouton, ceci va nous permettre également de nous familiariser avec la notion de système de capacité glouton-compatible.

Finalement, nous allons analyser, sous différents graphes ces données et nous allons traiter également la notion de complexité pseudo-polynomiale.

Partie théorique

2.1 Algorithme I:

Question 1)

a)

- m(0, i) = 0, S = 0, m(S) = m(0) = 0
- m(s, 0) = +inf, S=s (avec S >= 1), m(S) = +inf
- m(s, i) = +inf, S=s (avec S < 0), m(S) = +inf

Donc, m(S) = min{m(S,i), m(S, i-1), … , m(S, 1), m(S, 1)} = m(S, i) = m(S, k)

b) Par récurrence faible:

Soit la propriété P(i) : "Pour $\forall i \in \{1, \dots, k\}$ $m(s,i) = \begin{cases} 0 & \text{si } s = 0 \\ \min\{m(s, i-1),\ m(s - V[i], i) + 1\} & \text{sinon} \end{cases}$ "

Montrons par récurrence faible sur i que la propriété est vraie et renvoie le nombre minimal de bocaux.

Base : Pour i = 1 on a 3 cas:

- S = 0 => ∀i m(0, i) = 0 (cas de base vérifié)
- S < 0 => m(S, 1) = min{m(S, 0), m(S - V[1], 1)+1} = min{ inf, inf + 1 } = inf (cas de base vérifié)
- S > 0 => m(S, 1) = min{m(S, 0), m(S - V[1], 1)+1} = min{ inf, m(S - V[1], 1) + 1 } = m(S - V[1], 1) + 1 (cas de base vérifié)

  Selon la définition de V (V[1] = 1) on a m(S, 1) = m(S-1, 1) + 1

  Si on cherche le nombre minimum de bocaux pour S - 1, alors pour trouver m(S, 1) il nous faut ajouter un bocal en plus de m(S - 1, 1)

Donc, P(1) est vérifiée.

Induction : Supposons que P(i) est vérifiée. Montrons que P(i) => P(i + 1)

Pour i +1 on considère 3 cas:

- S = 0 => ∀i m(0, i+1) = 0
- S < 0 => m(S, i+1) = min{m(S, i), m(S - V[i + 1], i + 1)+1} = min{ inf, inf + 1 } = inf
- S > 0 => m(S, i+1)

  = min{m(S, i), m(S-V[i+1], i+1)+1}

  = min{m(S, i), min{m(S-V[i+1], i), m(S-2V[i+1], i+1)+1}+1}

  = min{m(S, i), min{m(S-V[i+1], i), min{m(S-2V[i+1], i), m(S-3V[i+1], i+1)+1}+1}+1}

  = min{m(S, i), min{m(S-V[i+1], i), min{m(S-2V[i+1], i), min{m(S-3V[i+1], i), m(S-4V[i+1], i+1)+1}+1}+1}+1}

  avec m(S, i), m(S-V[i+1], i), m(S-2V[i+1], i), m(S-3V[i+1], i) vérifiées par hypothèse de récurrence.

  Si on continue à développer m(S-4V[i+1], i+1) on va arriver à un moment ou S-p*V[i+1] < 0 (avec p ∈ N). Selon la définition m(S, i) = + infini ∀i ∈ {1, …, k}, ∀S < 0

Donc, la fonction minimum va prendre l'expression de gauche : m(S-(p-1)*V[i + 1], i).

Or selon l'hypothèse de récurrence, m(S-(p-1)V[i + 1], i) renvoie le nombre minimum de bocaux pour S.

Donc, m(S, i+1) = min{m(S, i), m(S-V[i+1], i+1)+1} renvoie bien le nombre minimum de bocaux pour S.

Conclusion : On a montré par récurrence faible que pour ∀i ∈ {1, … , k} la relation :

$$m(s,i) = \begin{cases} 0 & \text{si } s = 0 \\ \min\{m(s, i-1),\ m(s - V[i], i) + 1\} & \text{sinon} \end{cases}$$

renvoie le nombre minimum de bocaux pour S.

Question 2)

