Rapport sur le projet d'algorithmique Confitures

Yuxiang ZHANG 21202829
Sam ASLO 21210657

LU3IN003

Licence d'informatique L3
Sorbonne Université

Année universitaire 2024-2025

Partie théorique

Algorithme I

Question 1

a)

La valeur de m(S) en fonction des valeurs m(s,i) définies dans la section précédente c'est pour tout i ∈ {1, . . . , k} :

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

b)

Hypothèse:

Pour une quantité s de confiture à mettre en bocaux et un système de capacités V[1],V[2],…,V[i], le nombre minimum de bocaux nécessaires, m(s,i), satisfait la relation suivante :

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

Démonstration:

Cas de base (quand i = 1):

Cas 1 : s = 0

Lorsque s = 0, aucune confiture n'a besoin d'être mise en bocal.

D'après la définition donnée dans l'énoncé : $m(0,i) = 0$ pour tout $i \in \{1,\ldots,k\}$.

La relation de récurrence pour m(s,i) devient : $m(0,1) = \{0$ si $s = 0$ et $i = 1$

Cela est cohérent avec la définition donnée pour m(s,i) lorsque s = 0,et $\forall i \in \{1, \ldots, k\}$.

Cas 2 : $s < 0$

Lorsque $s < 0$, il est impossible de réaliser une capacité négative.

D'après la définition donnée dans l'énoncé:

$m(s,i) = +\infty$ pour tout $i \in \{1,\ldots,k\}$.

La relation de récurrence pour m(s,i) est donnée par :

$m(s,i) = \min\{m(s,i-1), m(s-V[i],i) + 1\}$.

- **Pour m(s,i− 1) :**
  Si $s < 0$, alors $m(s,i-1) = +\infty$ par définition.
- **Pour m(s−V[i],i) + 1 :**
  Si $s < 0$, alors $s-V[i] < 0$. Par définition, $m(s-V[i],i) = +\infty$. Ainsi, $m(s-V[i],i)+1 = +\infty$.

La relation de récurrence devient donc :

$m(s,i) = +\infty \Leftrightarrow \min(+\infty,+\infty) = +\infty$.

Cela est cohérent avec la définition donnée pour m(s,i) lorsque $s < 0$ et $\forall i \in \{1, \ldots, k\}$.

Cas 3 : $s \geq 1$

Le seul bocal disponible est V[1]. La relation de récurrence pour m(s,1) devient montrons que: $m(s,1) = \min\{m(s,0),m(s-V[1],1)+1\}$.

- $m(s,0)=+\infty$ pour $s>0$, car il n'y a pas de solution sans bocal.

- m(s−V[1],1) est valide si s≥V[1], et on ajoute un bocal

donc m(s,1) = min{+∞,m(s−V[1],1)+1} = m(s−V[1],1)+1

Conclusion pour le cas de base (quand i = 1):

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

Hérédité（pour tout i′>1）

Supposons que la relation soit valide pour **i = i′**. C'est-à-dire, pour tout s, nous avons :

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

montrons que la relation est vraie pour **i = i′+1**

Cas 1 : s = 0

Si s = 0, par définition : m(0,i′+1) = 0

La relation devient prouver que: m(0,i′+1) = min{m(0,i′),m(0−V[i′+1],i′+1)+1} = 0.

- Par hypothèse de récurrence,m(0,i′) = 0
- Puisque 0−V[i′+1] < 0, donc m(0−V[i′+1],i′+1) = +∞

Donc m(0,i′+1) = min{0,+∞} = 0

Cas 2 : s < 0

Si $s < 0$, par définition : $m(s,i'+1) = +\infty$

La relation devient prouver que: $m(s,i'+1) = \min\{m(s,i'),m(s-V[i'+1],i'+1)+1\} = +\infty$.

- Par hypothèse de récurrence, $m(s,i') = +\infty$ lorsque $s < 0$.
- Puisque $s < 0$, alors $s-V[i'+1] < 0$, et donc $m(s-V[i'+1],i'+1)=+\infty$.

Donc $m(s,i'+1) = \min\{+\infty,+\infty\} = +\infty$

Cas 3 : $s > 0$

Si $s > 0$, la relation devient prouver que:

$m(s,i'+1) = \min\{m(s,i'),m(s-V[i'+1],i'+1)+1\}$.

1. **$m(s,i')$:**
   Par hypothèse de récurrence, $m(s,i')$ donne le nombre minimum de bocaux pour réaliser s avec les i' premières capacités.
2. **$m(s-V[i'+1],i'+1)+1$:**
   - Si $s-V[i'+1] \geq 0$, par hypothèse de récurrence, $m(s-V[i'+1],i'+1)$ est correcte. En ajoutant un bocal de capacité $V[i'+1]$, cela représente une solution valide pour s.
   - Si $s-V[i'+1] < 0$, alors $m(s-V[i'+1],i'+1) = +\infty$, donc $m(s-V[i'+1],i'+1)+1 = +\infty$.

En prenant le minimum des deux termes ($m(s,i')$ et $m(s-V[i'+1],i'+1)+1$), nous obtenons le nombre minimal de bocaux nécessaires pour réaliser s avec les $i'+1$ premières capacités.

Conclusiton:

Par récurrence :

- La relation est valide pour $i = 1$ (cas de base).
- Nous avons supposé qu'elle est valide pour $i = i'$,et nous avons réussi à prouver qu'elle est également valide pour $i = i'+1$.

Donc la relation est démontrée pour tout $i \in \{1,\ldots,k\}$

Question 2

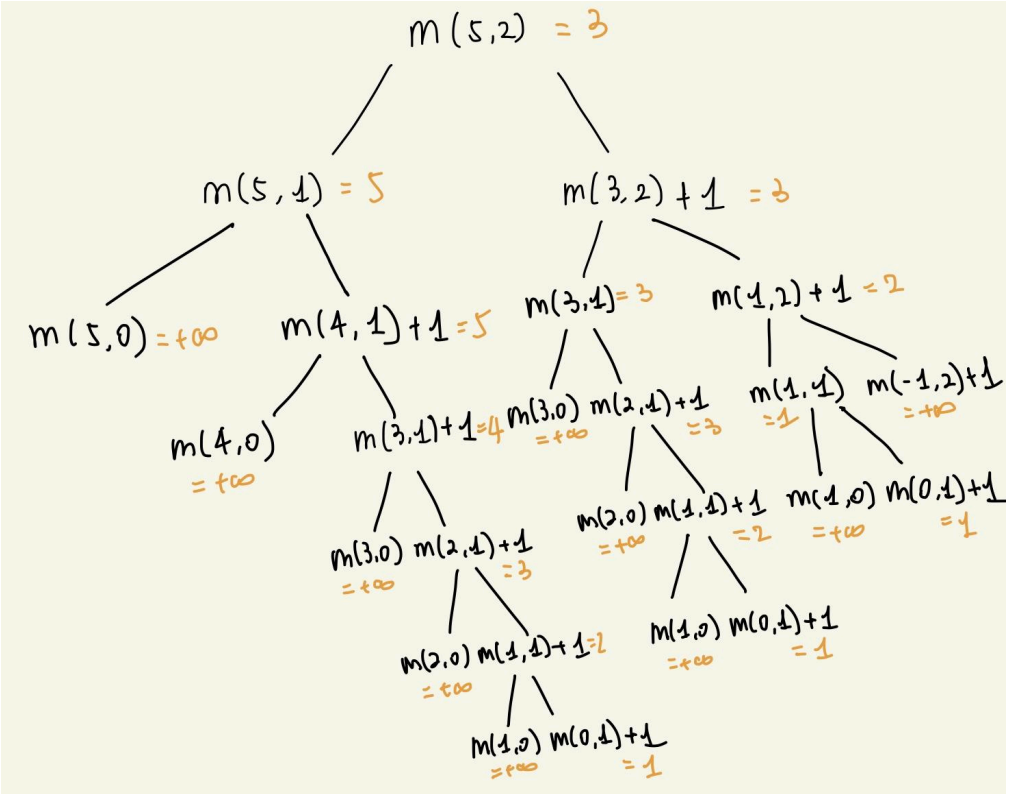
Pseudo-code pour Algorithme I:

```
Fonction m(s, i, V):
    Si s < 0:
        Retourner +∞  // impossible d'atteindre une capacité négative
    Sinon si s = 0:
        Retourner 0  // aucun bocal n'est nécessaire pour une capacité de 0
    Sinon si i = 0 et s>=1:
        Retourner +∞  // il n'est pas possible de réaliser une capacité s > 0 sans aucun bocal
    Sinon:
        // Cas général : appliquer la relation de récurrence
        // Soit m(s, i-1) le cas où on ne prend pas le i-ème bocal, et m(s - V[i], i) + 1 où on prend ce bocal.
        Retourner min(m(s, i-1), m(s - V[i], i)+1)
```

La Complexité temporelle est en O(2^i), soit exponentielle.

Question 3

Quand S = 5 et k = 2 avec V [1] = 1 et V [2] = 2, l'arbre est suivant:



Donc après appel récursif nous avons trouvé que m(5,2) = 3

Question 4

m(1, 1) elle est calculé 3 fois à la question précédente

Si k = 2 avec V [1] = 1 et V [2] = 2, pour un S impair quelconque, m(1,1) est calculée (S+1)/2 fois.

