Mattias Culevski & Antoine Lecomte

Algorithme I :

Question 1 : a)
Pour initialiser la récursion, on pose :
-m(0, i) = 0 $\forall i \in \{1, ... , k\}$: il est possible de réaliser la capacité totale 0 avec 0 bocal. Il n'y a pas de confiture à placer dans les bocaux.
-m(s, 0) = + ∞ $\forall s \geq 1$: il n'est pas possible de stocker de la confiture (si s ≥ 1) sans utiliser au moins un bocal.
-m(s, i) = + ∞ $\forall i \in \{1, ... , k\}$ $\forall s < 0$: il n'est pas possible d'avoir une quantité négative de confiture.

Avec k le nombre de types de bocaux disponibles (car tableau V de taille k), on peut exprimer la valeur m(S) en sous-problèmes m(s,i) de la façon suivante :
m(S) = m(S,k).
Cela donne la relation de récurrence suivante : m(s,i) = min(m(s,i−1), 1+m(s−V[i],i)).

Explications : -Option 1 (m(s,i−1)) : Ne pas utiliser de bocaux de capacité V[i], on se limite aux i−1 premiers types (dans le cas où s<V[i] si on utilisait un bocal i, sa capacité serait trop grande et ce bocal ne serait pas rempli au complet, cela nous renvoie au cas où s<0).
-Option 2 (1+m(s−V[i],i)) : On utilise un bocal de capacité V[i], ce qui laisse une quantité s−V[i] à remplir avec i types de bocaux disponibles. Cela donne 1+m(s−V[i],i), où 1 représente le bocal utilisé.
La solution optimale sera donc le minimum de bocaux utilisé entre les 2 options : m(s,i) = min{m(s,i−1), m(s−V[i],i)+1}.

Exemple avec V=[1,6,10] et S=13 :
Calculons m(13,3) (en utilisant jusqu'à V[3]=10).
m(13,3) = min(m(13,2), 1+m(3,3))
Option 1 : m(13,2) (n'utiliser que V[1] et V[2]).
Option 2 : 1+m(3,3) (utiliser un bocal de 10, puis résoudre pour S=3).

Calculons chaque terme :
m(13,2) avec V=[1,6] :
Pour S=13, on peut utiliser deux bocaux de 6 (12 dg) et un bocal de 1.
m(13,2)=3.
1+m(3,3) : On utilise un bocal de 10, ce qui laisse S=3 à remplir.
Pour m(3,3), on ne peut qu'utiliser V[1]=1, donc m(3,3)=3.
Donc 1+m(3,3)=1+3=4 et donc m(13,3)=min(3,4)=3. La solution optimale est d'utiliser deux bocaux de 6 et un bocal de 1 (d'où le fait d'utiliser le min).

b)
Preuve de la récurrence :

Initialisation : Pour i=1 :
-si s=0 : m(s,i) = m(0,1) = 0 d'après la définition du problème.

-si s<0 : m(s,i) = m(s,1) = +∞ d'après la définition du problème, impossible de remplir la quantité avec des bocaux.
On trouve m(s,0) = +∞ et m(s-V[1],1)+1 = +∞ (et min(+∞,+∞) = +∞).

-si s>0 : on dispose uniquement des bocaux de taille V[1]=1 car i=1, cela nécessite s bocaux pour contenir s décigrammes de confiture.
m(s,i) = m(s,1) = s. Cela respecte la relation de récurrence car m(s,i-1) = m(s,0) = +∞.
On choisit donc m(s-V[1],1)+1 = m(s-1,1)+1 = s (et min(+∞,s) = s).
Pour i=1, la relation de récurrence est donc vérifiée.

Hérédité : Supposons que la relation est vraie pour i-1, donc que m(s,i-1)=0 si s=0, que m(s,i-1)=+∞ si s<0 et que m(s,i-1) = min{m(s,i-2), m(s-V[i-1],i-1)+1} sinon.
Montrons alors que la relation reste vraie pour i ∈ {2, ... , k}.