algorithme AlgoRecursif(S: entier, k: entier, V: tableau de k entiers): entier

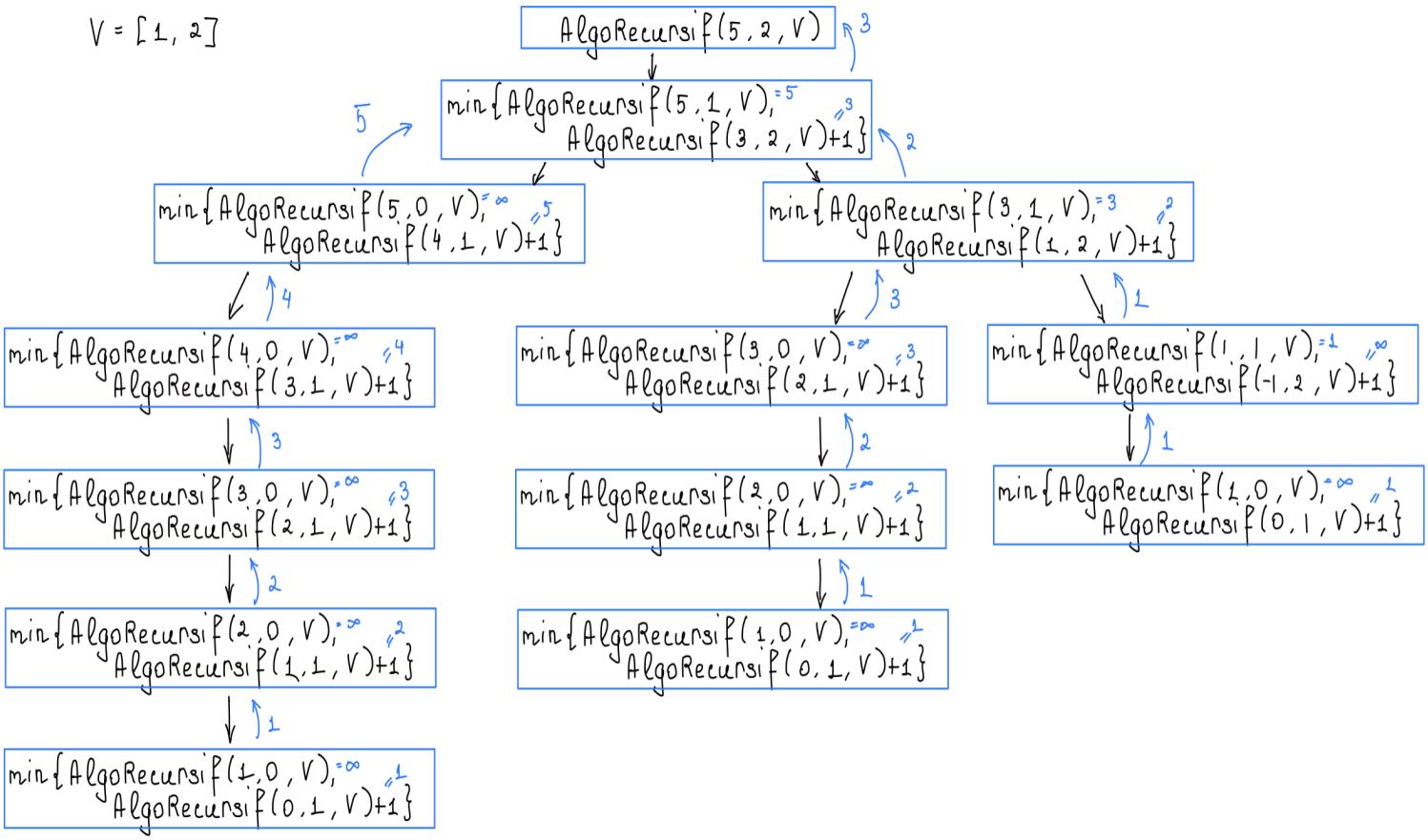
si (S = 0) **alors retourner** 0

sinon si (S < 0) ou (k = 0) **alors retourner** infini

sinon retourner min{AlgoRecursif(S, k-1, V), AlgoRecursif(S-V[k], k, V)+1}

L'appel initial de l'algorithme est AlgoRecursif(S, k, V) avec S la quantité de confiture totale, k taille du tableau du système de capacités (V) et V les valeurs initiales du système de capacité choisies.

Question 3)



Question 4)

Dans l'arbre précédent, la valeur m(1, 1) est calculée 3 fois.
Si S est impair et que k=2 avec V=[1, 2] on aura le nombre d'occurrences de m(1, 1) = S/2+1 avec / la division entière.
Dans les mêmes conditions, si S est pair, le nombre d'occurrences de m(1, 1) = S/2.
On remarque une corrélation, le nombre d'occurrences de m(1, 1) est égal au nombre de bocaux minimum pour m(S).

2.2 Algorithme II:

Question 5)

a) Les cases du tableau M peuvent être remplies des S petits jusqu'à des S grands en utilisant tous les i pour une valeur donnée S. En plaçant les S sur les lignes et les i sur les colonnes, on remplira ligne par ligne, c'est-à-dire en partant de i=0 jusqu'à i=k pour chaque s allant de 0 jusqu'à S.

En faisant cela on pourra utiliser les valeurs déjà remplies auparavant dans M.

b) **algorithme AlgoOptimise(**S: entier, k: entier, V: tableau de k entiers): entier

```
création d'une matrice M de taille S*k avec toutes les valeurs initialisées à l'infini
c, i: entiers
pour c de 0 à S faire
    pour i de 0 à k faire
        si (c = 0) alors
            M[c][i] = 0
        fin si
        sinon si (i = 0) alors
            on passe directement au prochain tour de boucle
        fin sinon si
        sinon alors
            si ((c - V[i]) < 0) alors
                M[c][i] = M[c][i-1]
            fin si
            sinon alors
                M[c][i] = min{M[c][i-1], M[c-V[i]][i]+1}
            fin sinon
        fin alors
    fin pour
fin pour
retourner M[S][k]
```

c) La complexité temporelle de l'algorithme **AlgoOptimise(S, k, V)** est en O(S*k) car on parcourt tous les k (de 0 jusqu'à k en incrémentant à chaque fois de 1) pour chaque S (de 0 jusqu'à S en incrémentant de a chaque fois de 1).

La complexité spatiale de l'algorithme **AlgoOptimise(S, k, V)** est également en O(S*k) car on utilise une matrice M auxiliaire pour sauvegarder des valeurs déjà calculées, or cette matrice a S*k cases, d'où la complexité spatiale en O(S*k).

Question 6)

a) Pour cet algorithme, on crée une liste A de taille i (avec i l'indice du tableau V des systèmes de capacité), donc A aura la taille du système de capacité qu'on utilisera à ce moment. On procédera d'une manière similaire que pour l'algorithme précédent, en effet si (c-V[i]<0) on prend le tableau A de l'indice précédent c'est-à-dire M[c][i-1] (la variable c est la quantité totale S pendants les tours de boucle).

Sinon on doit prendre le minimum entre M[c][i-1], M[c-V[i]][i]+1 comme pour l'algorithme précédent, pour cela on calcule la somme de tous les bocaux de M[c][i-1] et (M[c-V[i]][i])+1 et on prend le minimum. Si le minimum est (M[c-V[i]][i])+1 on incrémente A[i] de 1.

Enfin on retourne le tuple (n, A) avec n le nombre de bocaux dans A.