Algorithme II

Question 5

a)

Ordre de remplissage du tableau M :

Le tableau M[s,i] contient les résultats des sous-problèmes pour toutes les valeurs s (quantité de confiture) et i (types de bocaux disponibles). Pour le remplir en utilisant la formule de récurrence :

$$M[s,i] = \begin{cases} +\infty & \text{si } i = 0 \text{ et } s > 0, \\ 0 & \text{si } s = 0, \\ \min(M[s,i-1], M[s-V[i],i]+1) & \text{si } s \geq V[i], \\ M[s,i-1] & \text{sinon.} \end{cases}$$

Ordre des calculs :

1. **Initialisation :**
   - Pour s = 0, M[0,i] = 0 pour tout $i \in \{0,\ldots,k\}$.
   - Pour i = 0 et s > 0, M[s,0] = ∞.
2. **Récurrence (pour s > 0) :**

   Pour remplir M[s,i],

   - si s < V[i], alors nous avons besoin des valeurs M[s,i−1], car nous ne pouvons pas utiliser le bocal i.
   - si s ≥ V[i], alors nous avons besoin des valeurs M[s,i−1] et M[s−V[i],i],car nous pouvons utiliser au moins un bocal de type i.

Cela impose que nous progressions **de bas en haut sur s** et **de gauche à droite sur i**.

Ordre pratique :

On remplit M **en balayant les valeurs de i de 1 à k** (les types de bocaux), et pour chaque i, **on parcourt s de 0 à S** (les quantités de confiture).

b)

Pseudo-code pour AlgoOptimise(qui renvoie M):

AlgoOptimise(S, k, V)

```
    // Initialiser un tableau M de dimensions (S+1) × (k+1) avec ∞ (en O(S × k))
    M ← tableau[S+1][k+1] rempli avec ∞

    // Initialisation des cas de base en O(k)
    pour i de 0 à k faire
        M[0][i] ← 0 // Pour s = 0, m(0, i) = 0 (en O(1))
    fin pour

    // Remplir le tableau selon la relation de récurrence(programmation dynamique)
    pour i de 1 à k faire //O(k)
        pour s de 1 à S faire //O(S)
            M[s][i] ← M[s][i-1] // On ne peut pas utiliser V[i], donc on exclut ce type de bocal (en O(1))
            si s ≥ V[i] alors
                M[s][i] ← min(M[s][i-1], M[s-V[i]][i] + 1) // Cas général en O(1)
            fin si
        fin pour
    fin pour

    // Retourner le résultat final
    retourner M[S][k] //en O(1)
fin AlgoOptimise
```

c)

Analyse de la complexité temporelle :

1. **Initialisation du tableau M :**
    - Le tableau M est de taille (S+1) × (k+1), et tous ses éléments sont initialisés à ∞. L'initialisation du tableau prend donc $O((S+1)\times(k+1)) = O(S\times k)$.
2. **Initialisation des cas de base :**
    - La première boucle pour i de 0 à k initialise M[0][i] à 0, ce qui prend k+1 opérations, donc en O(k).
3. **Remplissage du tableau :**
    - La boucle externe sur i (de 1 à k) s'exécute k fois.
    - La boucle interne sur s (de 1 à S) s'exécute S fois pour chaque valeur de i.
    - À chaque itération, il y a une seule opération constante pour assigner la valeur M[s][i], soit O(1).
    - Donc la complexité est $O(k\times S)$.
4. **Retour du résultat :**
    - Le résultat final est simplement l'accès à la valeur M[S][k], ce qui est une opération constante O(1).

Complexité Temporelle Totale

En combinant toutes les étapes, la complexité temporelle totale est :

$O(S\times k) + O(k) + (k\times S) + O(1) = O(S\times k)$

Donc, la complexité temporelle totale est **$O(S\times k)$**.

Analyse de la complexité spatiale :

1. **Tableau M :**
    - Le tableau M a des dimensions (S+1) × (k+1), donc l'espace mémoire requis pour stocker ce tableau est proportionnel à la taille de ce tableau. Donc la complexité spatiale pour M est en **$O(S\times k)$**.
2. **Autres variables :**

- L'algorithme utilise des variables comme i, s, et un tableau V pour les capacités des bocaux, mais ces variables occupent une espace constant O(1).

Complexité Spatiale Totale

L'espace mémoire principal est occupé par le tableau M, donc la complexité spatiale totale est **O(S×k)**.

Résumé

- Complexité temporelle **: O(S×k)**
- Complexité spatiale **: O(S×k)**

Question 6:

a)

Pseudo-code pour AlgoOptimise2(qui renvoie A):

```
AlgoOptimise2(S, k, V)
    // Initialiser un tableau M de dimensions (S+1) × (k+1) avec ∞ (en O(s×k))
    M ← tableau[S+1][k+1] rempli avec ∞
    // Initialiser un tableau A de dimensions (S+1) × (k+1) avec des listes vides (en O(s×k))
    A ← tableau[S+1][k+1] rempli avec des listes vides

    // Initialisation des cas de base (en O(k))
    pour i de 0 à k faire
        M[0][i] ← 0 // Pour s = 0, m(0, i) = 0 (en O(1))
        A[0][i] ← [] // Pas de bocaux nécessaires pour s = 0 (en O(1))
```

```
    fin pour


    // Remplir le tableau selon la relation de récurrence (programmation dynamique)
    pour i de 1 à k faire // O(k)
        pour s de 1 à S faire // O(S)
            // Cas où on ne prend pas le bocal V[i]
            M[s][i] ← M[s][i-1]
            A[s][i] ← A[s][i-1] // On ne prend pas V[i], donc on copie la solution précédente

            // Cas où on prend le bocal V[i]
            si s ≥ V[i] alors
                si M[s][i] > M[s-V[i]][i] + 1 alors
                    M[s][i] ← M[s-V[i]][i] + 1
                    A[s][i] ← A[s-V[i]][i] // On prend la solution précédente pour s-V[i]
                    A[s][i] ← A[s][i] + [V[i]] // Ajouter V[i] à la solution
                fin si
            fin si
        fin pour
    fin pour

    // Retourner le résultat final (tableau A contenant les bocaux utilisés)
```

retourner A[S][k]

fin AlgoOptimise2

Les modifications par rapport à AlgoOptimise (Question 5b):

1. **Initialisation des tableaux M et A** :
    - Le tableau M est initialisé comme avant, mais le tableau A est initialisé avec des listes vides. A[s][i] contiendra la liste des bocaux utilisés pour atteindre la capacité s avec les bocaux disponibles parmi V[1],…,V[i].
2. **Remplissage de M et A :**
    - Pour chaque i (type de bocal) et chaque s (quantité à atteindre), on met à jour M[s][i] et A[s][i] en suivant les mêmes relations de récurrence, mais en ajoutant un élément au tableau A[s][i] chaque fois qu'un nouveau bocal est utilisé.
    - Lorsque le bocal V[i] est pris, on met à jour A[s][i] en ajoutant V[i] à la solution qui correspond à $s-V[i]$.
3. **Retourner A[S][k]** :
    - Le tableau A[S][k] contient la liste des bocaux utilisés pour atteindre la capacité S avec les k types de bocaux.

Analyse de la complexité spatiale :

1. **Tableau M** :
    - Le tableau M a des dimensions $(S+1)\times(k+1)$, et il contient un entier pour chaque case. La complexité spatiale pour M est donc $O(S\times k)$.
2. **Tableau A** :
    - Le tableau A a également des dimensions $(S+1)\times(k+1)$, mais chaque case A[s][i] contient une liste des bocaux utilisés. La taille de ces listes dépend de la quantité s et des bocaux utilisés, mais dans le pire des cas, chaque liste pourrait contenir O(S) éléments (si la solution consiste à utiliser beaucoup de petits bocaux).
    - La complexité spatiale totale pour A est donc $O(S\times k\times S)$, car chaque entrée de A pourrait contenir jusqu'à O(S) bocaux.

Complexité spatiale totale :

O(S×k)(pour M) + O(S×k×S)(pour A) = **O(S^2×k)**

b)

Pseudo-code pour AlgoRetour:

```
AlgoRetour(S, k, V)
    // Étape 1 : Initialiser et remplir le tableau M (en O(s×k))
    M ← tableau[S+1][k+1] rempli avec ∞ // (en O(s×k))

    // Initialisation des cas de base
    pour i de 0 à k faire (en O(k))
        M[0][i] ← 0 // Pour s = 0, m(0, i) = 0
    fin pour

    // Remplir le tableau selon la relation de récurrence
    pour i de 1 à k faire (en O(k))
        pour s de 1 à S faire (en O(S))
            M[s][i] ← M[s][i-1] // Cas où V[i] n'est pas utilisé
            si s ≥ V[i] alors
                M[s][i] ← min(M[s][i-1], M[s-V[i]][i] + 1) // Cas général
            fin si
        fin pour
    fin pour
```

```
    // Étape 2 : Processus de retour pour déterminer les bocaux utilisés (en O(S))
    A ← liste vide // Pour stocker les bocaux utilisés, au début initialisé à 0
    s ← S
    i ← k

    tant que s > 0 et i > 0 faire // en O(S)
        si s ≥ V[i] et M[s][i] = M[s-V[i]][i] + 1 alors
            // Le bocal V[i] a été utilisé
            A ← A + [V[i]]  // Ajouter V[i] à la liste des bocaux utilisés
            s ← s - V[i]  // Réduire la capacité restante
        sinon
            // Passer au type de bocal précédent
            i ← i - 1
        fin si
    fin tant que

    // Retourner la liste des bocaux utilisés
    retourner A // en O(1)
fin AlgoRetour
```

Explications de cet algorithme

1. **Construction de M** :

    - La première partie suit exactement la logique de programmation dynamique pour remplir le tableau M, qui contient le nombre minimum de bocaux nécessaires pour chaque capacité s et chaque type i.
2. **Processus de retour** :
    - Une fois M construit, nous utilisons un algorithme de retour (backward) pour déterminer quels bocaux ont été utilisés dans la solution optimale.
    - Nous commençons par la case M[S][k] et on revient en arrière en vérifiant si le bocal V[i] a été utilisé (grâce à la condition $M[s][i]=M[s-V[i]][i]+1$).
3. **Liste des bocaux utilisés (A)** :
    - Les bocaux utilisés sont stockés dans une liste A, qui est remplie pendant le processus de retour.