-si s=0 : m(s,i) = m(0,i) = 0 d'après la définition du problème.

-si s<0 : m(s,i) = +∞ d'après la définition du problème, impossible de remplir la quantité avec des bocaux.
On trouve m(s,i-1) = +∞ par hypothèse de récurrence et m(s-V[i],i) = +∞ (car s-V[i]<0), et min(+∞,+∞) = +∞.

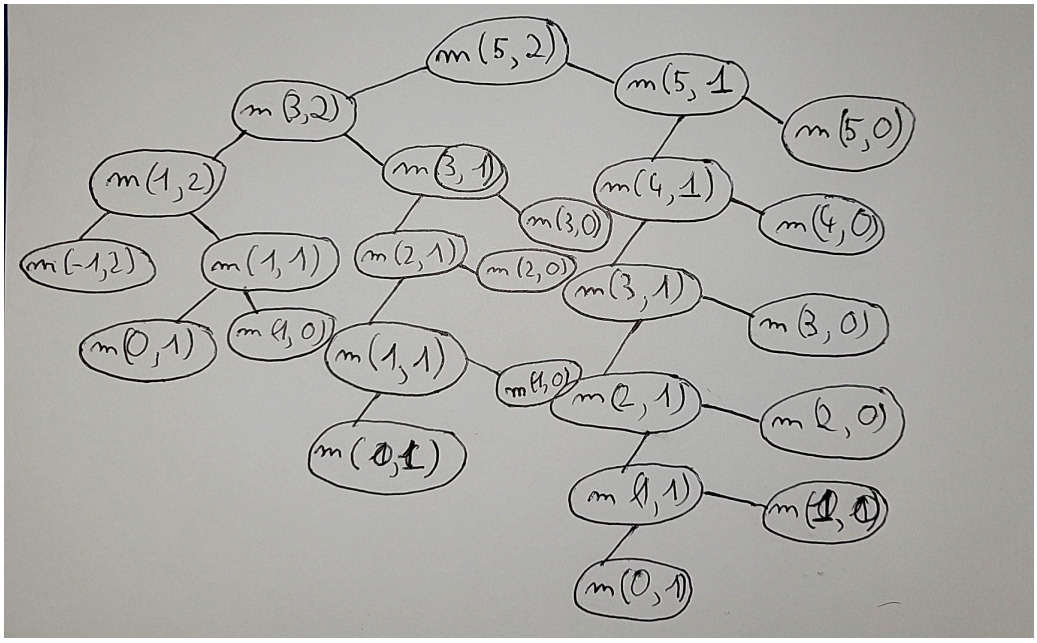
-si s>0 : on doit montrer que m(s,i) = min{m(s,i-1), m(s-V[i],i)+1}.
-Sans utiliser de bocal V[i] : on se limite aux i-1 premiers types de bocaux, on calcule m(s,i-1).
-En utilisant un bocal V[i] : il reste une quantité à remplir de s-V[i]. Tous les bocaux sont disponibles, on calcule donc m(s-V[i],i)+1 (où 1 représente le bocal utilisé).
La solution optimale sera donc le minimum de bocaux utilisé entre les 2 options : m(s,i) = min{m(s,i−1), m(s−V[i],i)+1}.
Par hypothèse de récurrence, m(s,i-1) et m(s-V[i],i) sont bien définis, donc la relation est également vérifiée pour i ∈ {2, ... , k}.

Conclusion : La relation est vraie pour tout i ∈ {1, ... , k} et pour toutes les valeurs de s.