La complexité spatiale se voit modifiée par rapport a l'algorithme précédent suite à l'ajout dans les cases de la matrice d'un tableau A de taille i (i varie de 0 à la taille du système de capacité V). Donc dans la matrice M on aura S lignes de k tableaux de taille variable (car k colonnes) , la taille varie de 1 à k. D'où une complexité spatiale en $O(S * \sum_{i=0}^{k} i) = O(S * \frac{k*(k+1)}{2}) = O(S * k^2)$

b) **algorithme** AlgoRetour(S: entier, k: entier, V: tableau de k entiers): tuple

création d'une matrice M de taille S*k avec toutes les valeurs initialisées à l'infini

c, i: entiers

pour c de 0 à S **faire**

pour i de 0 à k **faire**

si (c = 0) **alors**

M[c][i] = 0

fin si

sinon si (i = 0) **alors**

on passe directement au prochain tour de boucle

fin sinon si

sinon alors

si ((c - V[i]) < 0) **alors**

M[c][i] = M[c][i-1]

fin si

sinon alors

M[c][i] = min{M[c][i-1], M[c-V[i]][i]+1}

fin sinon

fin alors

fin pour

fin pour

création d'un tableau A de taille k a vec toutes les valeurs initialisées à 0

nbBoc: entier

nbBoc <- M[S][k]

si (S = 0) **alors**

si (k =0)

retourner (0, [0])

```
        fin si
        sinon alors
                retourner (0, A)
        fin sinon
fin si
sinon si (k = 0) alors
        retourner (infini, [inf])
fin sinon si
si (c < 0) alors
         retourner (infini, tableau contenant un nombre k d'infinis)
fin si
tant que (nbBoc différent de 0) faire
        si ((c - V[i]) < 0) alors
                i <- i - 1
        fin si
        sinon alors
                si (M[c][i-1] < M[c-V[i]][i]) alors
                        i <- i - 1
                        nbBoc <- nbBoc - 1
                        c <- c - V[i]
                        A[i] <- A[i] + 1
                fin si
                sinon alors
                        nbBoc <- nbBoc - 1
                        c <- c - V[i]
                        A[i] <- A[i] + 1
                fin sinon
        fin sinon
fin tant que
retourner (somme(A), A)
```

La complexité temporelle de l'algorithme **AlgoRetour(S, k, V)** est la complexité de l'algorithme initial + la complexité de ce qu'il vient après le code de l'algorithme initial. En effet, après ce code on a une boucle qui s'exécute jusqu'à ce que la variable nbBocaux atteigne 0. Cependant, la complexité pire cas est bien en O(k) car dans le pire des cas on a S = 1 et on est obligé de décrémenter i jusqu'à ce que i=1 (sachant que i=k au

début de la boucle) pour pouvoir mettre à jour la liste A et donc décrémenter nbBocaux.
Donc la complexité est en $O(S * k + k) = O(S * k)$

La complexité spatiale de cet algorithme est la même que l'algorithme initial + la taille de la liste A, donc on a une complexité spatiale en $O(S * k + k) = O(S * k)$

Question 7)

Sachant que n, n², n³, n⁴ … n^n sont des complexités polynomiales et sachant que l'algorithme **AlgoRetour(S, k, V)** est en $O(S * k)$, on peut dire que l'algorithme ainsi obtenu est de complexité polynomiale. De même, si k = S on obtient $O(S^2) = O(k^2)$ qui est également une complexité polynomiale.

2.3 Algorithme III:

Question 8)

```
algorithme AlgoGlouton(S: entier, k: entier, V: tableau de k entiers): tuple
    création d'un tableau A de taille k avec toutes les valeurs initialisées à 0
    si (S = 0) alors
        si (k = 0) alors
            retourner (0, [0])
        fin si
        sinon alors
            retourner (0, A)
        fin sinon
    fin si
    sinon si (k = 0) alors
        retourner (infini, [infini])
    fin sinon si
    sinon si (S < 0)
        retourner (infini, tableau contenant un nombre k d'infinis)
    c <- S
    i <- k-1
    tant que (c différent de 0) faire
        si ((c - V[i]) < 0) alors
            i <- i -1
        fin si
        sinon alors
            A[i] <- A[i] + 1
            c <- c - V[i]
```

fin sinon

fin tant que

retourner (somme(A), A)

En analysant le code de l'algorithme **AlgoGlouton(S, k, V)** on se rend compte que la complexité temporelle dépend de S et k. En effet, la création de la liste A et la création de la liste d'infinis est en $O(k)$. De l'autre côté, la boucle while qui permet de remplir la liste A est effectuée tant que la quantité S (c dans le code) n'est pas nulle. Or au pire des cas on a V[k], V[k-1], … , V[3], V[2] > S et donc on est obligé de faire S tours de boucle puisque le seul bocal qui peut être rempli est celui de taille 1 (V[1]), donc ici la complexité temporelle est $O(S)$.

Finalement la complexité temporelle de cet algorithme est en $O(S + k) \simeq O(S)$ car généralement la quantité S est bien supérieure à l'indice k des systèmes de capacités V.

Question 9)

Un système de capacité glouton-compatible produit la solution optimale quelle que soit la quantité totale S.

Soit ici, la solution optimale est le nombre minimal de bocaux utilisés.

Prenons S = 34 et V = [1, 10, 14] avec k = 3.

L'algorithme glouton produit le couple (8, [6, 0, 2]) soit un nombre minimal de 8 bocaux.

Alors qu'une réponse optimale est (3, [0, 2, 1]) soit un nombre minimal de bocaux = 3 seulement.

Par conséquent, il existe des systèmes de capacité non glouton-compatibles.

Question 10)

On souhaite montrer que tout système de capacité V avec k=2 est glouton-compatible.

Soit V = [1, x] avec x $\in N^+$ et x > 1. On peut distinguer 2 cas selon x et S avec S > 0 (sinon pas de solution).

Soit x > S:

- On ne peut remplir aucun bocal de taille x avec la quantité S, on est obligé d'utiliser les bocaux de taille 1. Donc, A = [S, 0] qui est la solution optimale unique dans ce cas, donc le système de capacité V = [1, x] avec x > S > 0 pour k = 2 est glouton-compatible.

Soit x = S:

- Un seul bocal de taille x suffit pour stocker toute la quantité S. Donc, A = [0, 1] qui est la solution optimale unique dans ce cas, donc le système de capacité V = [0, 1] avec x = S > 0 pour k = 2 est glouton-compatible.

Soit x < S:

- On utilise autant de fois que possible le bocal de taille x, soit S/x fois (avec / la division entier). Ensuite on utilise les bocaux de 1 pour stocker ce qu'il reste, soit S%x.