Analyse de la complexité temporelle :

- La complexité temporelle de la partie programmation dynamique (pour construire le tableau M) est déjà en $O(S \times k)$.
- **Partie de retour** :
    - Dans le pire des cas, nous revenons de M[S][k] à M[0][0]. À chaque étape, la complexité est en O(1).
    - Ainsi, la complexité du processus de retour est en O(S) (dans le pire des cas, faut nou parcourir chaque capacité de S pour reconstruire la solution optimale).
- Par conséquent, la **complexité temporelle totale** est : $O(S \times k) + O(S) =$ **$O(S \times k)$**

Analyse de la complexité spatiale :

- Le tableau de programmation dynamique M nécessite de stocker $(S+1) \times (k+1)$ valeurs, ce qui donne une complexité spatiale de $O(S \times k)$
- Le processus de retour utilise simplement une liste A pour stocker la solution optimale. Dans le pire des cas, la taille de cette liste est O(S), ce qui donne une complexité spatiale de O(S) pour la partie retour.

- Par conséquent, la **complexité spatiale totale** est : $O(S \times k) + O(S) =$ **$O(S \times k)$**

Question 7:

Oui, l'algorithme II est de **complexité polynomiale**

Démonstration :

L'algorithme II dont nous parlons est un algorithme de programmation dynamique pour résoudre un problème d'optimisation avec une étape de retour pour reconstituer la solution optimale. Nous avons deux étapes principales dans cet algorithme :

1. La partie dynamique (programming dynamique) : Cela consiste à remplir le tableau M où chaque case M[s][i] représente le minimum de bocaux nécessaires pour atteindre une capacité s en utilisant les i premiers types de bocaux. La complexité temporelle de cette étape est : **$O(S \times k)$** où S est la capacité maximale à atteindre et k est le nombre de types de bocaux disponibles.
2. Le processus de retour (backward) : Une fois que le tableau M est rempli, nous effectuons un retour pour déterminer quels bocaux ont été utilisés dans la solution optimale. Ce processus consiste à parcourir la table M de manière rétroactive et à reconstruire la solution en vérifiant si un bocal particulier a été utilisé. La complexité temporelle de cette étape est : **$O(S)$** car, dans le pire des cas, nous devons parcourir chaque capacité de S pour reconstruire la solution optimale.

Complexité totale de l'algorithme II :

- La partie dynamique a une complexité de **$O(S \times k)$**.
- La partie de retour a une complexité de **$O(S)$**.

Donc, la complexité totale de l'algorithme II est : $O(S \times k) + O(S) =$ **$O(S \times k)$**

Cela dépend de S (la capacité maximale) et de k (le nombre de types de bocaux), et comme cela est une fonction polynomiale de ces deux paramètres, l'algorithme II est bien de complexité polynomiale.

Algorithme III : Cas particulier et algorithme glouton

QUESTION 8

```
AlgoGlouton(S, k, V)
    index ← k - 1
    nb_bocaux ← 0
    tab_bocaux ← tableau de taille k initialisé avec 0

  Tant que S ≠ 0 et index ≥ 0 faire
      Si V[index] ≤ S alors
          S ← S - V[index]
          tab_bocaux[index] ← tab_bocaux[index] + 1
          nb_bocaux ← nb_bocaux + 1
      Sinon
          index ← index - 1
      Fin Si
  Fin Tant que

  Si S > 0 alors
      Retourner None
  Sinon
      Retourner (nb_bocaux, tab_bocaux)
  Fin Si
Fin Algorithme
```

```
AlgoGlouton_V2(S, k, V)
    index ← k - 1
    nb_bocaux ← 0
    tab_bocaux ← tableau de taille k initialisé avec 0

  Tant que S ≠ 0 et index ≥ 0 faire
      Si V[index] ≤ S alors
          nb_bocaux_temp ← S // V[index]
          S ← S - nb_bocaux_temp * V[index]
          tab_bocaux[index] ← tab_bocaux[index] + nb_bocaux_temp
          nb_bocaux ← nb_bocaux + nb_bocaux_temp
      Fin Si
      index ← index - 1
  Fin Tant que

  Si S > 0 alors
      Retourner None
  Sinon
      Retourner (nb_bocaux, tab_bocaux)
  Fin Si
Fin Algorithme
```

CALCULONS LA COMPLEXITÉ TEMPORELLE:

Hypothèses :

- Toutes les opérations élémentaires (affectations, comparaisons, etc.) sont effectuées en O(1).
- Le tableau V est trié de manière croissante (avec des éléments allant de 1 à k) et V[1] = 1 donc il existe toujours une solution dans V Pour S.

Analysons la complexité pour l'algorithme 1:

1. **Initialisations :**
   - `index ← k - 1` : O(1).
   - `nb_bocaux ← 0` : O(1).
   - `tab_bocaux ← tableau de taille k initialisé avec 0` : O(k), car chaque élément du tableau est initialisé à 0.

2. **Boucle principale :**
   La boucle `Tant que S ≠ 0 et index ≥ 0` s'exécute tant que S ≠ 0. (La condition index ≥ 0 est toujours vraie, d'après l'hypothèse.)

   À chaque itération, plusieurs opérations sont effectuées :

   - **Comparaison** : On vérifie si V[index] est inférieur à S : O(1).
   - Si la condition est vraie :
     - Trois affectations sont effectuées : (3×O(1)) ≈ O(1).
   - Si la condition est fausse :
     - On décrémente `index` : O(1).

   Dans le pire cas, ça veut dire si le tableau V = [1], on décrémente S de 1 à chaque itération. Par conséquent, la boucle s'exécute S fois.

3. **Après la boucle :**
   - On effectue une comparaison avec la condition `Si S > 0`, ce qui prend O(1).

4. **Conclusion :**
   - L'initialisation prend O(k), en raison de l'initialisation du tableau `tab_bocaux`.
   - La boucle principale a une complexité de O(S), car elle peut exécuter jusqu'à S itérations dans le pire des cas.

Donc, la complexité totale de l'algorithme est **O(k+S)**.

Cependant, dans le cas où S est beaucoup plus grand que k, la complexité peut être approximée à **O(S)**.

Analysons la complexité pour l'algorithme 2 (AlgoGlouton_V2):

1. **Initialisations :**
   - `index ← k − 1` : O(1).
   - `nb_bocaux ← 0` : O(1).
   - `tab_bocaux ← tableau de taille k initialisé avec 0` : O(k), car chaque élément du tableau est initialisé à 0.

2. **Boucle principale :**
   La boucle `Tant que S ≠ 0 et index ≥ 0` s'exécute tant que S ≠ 0 (La condition index ≥ 0 est toujours vraie, d'après l'hypothèse.)

   À chaque itération, plusieurs opérations sont effectuées :

   - On vérifie si V[index] est inférieur ou égal à S : O(1).
   - Si la condition est vraie (le volume du bocal courant peut être utilisé) :
     - On calcule le nombre de bocaux à utiliser avec `nb_bocaux_temp = S // V[index]` : O(1).
     - On met à jour S avec `S -= nb_bocaux_temp * V[index]` : O(1).
     - On met à jour `tab_bocaux[index]` en ajoutant `nb_bocaux_temp` : O(1).
     - On met à jour `nb_bocaux` en ajoutant `nb_bocaux_temp` : O(1).
   - Enfin, on décrémente `index` : O(1).

   Dans le pire des cas, si nous ne pouvons utiliser que le plus petit volume V[1] (c'est-à-dire si tous les autres bocaux sont trop grands pour le montant restant de S), la boucle s'exécutera au maximum k fois, car l'algorithme parcourt la table V de manière décroissante.

3. **Après la boucle :**
   - On effectue une comparaison avec la condition `Si S > 0` : O(1).

4. **Conclusion :**
   - L'initialisation prend O(k).
   - La boucle principale s'exécute en O(k).
   - La vérification finale prend O(1).

La **complexité totale** de l'algorithme est donc **O(k)**.

QUESTION 9

Il existe des systèmes de capacités qui ne sont pas glouton-compatibles. Ces systèmes peuvent présenter deux types de problèmes :

1. **Pas de solution pour le problème :**
   Un système peut ne pas être glouton-compatible si aucune solution n'existe pour résoudre le problème donné. Par exemple, supposons que nous avons une quantité totale S = 200 à remplir avec des bocaux de capacités {3, 6, 9}.
   Dans ce cas, on observe que :

   - $200 \mod 3 = 2$
   - $200 \mod 6 = 2$
   - $200 \mod 9 = 2$

   Dans chacun de ces cas, le reste est de 2, ce qui signifie qu'il reste toujours une quantité de 2 à remplir, quel que soit le choix du bocal. Par conséquent, il est impossible de remplir exactement S = 200 avec ces bocaux, ce qui montre que ce système n'est pas glouton-compatible.

2. **Choix localement optimaux mais pas de solution optimale :**
   Un système peut être faisable avec l'algorithme glouton, mais l'algorithme peut ne pas produire la solution optimale. Cela se produit lorsque les choix locaux ne mènent pas à une solution globale optimale.

   Considérons le système de capacités suivant :
   $V = \{1, 3, 4\}$ Le problème est de remplir une quantité totale S = 6 de manière optimale.

   - Solution optimale :
     La solution optimale pour remplir S = 6 est d'utiliser deux bocaux de capacité 3 :
   - Solution donnée par l'algorithme glouton :
     L'algorithme glouton choisit toujours le plus grand bocal disponible qui est inférieur ou égal à S.
     Dans ce cas, il choisit d'abord un bocal de capacité 4, car c'est le plus grand bocal disponible.
     Après avoir pris ce bocal, la quantité restante est $S - 4 = 2$.
     Ensuite, l'algorithme choisit deux bocaux de capacité 1 pour remplir les 2 restants.
     Résultat : $1 \times 4 + 2 \times 1 = 6$.
     Nombre total de bocaux : 3.

Donc dans cet exemple, l'algorithme glouton ne trouve pas la solution optimale (2 bocaux au lieu de 3), car le choix localement optimal (prendre un bocal de 4) ne mène pas à la solution globale optimale. Ce type de système n'est donc pas glouton-compatible.

QUESTION 10

Soit

- k = 2 et V = {1, v_2}, où v_2 est une capacité strictement supérieure à 1.

Pour montrer que le système est glouton-compatible, nous devons prouver que l'algorithme glouton trouve toujours une solution optimale, quelle que soit la valeur de S.

Tout d'abord, il est évident que le problème est toujours faisable, car ($S \mod 1 = 0$), ce qui signifie que, peu importe la valeur de S, il sera toujours possible de remplir la quantité S avec des bocaux de capacité 1 (et éventuellement de capacité v_2).

L'algorithme glouton fonctionne en faisant le choix entre deux capacités à chaque étape : soit prendre un bocal de capacité 1, soit prendre un bocal de capacité v_2. À chaque itération, l'algorithme choisira toujours le plus grand bocal possible, c'est-à-dire v_2, tant que cela reste possible.

Preuve par l'absurde :

Supposons que, pour une certaine quantité S, l'algorithme glouton donne une solution n et qu'il existe une autre solution optimale n′ telle que $n' < n$. Cela signifierait que, dans la solution optimale n′, on aurait utilisé plus de bocaux de capacité v_2 que dans la solution donnée par l'algorithme glouton.

Cependant, cela est impossible, car l'algorithme glouton choisit toujours v_2 lorsque cela est possible. En d'autres termes, l'algorithme ne pourrait pas choisir moins de bocaux de capacité v_2 dans la solution optimale, car tant que v_2 peut être utilisé (c'est-à-dire tant que $S \geq v_2$)), il sera toujours privilégié par l'algorithme.

Ainsi, il n'existe pas de solution optimale n′ avec $n' < n$, ce qui prouve que l'algorithme glouton trouve toujours la solution optimale si k = 2.

Conclusion :

Le système de capacités $V = \{1, v_2\}$ avec k = 2 est toujours glouton-compatible, car l'algorithme glouton trouve systématiquement la solution optimale en privilégiant le plus grand bocal possible à chaque étape.

QUESTION 11

Soit :

- V: un tableau trié de k entiers, où V[1] = 1 et V[k] est le plus grand élément du tableau.

Pour analyser la complexité de l'algorithme, on doit étudier les boucles imbriquées.

1. **Boucle externe sur S** : La boucle externe itère sur les valeurs de S allant de (V[3]+2) à (V[k−1]+V[k]−1). Le nombre d'itérations de cette boucle dépend donc des valeurs de V[3], V[k−1] et V[k]. Soit **W** la différence entre les bornes supérieures et inférieures de cette boucle : W = (V[k]+V[k−1]−1) − (V[3]+2) + 1. Donc la boucle externe s' exécute W fois.

2. **Boucle interne sur j** : La boucle interne parcourt le tableau V en faisant varier j de 1 à k. Pour chaque paire (S,j), l'algorithme appelle `AlgoGlouton(S)` deux fois : une première fois pour S et une autre fois pour S − V[j]. Donc la boucle interne s'exécute k fois.

Analyse de la complexité :

- La boucle externe s'exécute W fois.
- La boucle interne s'exécute k fois pour chaque itération de la boucle externe.
- À chaque itération de la boucle interne, l'algorithme appelle deux fois `AlgoGlouton(S)`. Puisque la complexité de `AlgoGlouton(S)` est de O(S), chaque appel est en O(S). Comme la valeur la plus grande possible pour S est W, on peut approximer la complexité de chaque appel à `AlgoGlouton(S)` à O(W).

La complexité totale de l'algorithme peut donc être exprimée comme : $O(W^2 \times k)$ Ce qui donne une **complexité pseudo-polynomiale** car elle dépend des valeurs du tableau plutôt que la taille du tableau.

Mise en Oeuvre

Nous avons implémenté nos fonctions sur la base de la partie théorique décrite dans le fichier `projet.py`. Pour tester ces fonctions, nous avons utilisé un fichier nommé `donnee.txt` comme entrée-sortie, contenant les valeurs suivantes :
- Quantité totale (S) : 151 dl
- Nombre de types de bocaux (k) : 3
- Tableau trié des capacités (V) : [1, 5, 20]

Résultats des tests :
1. Test de m(s, i, V) avec s = 151 et i = 3 : 10
2. Test de AlgoOptimise(S, k, V) : 10
3. Test de AlgoOptimise2(S, k, V) : [1, 5, 5, 20, 20, 20, 20, 20, 20, 20]
4. Test de AlgoRetour(S, k, V) : [20, 20, 20, 20, 20, 20, 20, 5, 5, 1]
5. Test de compatibilité avec la méthode gloutonne (Glouton Compatible) : True
6. Test de AlgoGlouton(S, k, V) : 10
7. Test de AlgoGloutonV2(S, k, V) : 10

Question 12

Pour tester nos algorithmes, nous utilisons le système basé sur la base d, où $d = [2, 3, 4]$. Ce système est défini de la manière suivante :

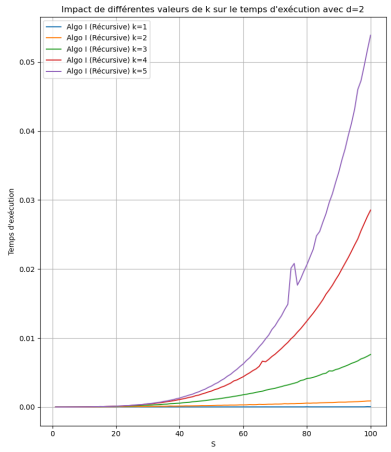
$$V[1] = 1,\ V[2] = d,\ V[3] = d^2, \ldots,\ V[k] = d^{k-1}$$

Les paramètres associés à chaque test sont ajustés en fonction de critères tels que la performance temporelle des algorithmes. Et l'objectif de ces expérimentations.

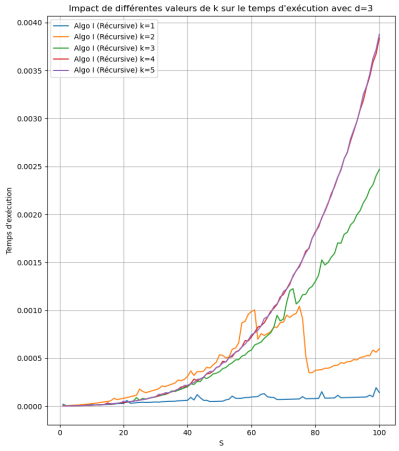
Premier test

Dans un premier temps, nous testons chaque algorithme individuellement pour observer son comportement en fonction de s et k.