  Donc, on a A = [S%x, S/x] qui est une solution optimale, car on ne peut pas utiliser moins de bocaux. En effet, on a que deux choix de bocaux (1 et x), on peut pas choisir plus de S/x fois le bocal x car

sinon x*(S/x) > S ce qui est faux. Si on utilise moins de bocaux de poids S/x, on sera obligés de compléter avec des bocaux de poids 1, cependant en faisant cela, la réponse est correcte mais non optimale car on a pas un nombre minimal de bocaux.

Finalement, le fait d'avoir k=2 nous contraint d'avoir une réponse optimale pour tout système de capacité V et donc par conséquent ce type de système de capacité est glouton-compatible.

Question 11)

L'algorithme **TestGloutonCompatible(k, V)** contient une boucle dans une boucle dans laquelle on fait appel à l'algorithme **AlgoGlouton(S, k, V)**.
Pour la première boucle on effectue (V[k−1]+V[k]−1) - (V[3]+2) = V[k-1] + V[k] - V[3] - 3 itérations, donc une complexité temporelle en $O(V[k])$ (car V[k] le plus grand en ordre de grandeur).
La deuxième boucle réalise k itérations dans laquelle on appelle la fonction AlgoGlouton(S) et AlgoGlouton(S-V[j]). On sait également que la complexité temporelle de AlgoGlouton(S) est $O(S)$, or la valeur maximale de S dans le pire des cas est $O(V[k-1]+V[k]-1-(V[3]+2)) = O(V[k])$.
Comme on appelle deux fois l'algorithme glouton, la complexité est multiplié par deux, mais il s'agit d'une constante et on garde donc la complexité temporelle en $O(V[k])$.
Par conséquent, la complexité temporelle de **TestGloutonCompatible(k, V)** est en $O(V[k] * k + V[k]) = O(V[k] * k)$ qui est un polynôme en les valeurs numériques de l'entrée.

Concernant la complexité temporelle en la taille en mémoire de l'entrée (nombre de bits), le nombre de bits nécessaires pour coder $V[k]_{bin} = log_2([V[k]]) + 1$ ce qui signifie que la valeur maximale (tous les bits à 1) pour l'entrée $V[k] = 2^{V[k]_{bin}-1}$. De même pour k, $k_{bin} = \lfloor log_2(k) \rfloor + 1$ et la valeur maximale (tous les bits à 1) que peut prendre l'entrée $k = 2^{k_{bin}-1}$.
Donc la complexité en la taille mémoire de l'entrée est la multiplication des valeurs maximales des 2 entrées V[k] et k, ce qui donne une complexité en $O(2^{V[k]_{bin}-1} * 2^{k_{bin}-1}) = O(2^{V[k]_{bin}+k_{bin}-2})$.

Finalement, l'algorithme **TestGloutonCompatible(k, V)** a une complexité temporelle polynomiale en la valeur numérique de l'entrée $O(V[k] * k)$ et une complexité différente en la taille en mémoire de l'entrée, cette dernière est exponentielle $O(2^{V[k]_{bin}+k_{bin}-2})$.
Donc on peut dire qu'il s'agit d'une complexité pseudo-polynomiale pour cet algorithme.

Mise en oeuvre

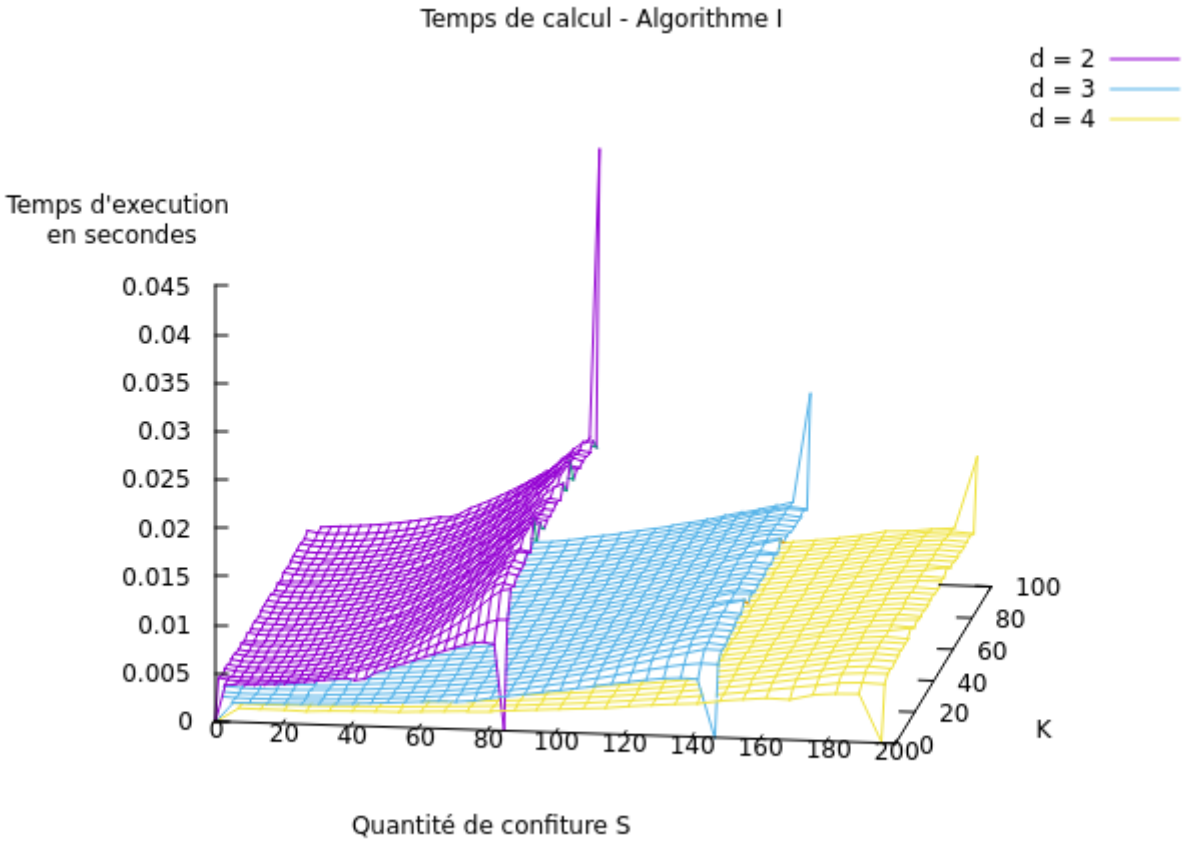
3.1 Implémentation:

Voir les fichiers “Algorithme_I.py”, “Algorithme_II.py” et “Algorithme_III.py”.
Ces fichiers possèdent les différents algorithmes présentés dans leurs sections ainsi qu’une fonction de lecture appelés **LectureAlgoRecursif(nomFichier)**, **LectureAlgoRetour(nomFichier)** et **LectureAlgoGlouton(nomfichier)** respectivement. Le paramètre “nomFichier“ de chaque fonction de lecture est le fichier texte possédant les données à lire (il s’agit du fichier “text.data”).

3.2 Analyse de complexité expérimentale:

Question 12)

Analyse des courbes de l’**Algorithme I:**

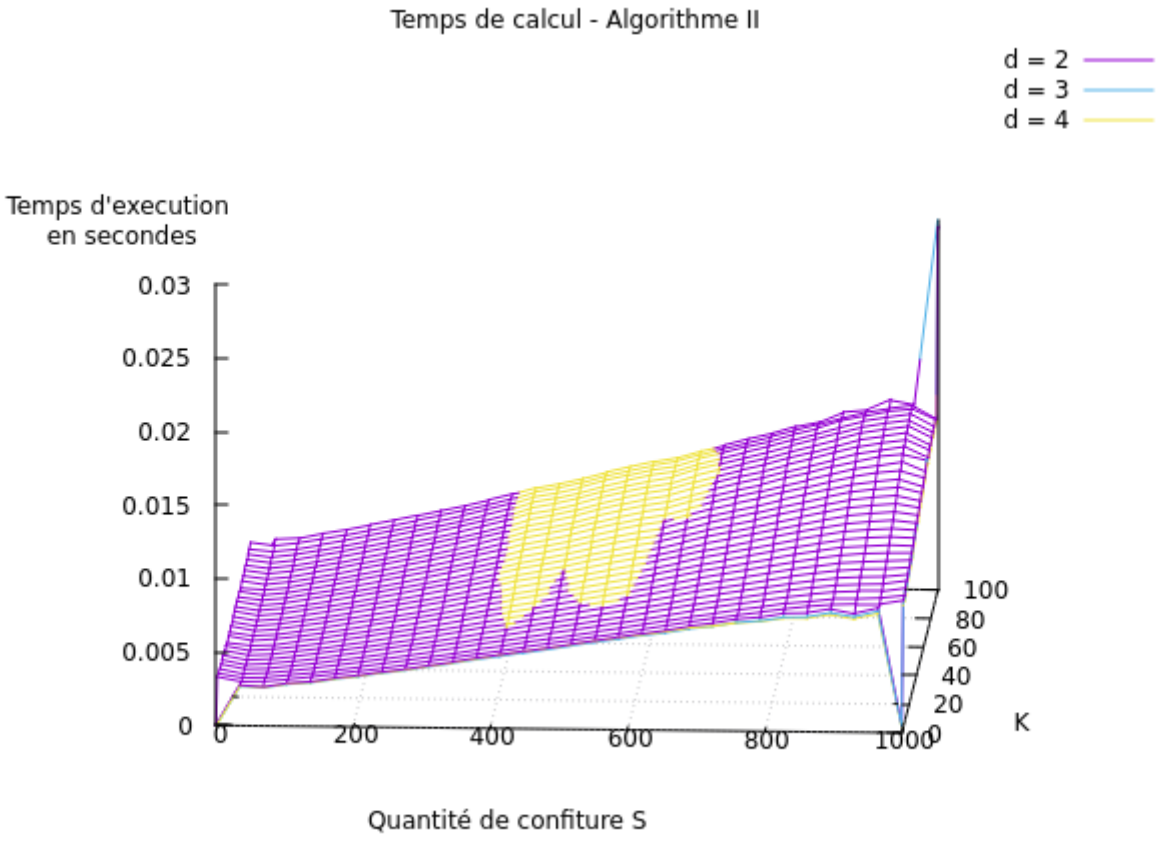


- L’algorithme I correspond a l’algorithme récursif qui recalcule à chaque fois les valeurs dont il a besoin. Donc au bout d’un certain temps il se retrouve à calculer les mêmes valeurs un nombre très élevé de fois.

  Dans ce graphe 3D, représenté en fonction de 3 composantes qui sont le temps d'exécution en secondes, la quantité de confiture S et la valeur k qui correspond au plus grand indice du système de capacité V, nous pouvons constater l'évolution exponentielle de l’algorithme dans les 3 cas (d=2, d=3 et d=4).

On constate également que plus la valeur augmente et plus le temps de calcul diminue. En effet, en augmentant d, on augmente la taille des bocaux disponibles dans le système de capacités (croissance exponentielle) et donc on pourra remplir plus rapidement des grandes quantités de S. Cet effet est également montré à travers le nombre de résultats retournés par l'algorithme pendant 1 minute (temps d'exécution du CPU). Pendant ce lapse de temps, plus la valeur de d est grande et plus on pourra retourner des résultats et donc par conséquent remplir des bocaux.