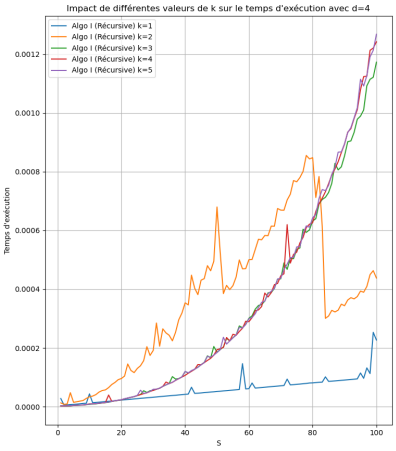
ALGO I



Impact de k et s sur l'algorithme I pour d=2



Impact de k et s sur l'algorithme I pour d=3

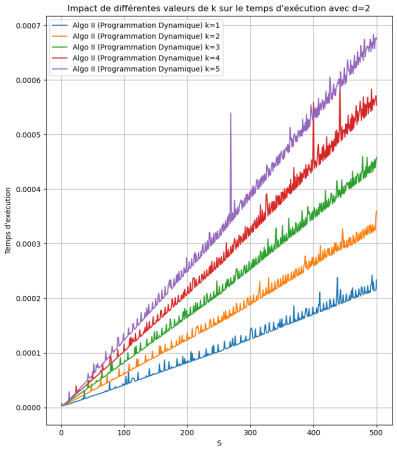


Impact de k et s sur l'algorithme I pour d=4

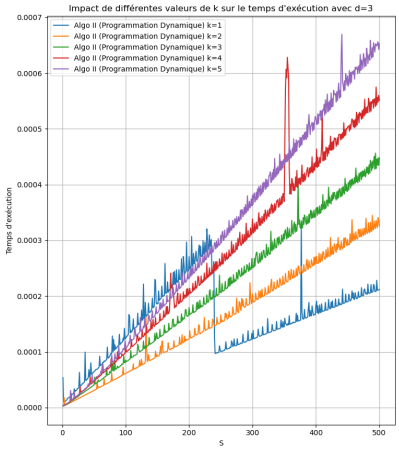
Le premier algorithme est évalué avec $k = [1, 2, 3, 4, 5]$. Les résultats nous montre que:

- Pour les trois valeurs de d, le temps d'exécution augmente avec k et s.
- Plus d est grand, plus le temps d'exécution varie fortement.

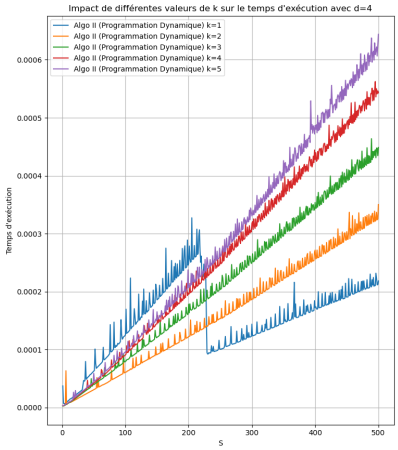
ALGO II



Impact de k et s sur l'algorithme II pour d=2



Impact de k et s sur l'algorithme II pour d=3

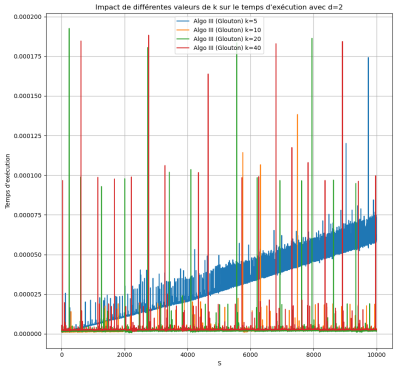


Impact de k et s sur l'algorithme II pour d=4

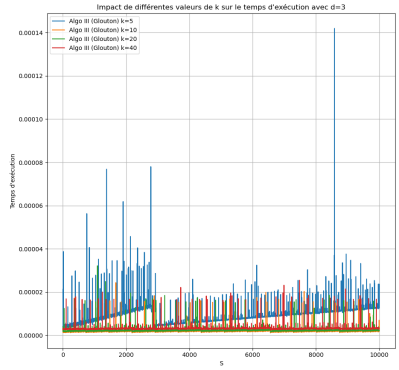
Le deuxième algorithme est évalué avec $k = [1, 2, 3, 4, 5]$. Les résultats montrent que :

- Le temps d'exécution augmente avec k et s pour les trois valeurs de d.
- Lorsque d est égal à 3 ou 4, on observe que pour les petites valeurs de s, $k = 1$ présente un temps d'exécution plus élevé. Cela peut s'expliquer par la simplicité du problème avec $k = 1$, qui mène à un algorithme moins flexible et donc moins optimal pour ces petites valeurs de s.
- Cependant, lorsque s devient grand, le temps d'exécution pour $k = 1$ devient naturellement plus petit que pour les autres valeurs de k. En effet, avec $k = 1$, l'algorithme n'a qu'une seule option à chaque étape (répéter le même bocal), ce qui simplifie le calcul. Pour des valeurs plus grandes de s , la complexité liée à la recherche de solutions alternatives est réduite, et l'algorithme devient plus rapide que lorsqu'il doit explorer plusieurs types de bocaux pour des valeurs de k plus grandes.

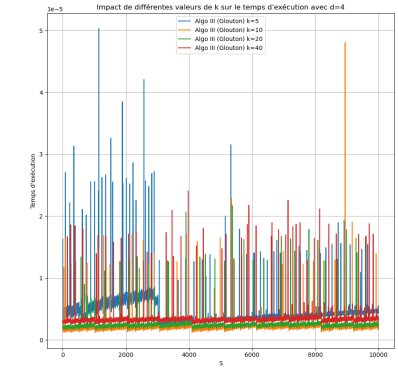
ALGO III



Impact de k et s sur l'algorithme III pour d=2



Impact de k et s sur l'algorithme III pour d=3

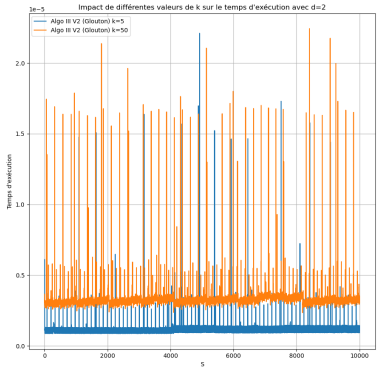


Impact de k et s sur l'algorithme III pour d=4

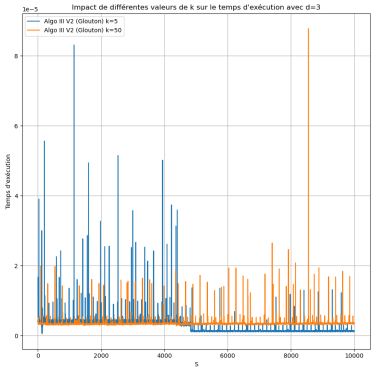
Le troisième algorithme est évalué avec $k = [5, 10, 20, 40]$. Les résultats montrent que :

- Le temps d'exécution augmente avec s, mais pas nécessairement avec k.
- Au contraire, on observe que lorsque k augmente, le temps d'exécution diminue.
- Cependant, pour la valeur $d = 4$, on remarque qu'un k très grand (40) a augmenté le temps d'exécution.

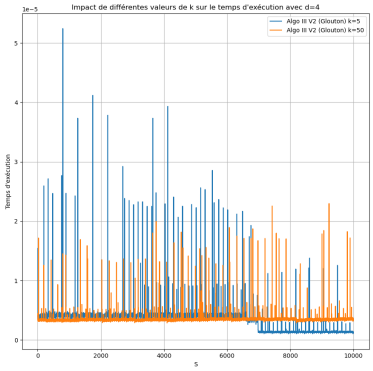
ALGO III V2



Impact de k et s sur l'algorithme III V2 pour d=2



Impact de k et s sur l'algorithme III V2 pour d=3



Impact de k et s sur l'algorithme III V2 pour d=4

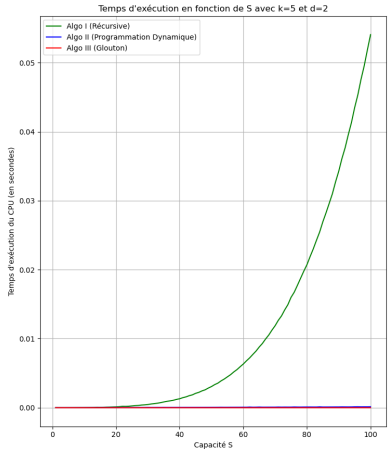
La deuxième version de l'algorithme du glouton est évaluée avec $k = [5, 50]$. Les résultats montrent que :

- Le temps d'exécution ne change pas beaucoup lorsque s varie, mais il change plutôt avec k.
- En général plus k est petit, plus le temps d'exécution est faible.

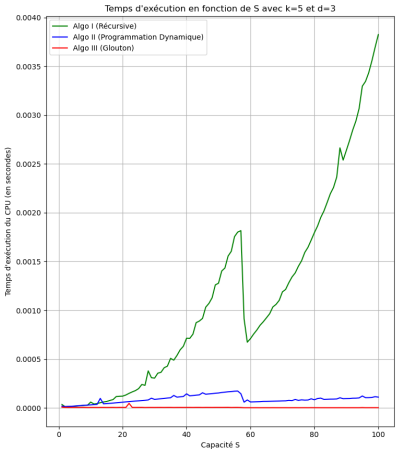
Deuxième test

Dans cette deuxième expérience, nous avons comparé les algorithmes entre eux pour déterminer celui qui a le temps d'exécution le plus rapide. Pour cela, nous avons fixé une valeur de k et avons fait varier la capacité s.