Analyse des courbes de l'**Algorithme II:**

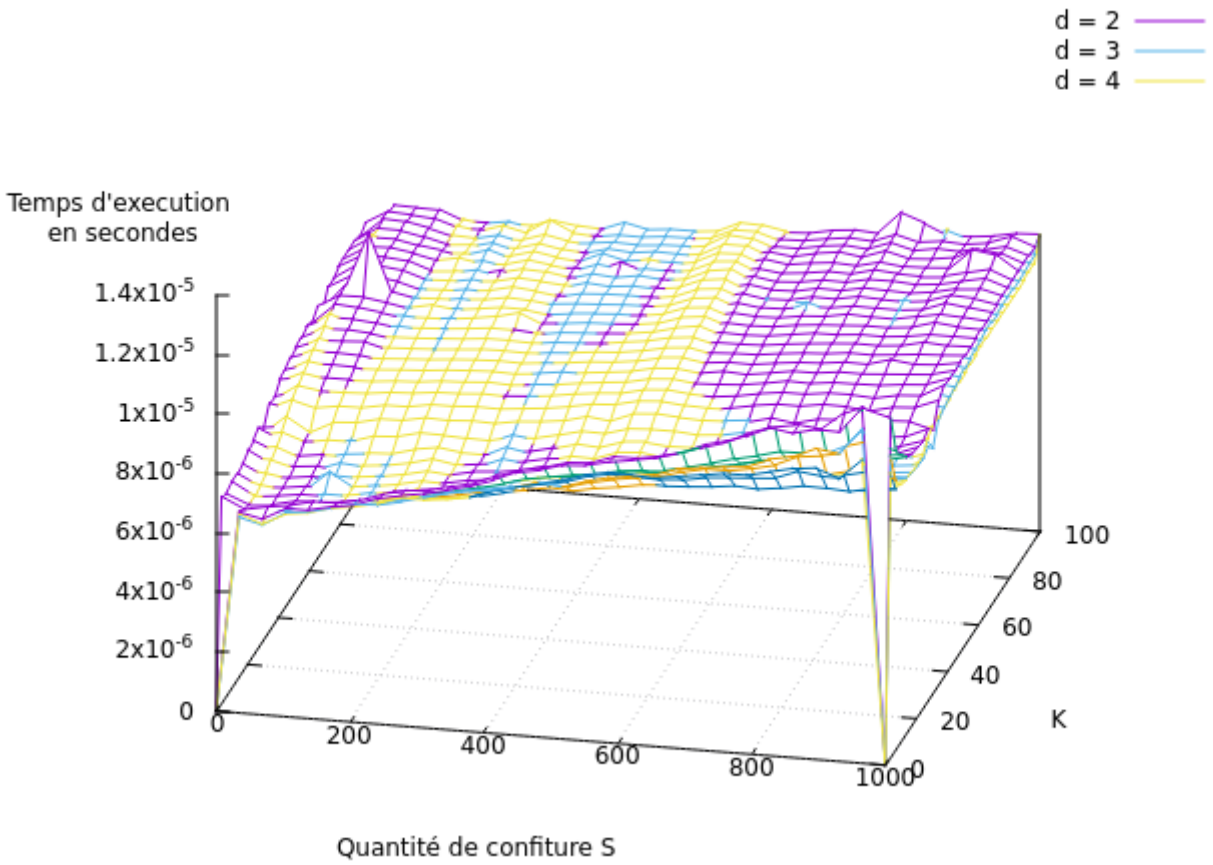


- L'algorithme II correspond a l'algorithme de retour, programmé de manière dynamique. Cette fois-ci il n'y a pas eu de contrainte de temps concernant le temps mis par le CPU a calculer les différents résultats, donc on a peut aller jusqu'aux valeurs qu'on a fixé pour S et k. Cet algorithme utilise une matrice contenant le nombre de bocaux minimaux qui lui permet de ne plus recalculer ces valeurs un nombre élevé de fois, ce qui réduit considérablement sa vitesse de calcul comme on peut également le voir sur le graphe ci-dessus.

  Ce graphe, comme le précédent est un graphe 3D, représenté en fonction de 3 composantes qui sont le temps d'exécution en secondes, la quantité de confiture S et la valeur k qui correspond au plus grand indice du système de capacité V, nous pouvons constater l'évolution similaire de l'algorithme dans les 3 cas (d=2, d=3 et d=4). Les temps sont donc sensiblement les mêmes et évoluent d'une manière linéaire, ce qui est conforme à la complexité temporelle théorique en $O(S * k)$ de cet algorithme.

SIMA Daniel DM Info/EEA 3800992
SADYKOVA Karima DM Info/EEA 28610100

Analyse des courbes de l'**Algorithme III:**



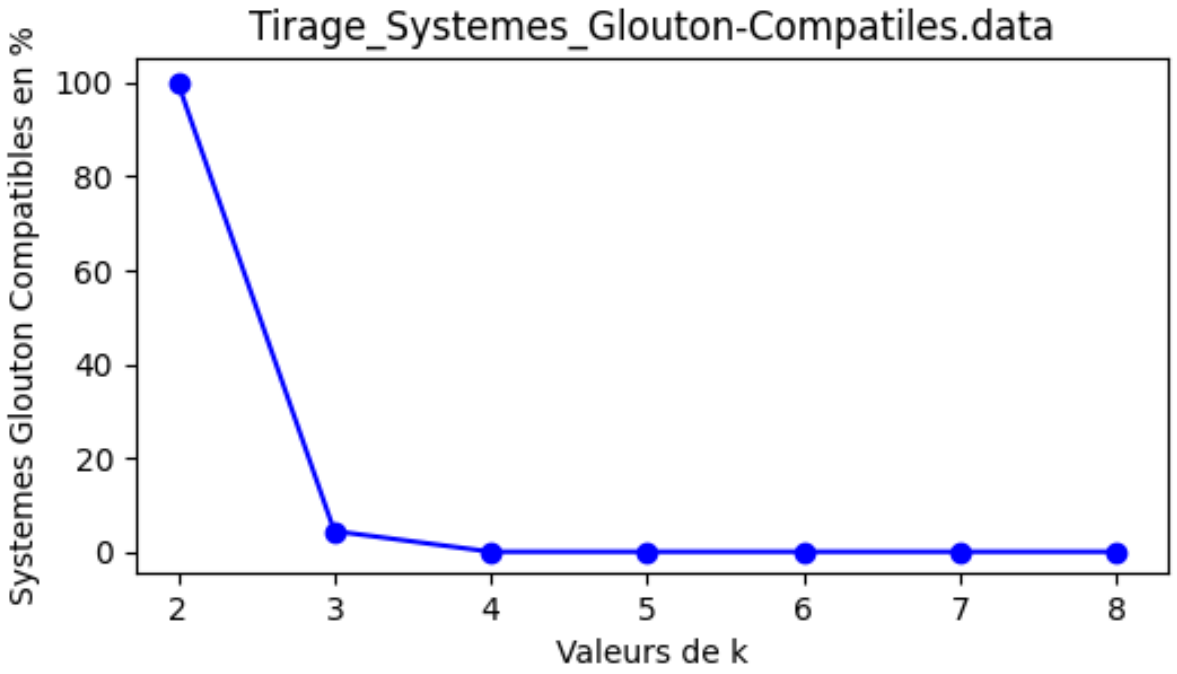
- L'algorithme III correspond à l'algorithme glouton, ce dernier prend à chaque étape une décision qui ne sera plus remise en cause. Dans notre cas, il va essayer de remplir au maximum les bocaux de plus grande taille, ce qui n'est pas toujours la réponse optimale attendue.