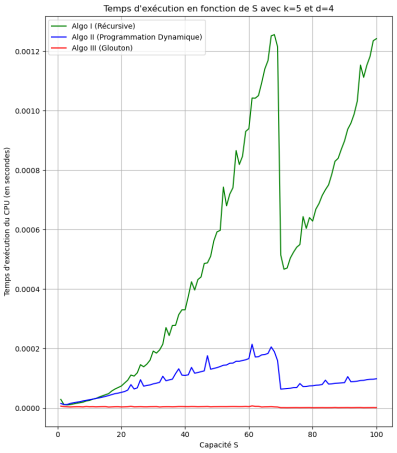
Tous les algorithmes



Temps d'exécution pour chaque algorithme pour k=5, d=2



Temps d'exécution pour chaque algorithme pour k=5, d=3

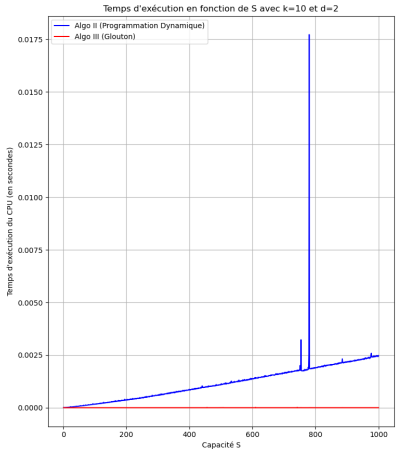


Temps d'exécution pour chaque algorithme pour k=5, d=4

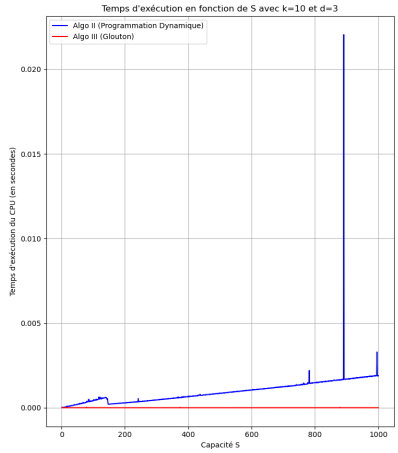
- **Premier algorithme (récursif)** : Il est évident que l'algorithme récursif présente une complexité exponentielle. En conséquence, il est beaucoup plus lent que les autres algorithmes, ce qui le rend moins performant, surtout pour des valeurs élevées de s.

Dynamique vs Glouton

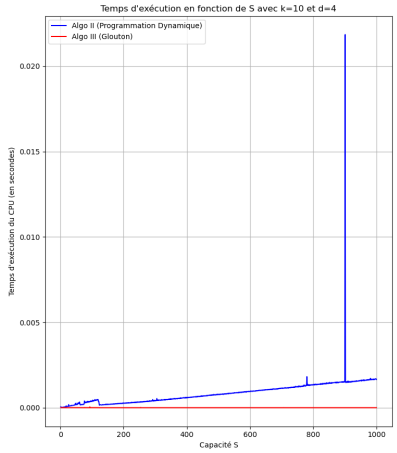
Pour tester des valeurs plus élevées de S et de k, nous avons comparé seulement l'algorithme dynamique récursif (algorithme 2) et l'algorithme glouton (algorithme 3).



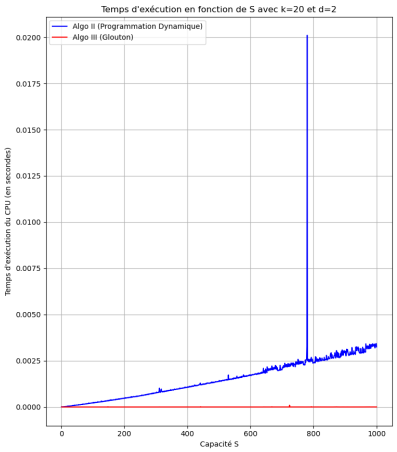
Temps d'exécution pour les algorithmes II et III pour k=10, d=2



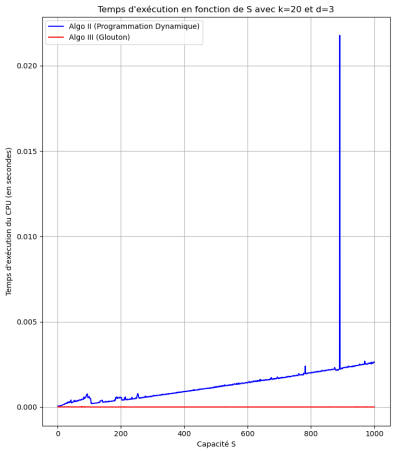
Temps d'exécution pour les algorithmes II et III pour k=10, d=3



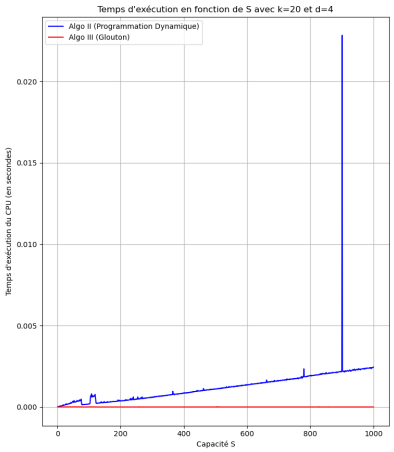
Temps d'exécution pour les algorithmes II et III pour k=10, d=4



Temps d'exécution pour les algorithmes II et III pour k=20, d=2



Temps d'exécution pour les algorithmes II et III pour k=20, d=3

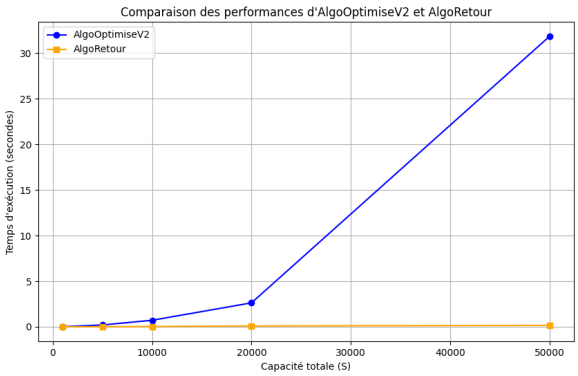


Temps d'exécution pour les algorithmes II et III pour k=20, d=4

- Nous avons effectué les tests pour des valeurs de $k = [10, 20]$. Et observé que:
  - quel que soit k, l'algorithme glouton est toujours plus rapide que l'algorithme de programmation dynamique.
  - Cette différence de performance peut être expliquée par le fait que l'algorithme glouton adopte une approche plus simple : il choisit toujours le plus grand bocal possible, ce qui évite d'explorer toutes les combinaisons possibles. En revanche, l'algorithme dynamique analyse de nombreuses solutions et calcule de manière exhaustive toutes les options, ce qui augmente considérablement le temps d'exécution, surtout pour des valeurs élevées de s.

AlgoOptimiseV2 vs AlgoRetour

Nous avons comparé les performances des deux algorithmes, **AlgoOptimiseV2** et **AlgoRetour**, en utilisant une liste de capacités $(S = [1000, 5000, 10000, 20000, 50000])$, $(k = 10)$ types de bocaux, et des capacités $(V = [1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000])$.



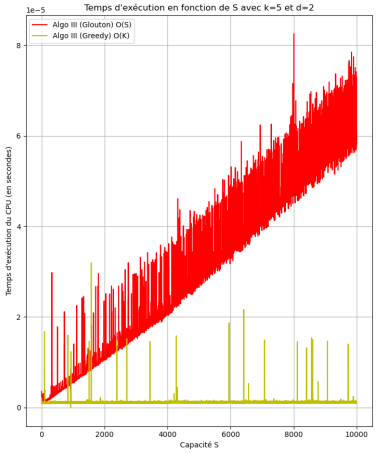
Temps d'exécution pour AlgoOptimise et AlgoRetour

- **Observation principale :**

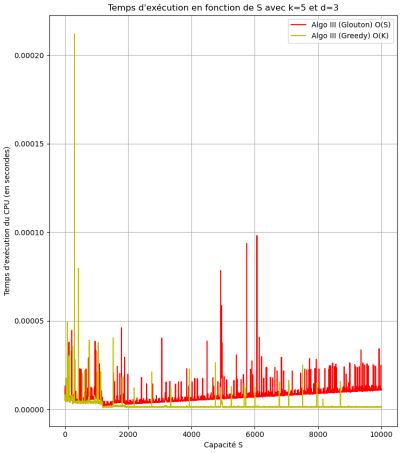
- Le temps d'exécution de **AlgoOptimiseV2** augmente fortement avec S, notamment pour des valeurs élevées comme 50000.
- **AlgoRetour**, en revanche, reste beaucoup plus rapide, même pour de grandes valeurs de S.
- **AlgoRetour** est plus efficace grâce à sa structure de retour direct, minimisant les calculs redondants.
- **AlgoOptimiseV2**, avec des structures complexes comme le tableau A, peut offrir des avantages dans certains cas, mais cela se fait au prix d'un temps d'exécution plus long.

Glouton vs Glouton V2

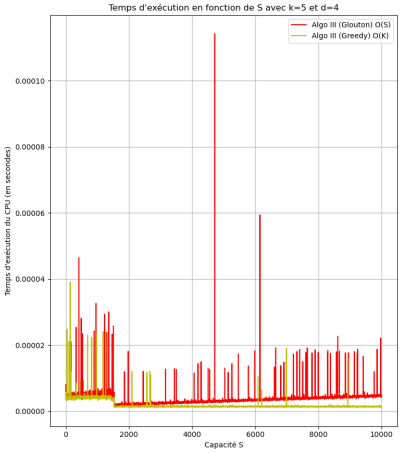
Enfin, nous avons comparé les deux versions de l'algorithme glouton.