  Ce graphique 3D, présentant les mêmes composantes que les précédents, nous montre que l'algorithme glouton est le plus efficace en termes de rapidité de calcul. En effet, on est de l'ordre de quelques micros secondes pour retourner le couple (nombre minimum de bocaux, liste A de bocaux utilisés). Ce temps se conserve malgré l'augmentation de la valeur d, et donc de l'augmentation de la taille des bocaux. On voit donc une similarité entre les temps de calcul pour les differents d, comme pour l'algorithme de retour.

  Le moment ou S sera grand et k petit, l'algorithme glouton mettra le plus de temps à calculer le résultat, comparé aux temps mis pour d'autres valeurs de S et k (même si la différence n'est pas très grande).

  Ces temps de calculs se justifient parfaitement par la complexité théorique obtenue dans la partie précédente, en effet cet algorithme possède la plus petite complexité théorique des algorithmes de calcul de nombre de bocaux de ce projet. Il s'agit d'un complexité polynomiale en $O(S + k)$.

Question 13)

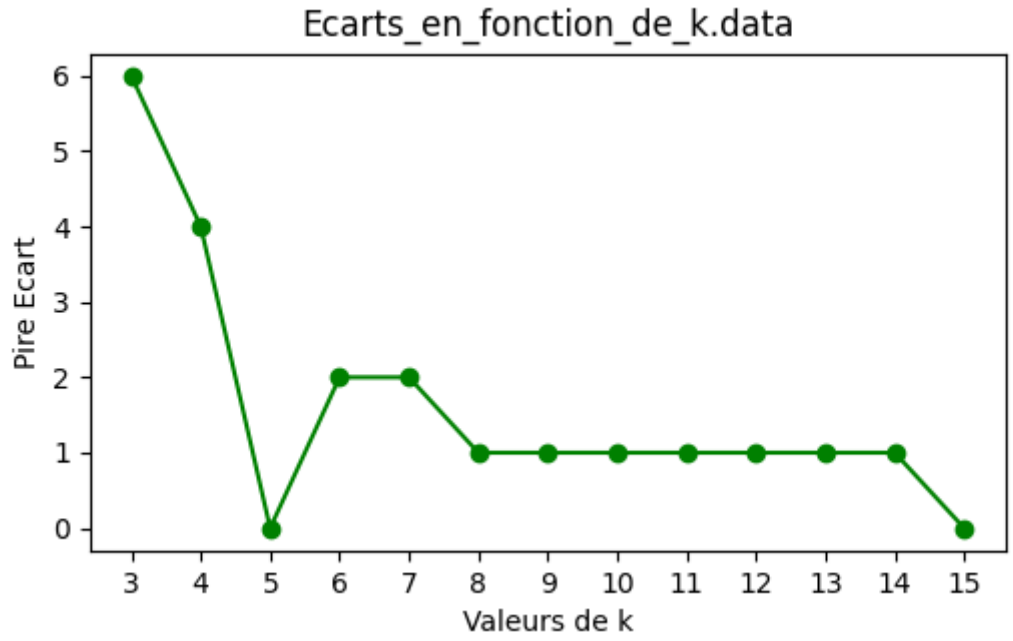


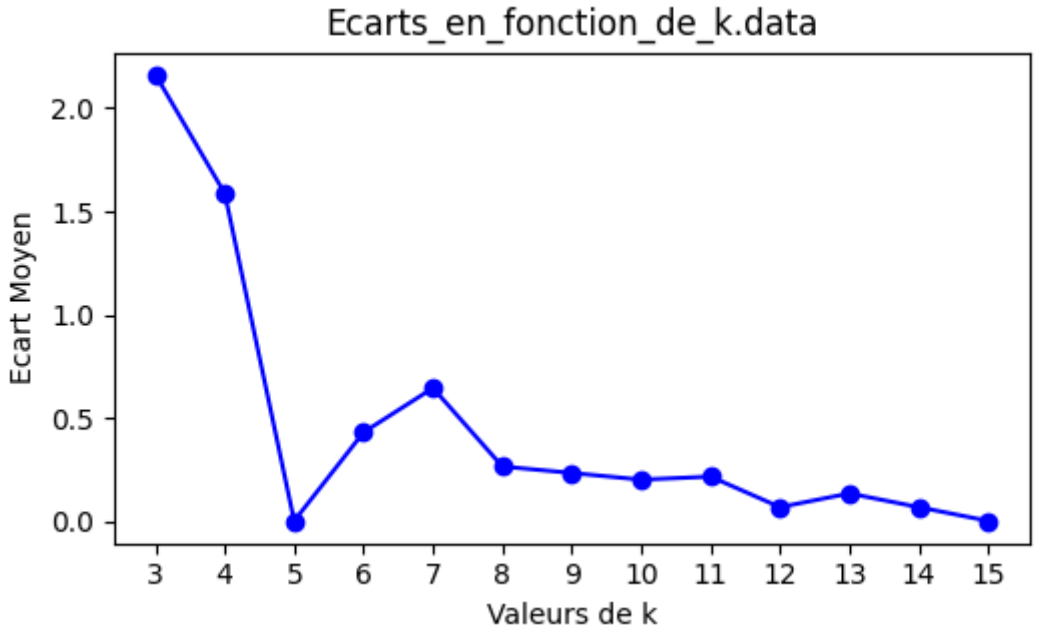
- Pour déterminer la proportion de systèmes glouton-compatibles parmi l'ensemble des systèmes nous avons fait varier la taille k et la valeur maximale pmax dans le tableau V. La courbe ci-dessus représente nos valeurs obtenues en faisant le test avec k=8 et pmax=100. Comme on le voit sur le graphe, le nombre de systèmes gloutons-compatibles vaut 0 a partir de k=4. De plus, comme on l'a montré dans les questions précédentes, un système de capacité V avec k = 2 est toujours glouton-compatible, d'où son taux de 100%.

  Les systèmes de capacité ayant un k = 3, peuvent être glouton-compatible, mais ceci à très faible proportion, le taux est légèrement inférieur à 10% (de l'ordre de 5/6%).

  Donc, on en déduit que l'algorithme Glouton n'est pas utilisable pour tous les k, car il ne trouve pas de solution optimisée à partir d'un k très petit (k = 4).

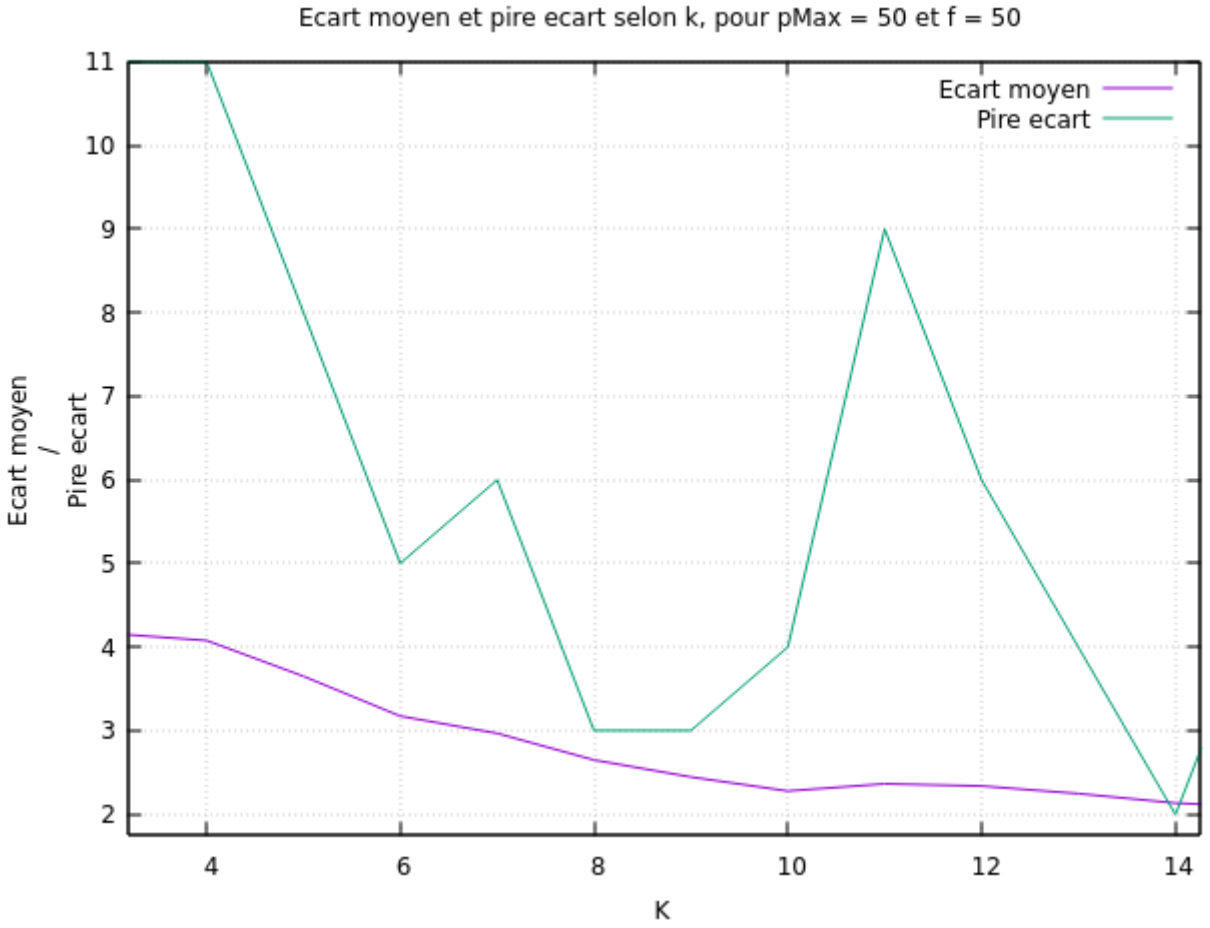
Question 14)





- En faisant varier la valeur de k et la valeur de S entre pmax et pmax*f nous avons calculé l'écart moyen et le pire écart entre deux algorithmes, AlgoRetour et AlgoGlouton pour tous les systèmes qui ne sont pas glouton-compatibles. Donc, pour chaque valeur k nous avons déterminé l'écart entre AlgoRetour et AlgoGlouton.
  Les deux courbes représentées ci-dessous sont avec la valeur k en abscisse et avec pmax=15 et f=50. La première courbe est en fonction de pire écart, la deuxième - en fonction de l'écart moyen.
  On peut remarquer que plus k est grand, plus l'écart moyen entre deux algorithmes devient petit ce qui

signifie que le nombre de bocaux minimum nécessaires pour la valeur donnée S tend à être la même pour les algorithmes gloutons-compatibles et non glouton-compatibles.



- En essayant pour des valeurs plus grandes de pmax, on remarque que l'écart moyen diminue de plus en plus ce qui renforce notre hypothèse concernant le fait que les algorithmes tendent à renvoyer le même nombre bocaux pour la même valeur S.

  Le pire écart tend à diminuer avec le k, cependant on pour différentes valeurs de S, ces écarts peuvent être très grands ou très petits..