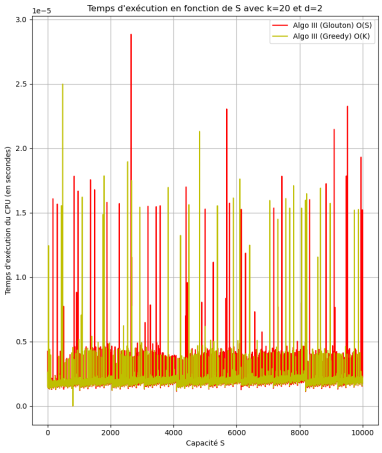
Temps d'exécution des deux algorithmes glouton pour k=5, d=2



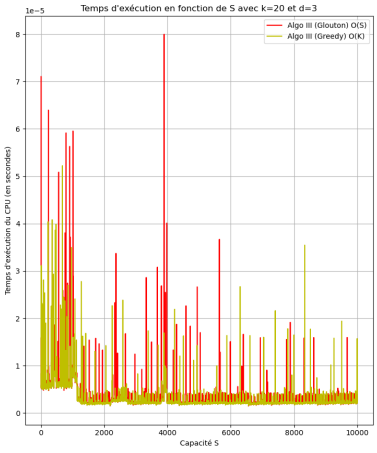
Temps d'exécution des deux algorithmes glouton pour k=5, d=3



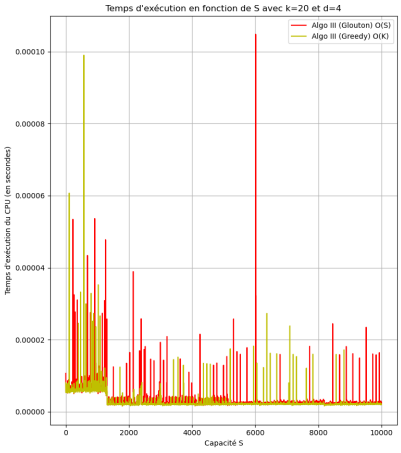
Temps d'exécution des deux algorithmes glouton pour k=5, d=4



Temps d'exécution des deux algorithmes glouton pour k=20, d=2



Temps d'exécution des deux algorithmes glouton pour k=20, d=3



Temps d'exécution des deux algorithmes glouton pour k=20, d=4

- Les résultats montrent que la deuxième version de l'algorithme est plus performante, surtout lorsque S est grand et k est petit.

En effet, dans la version 2 de l'algorithme glouton, l'optimisation consiste à utiliser autant de bocaux que possible d'un même type en une seule itération, ce qui réduit le nombre d'itérations et accélère le processus. Cette approche est particulièrement efficace lorsque la capacité à remplir, S, est grande, et le nombre de types de bocaux disponibles, k, est relativement petit.

Conclusion

En fin, les résultats de nos tests confirment que nos calculs théoriques de la complexité temporelle sont corrects. On peut résumer les performances des différents algorithmes comme suit :

- **Premier algorithme (récursif)** : L'algorithme récursif présente une complexité exponentielle, ce qui le rend extrêmement lent, notamment pour des valeurs élevées de S. En effet, l'algorithme explore toutes les combinaisons possibles, ce qui donne une complexité de $O(2^i)$. Cela signifie que le temps d'exécution double à chaque ajout d'un type de bocal, ce qui entraîne des performances très faibles à mesure que k et S augmentent.
- **Deuxième algorithme (Dynamique avec mémorisation)** : Cet algorithme utilise la programmation dynamique et la mémorisation, ce qui améliore considérablement ses performances par rapport à l'algorithme récursif. Sa complexité temporelle est $O(k \cdot S)$.
  En ce qui concerne **AlgoRetour** et **AlgoOptimiseV2**, on observe que :
  Si l'objectif est de minimiser le temps d'exécution, **AlgoRetour** est le choix privilégié, notamment pour de grands S et k.
  AlgoOptimiseV2, en revanche, demeure pertinent lorsqu'il est nécessaire d'obtenir des informations détaillées sur la composition des solutions.
- **Troisième algorithme (Glouton)** : L'algorithme glouton est le plus rapide, avec une complexité de $O(S)$. Il fonctionne en choisissant toujours le plus grand bocal possible à chaque étape, ce qui permet de remplir la capacité de manière optimale sans avoir besoin d'explorer toutes les combinaisons possibles. Cependant, il peut ne pas toujours fournir la solution optimale pour tous les cas, car il ne prend pas en compte d'autres possibilités plus petites. De plus, la version 2 de l'algorithme glouton, qui présente une complexité de $O(k)$, peut être encore plus rapide pour certaines valeurs de S et k.

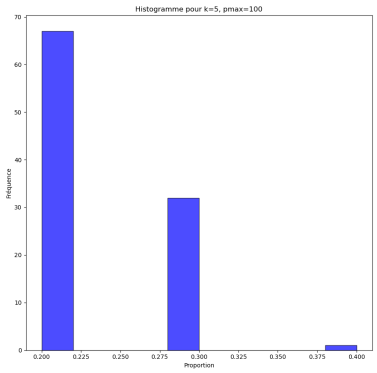
Question 13

Pour évaluer la proportion de cas où l'algorithme glouton fonctionne correctement (il trouve une solution optimale) pour des bocaux générés aléatoirement, nous avons procédé comme suit :

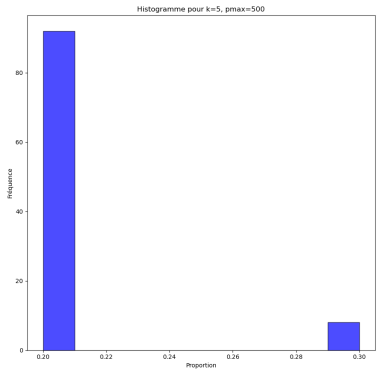
1. **Génération aléatoire d'instances** : des ensembles de bocaux ont été générés avec des capacités comprises entre k et $p_{\max}$.
2. **Évaluation de l'algorithme** : pour chaque ensemble de bocaux, nous avons vérifié si le système est glouton-compatible.
3. **Répétition des tests** : chaque configuration de $p_{\max}$ a été testée 100 fois, et la proportion de compatibilité a été calculée.

Paramètres des tests :

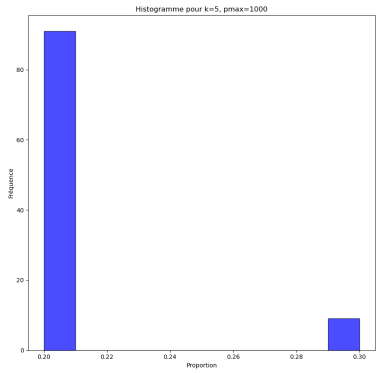
- **Nombre des bocaux (k)** : $k = 5$ et $k = 30$.
- **Capacités des bocaux ($p_{\max}$)** : trois plages différentes ont été testées :
  - $p_{\max} = 100$: capacités faibles,
  - $p_{\max} = 500$: capacités moyennes,
  - $p_{\max} = 1000$: capacités élevées.



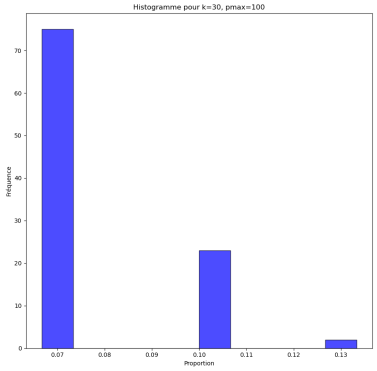
Proportion de compatibilité de l'algorithme glouton pour k=5, pmax=100



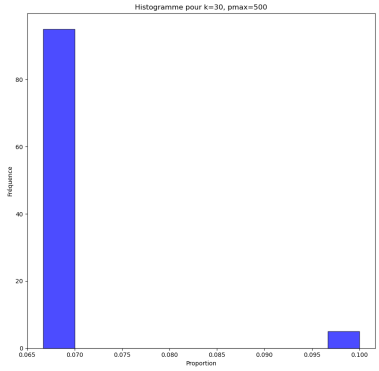
Proportion de compatibilité de l'algorithme glouton pour k=5, pmax=500



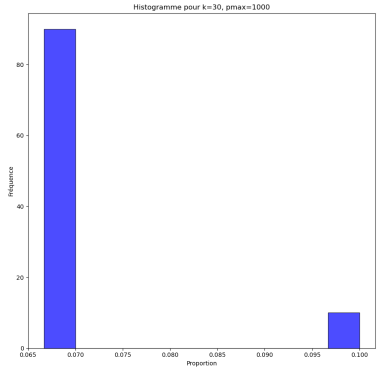
Proportion de compatibilité de l'algorithme glouton pour k=5, pmax=1000



Proportion de compatibilité de l'algorithme glouton pour k=30, pmax=100



Proportion de compatibilité de l'algorithme glouton pour k=30, pmax=500



Proportion de compatibilité de l'algorithme glouton pour k=30, pmax=1000

Observations pour $k = 5$:

- $p_{\max} = 100$:
  - La proportion est plus élevée qu'avec $k = 30$, atteignant souvent **20%**, avec des valeurs maximales de **40%**.
- $p_{\max} = 500$:
  - La proportion de succès est similaire à celle observée pour $p_{\max} = 100$, autour de **20%**, avec des valeurs maximales de **30%**.
- $p_{\max} = 1000$:

- La proportion reste inchangée avec les précédents $p_{\max}$, oscillant autor **20%**, avec des valeurs maximales de **30%**.

Observations pour $k = 30$:

- $p_{\max} = 100$:
  - La proportion de succès est faible, entre **6%** et **7%**, avec quelques cas isolés atteignant **10%**.
- $p_{\max} = 500$:
  - Les proportions sont similaires, restant majoritairement entre **6%** et **7%**, atteignant occasionnellement **10%**.
- $p_{\max} = 1000$:
  - La tendance reste inchangée, avec une proportion de succès entre **6%** et **7%**, et un cas maximum atteignant **13.33%**.

Résumé des résultats :

- Pour $k = 30$, les proportions restent généralement faibles, indépendamment de $p_{\max}$, indiquant une compatibilité limitée de l'algorithme glouton pour des cas complexes.
- Pour $k = 5$, l'algorithme glouton obtient de meilleurs résultats, avec des proportions de succès significativement plus élevées, jusqu'à **40%**. Cela montre que pour des cas simples (k faible), l'algorithme est plus souvent compatible avec une solution optimale.

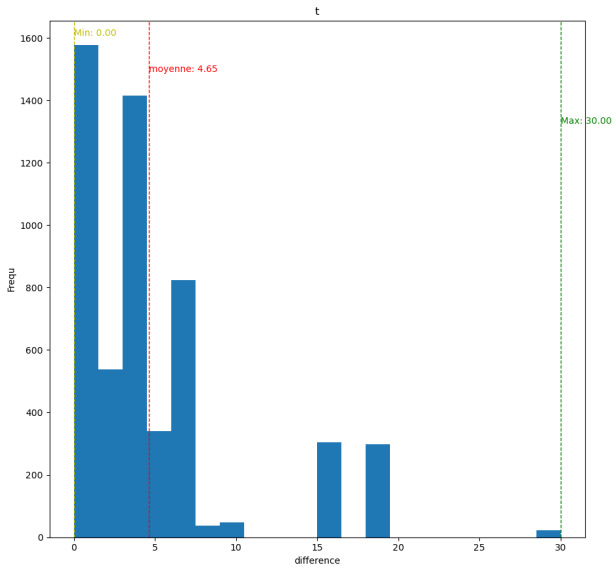
Question 14

Pour évaluer l'écart des résultats obtenus avec l'algorithme glouton par rapport à une solution optimisée, voici ce qui a été fait :

1. **Génération aléatoire d'instances** : des ensembles de bocaux ont été générés avec des capacités comprises entre k et $p_{\max}$.
2. **Évaluation de l'algorithme glouton** : pour chaque ensemble de bocaux, on a vérifié si si le système est glouton-compatible ou non. Si non, on calcule l'écart entre le nombre donné par l'algorithme glouton et par l'algorithme optimisé.
3. **Tests répétés** : pour chaque configuration de $p_{\max}$, les tests ont été répétés 2000 fois et les résultats d'écart ont été calculés.

Paramètres des tests :

- **Nombre de bocaux: (k)** = 5
- **Capacités des bocaux: ($p_{\max}$)** =200
- **Facteur multiplicatif (f)** :10



Proportion d'écart entre l'algorithme glouton et la solution optimale pour k=5, pmax=200

Les résultats montrent que :

- Pour les valeurs de k petites, même si l'algorithme glouton ne trouve pas la solution optimale, il sera dans beaucoup de cas très proche de la solution optimale.
- On voit aussi que, dans beaucoup de cas, l'algorithme glouton trouve la solution optimale, mais les systèmes sont jugés non compatibles. Pour mieux comprendre cela, nous analyserons cet exemple: Quand i = 3, V = [1, 3, 4], pmax = 10, f = 3, nous observons le résultat suivant :

```
S=11, AlgoGlouton=3, AlgoOptimise=3, ecart=0
S=12, AlgoGlouton=3, AlgoOptimise=3, ecart=0
S=13, AlgoGlouton=4, AlgoOptimise=4, ecart=0
S=14, AlgoGlouton=5, AlgoOptimise=4, ecart=1
S=15, AlgoGlouton=4, AlgoOptimise=4, ecart=0
S=16, AlgoGlouton=4, AlgoOptimise=4, ecart=0
S=17, AlgoGlouton=5, AlgoOptimise=5, ecart=0
S=18, AlgoGlouton=6, AlgoOptimise=5, ecart=1
S=19, AlgoGlouton=5, AlgoOptimise=5, ecart=0
S=20, AlgoGlouton=5, AlgoOptimise=5, ecart=0
S=21, AlgoGlouton=6, AlgoOptimise=6, ecart=0
S=22, AlgoGlouton=7, AlgoOptimise=6, ecart=1
S=23, AlgoGlouton=6, AlgoOptimise=6, ecart=0
S=24, AlgoGlouton=6, AlgoOptimise=6, ecart=0
S=25, AlgoGlouton=7, AlgoOptimise=7, ecart=0
S=26, AlgoGlouton=8, AlgoOptimise=7, ecart=1
S=27, AlgoGlouton=7, AlgoOptimise=7, ecart=0
S=28, AlgoGlouton=7, AlgoOptimise=7, ecart=0
```

```
S=29, AlgoGlouton=8, AlgoOptimise=8, ecart=0
S=30, AlgoGlouton=9, AlgoOptimise=8, ecart=1
```

- En observant les données réelles, nous constatons que:
  pour certaines valeurs de (S), l'**écart** est effectivement égal à zéro. Cela indique que, bien que la fonction `TestGloutonCompatible` juge l'instance comme non compatible, les résultats de `AlgoGlouton` et `AlgoOptimise` restent identiques pour certaines valeurs de (S).
  L'algorithme glouton peut "par chance" produire le même résultat que l'algorithme optimisé. Cela peut s'expliquer par la combinaison spécifique des pièces, où, même si l'algorithme glouton n'est pas optimal en théorie, il donne tout de même une solution correcte.
  Par exemple :
  Pour $(S = 11, 12, 13, 15, 16, 17)$, on observe que (écart = 0).
  Ces valeurs pourraient correspondre à une certaine décomposition localement optimale, permettant à l'algorithme glouton de "coïncider par hasard" avec l'algorithme optimisé.
  On peut donc conclure que même si le système est jugée non compatible, l'algorithme glouton reste correct pour de nombreuses valeurs de (S).

Conclusion du projet :

Dans le cadre du projet d'algorithmique *Confitures*, nous avons exploré trois approches principales pour résoudre le problème de minimisation du nombre de bocaux nécessaires pour atteindre une capacité donnée (S). Ces algorithmes se distinguent par leurs complexités temporelles :

1. **AlgoI (récursif)** : Bien qu'il offre une solution exacte, sa complexité exponentielle ($O(2^k)$) le rend impraticable pour de grandes valeurs de S ou k.
2. **AlgoII (programmation dynamique)** : Avec une complexité en $O(S \times k)$, il garantit une solution optimale. Il est plus rapide que l'algorithme précédent. Cependant, il est plus lent que l'approche glouton.
3. **AlgoIII (glouton)** : Très rapide ($O(k)$ ou $O(S)$), il est efficace dans les cas compatibles, mais son succès dépend fortement de la compatibilité glouton, qui n'est pas toujours assurée.

En implémentant ces algorithmes en Python, nous avons confirmé expérimentalement leurs complexités théoriques. Les résultats montrent que, lorsque l'algorithme glouton est compatible, il constitue le meilleur choix grâce à sa rapidité. En revanche, pour les cas non compatibles, bien que ses résultats soient souvent proches de l'optimum, il est préférable d'utiliser AlgoII pour garantir une solution optimale.

Observations clés :

Des tests sur des cas non compatibles ont révélé que l'algorithme glouton peut parfois produire des solutions exactes. Ces cas dépendent de la configuration des capacités (V) et des valeurs (S). Cela montre une certaine robustesse de l'approche glouton, même dans des situations où sa compatibilité n'est pas garantie.

Condition additionnelle :

Nous avons également observé une règle qui garantit la compatibilité glouton :

- Si chaque $V[i]$ (pour $i \geq 2$) est une puissance de $V[2]$, alors le système est toujours compatible avec l'algorithme glouton.
  Cette structure hiérarchique des capacités élimine les conflits possibles et permet à l'algorithme glouton de fournir systématiquement une solution optimale.

Conclusion générale :

Le choix de l'algorithme dépend donc du contexte :

- Pour des solutions rapides sur des instances simples ou compatibles, l'algorithme glouton est imbattable.
- Pour une fiabilité totale, notamment dans les cas non compatibles, AlgoII est indispensable.

Perspectives :

Il serait intéressant d'explorer des algorithmes hybrides combinant la rapidité du glouton avec des garanties partielles d'optimalité. Par ailleurs, améliorer la détection de compatibilité glouton pourrait réduire les échecs potentiels. Enfin, une analyse plus approfondie des configurations où V est basé sur des puissances permettrait de développer des solutions encore plus robustes et efficaces.