Question 2:
Entrées :
S : quantité de confiture (entier)
k : nombre de types de bocaux (entier)
V : tableau des capacités des bocaux, indexé de 1 à k
Sortie :
Nombre minimum de bocaux nécessaires pour S décigrammes, 0 si S=0, ou +∞ si k=0 ou S<0

```
Si S=0 alors
        Retourner 0
Si k=0 ou S<0 alors
        Retourner +∞
Sinon
        option1 ← m(s,k-1,V)
        option2 ← m(s-V[k],k,V)+1
        Retourner min(option1,option2)
```

Question 3:



Question 4 :
Dans notre cas, avec s=5 et k=2, on calcule 3 fois la valeur m(1,1).

On remarque que chaque fois que m(s−2,2) est calculé pour un s, cela implique une évaluation de m(1,1). m(1,1) est donc calculé dans m(5,2), m(3,2), et m(1,2).
De façon générale, si s est impair, les appels récursifs d'un nombre s seront séparés en deux parties : tout d'abord, l'algorithme va dérouler dans une branche les appels m(s,1) jusqu'à ce que s soit égal à -1 (car s est impair), cas d'arrêt. Ici, m(1,1) est calculé une fois. (Il s'agit de l'appel m(1,1) lié à l'appel initial m(S,2) qui est devenu m(S,1).)
Ensuite, une deuxième branche pour le cas m(s-V[2],2) : de la manière dont l'algorithme est écrit, cet appel refait le calcul m(s',1) avec s' = s-V[2] = s-2, jusqu'à ce que s' = -1, et cela pour chaque appel à cette deuxième branche. Cela correspond aux appels m(S-2,2), m(S-4,2),…, m(-1,2) qui sont devenus respectivement m(S-2,1), m(S-4,1),…, m(-1,1) suite à un appel m(s,i-1).
On remarque que pour s=1, on a 1 calcul de m(1,1). (car m(1,2) donne lieu à un appel)
Pour s=3, on a 2 calculs de m(1,1). (car m(3,2) et m(1,2) donnent lieu à un appel)
Pour s=7, on a 4 calculs de m(1,1). (car m(7,2), m(5,2), m(3,2) et m(1,2) donnent lieu à un appel)
Donc, de façon générale, on aura n = (s+1)/2 avec n le nombre de calculs de m(1,1).

Algorithme II :

Question 5: a)
Les cases du tableau M seront remplies dans l'ordre des appels récursifs de l'algorithme:
- Cas s=0:
On remplit donc toutes les cases de M[0,i] par 0, car il n'y a pas de confiture (i allant de 0 à k).
- Cas i=0:
On remplit ensuite toutes les cases de M[s,0] par +∞, car il est impossible de remplir s>0 sans bocaux (s allant de 1 à S).
- Cas général (i>0) :
Pour chaque valeur de i et s, i∈{1,…,k} et s∈{1,…,S} :
Si s<V[i], (le bocal i est trop grand) M[s,i] = M[s,i-1], car on ne peut pas utiliser ce bocal.
Sinon, on peut au moins utiliser un bocal de type i, donc M[s,i] = min(M[s,i-1], M[s-V[i],i]+1), d'après la formule de récurrence.

La relation de récurrence s'écrit donc :

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

Donc, on peut remplir les cases M[0,i] avec “0” pour i=0 à k.
Puis, remplir les cases M[s,0] avec “+∞” pour s=1 à S.
Et enfin, pour chaque i∈{1,…,k}, on remplit les cases M[s,i] pour s∈{1,…,S} en considérant les deux derniers cas de la récurrence.

b)
Algorithme “AlgoOptimisé” :

Entrées :
S : quantité de confiture (entier)
k : nombre de types de bocaux (entier)
V : tableau des capacités des bocaux, indexé de 1 à k

Sortie :
Nombre minimum de bocaux nécessaires pour S décigrammes, ou +∞ si impossible

```
Si S<0 :
    Retourner +∞ et []  // Retourner M[S][k] va créer une erreur sinon
Initialiser un tableau M de dimensions (S+1) x (k+1)
Pour i de 0 à k :
    M[0][i] ← 0    // Cas s = 0 : 0 confiture nécessite 0 bocal
Pour s de 1 à S :
    M[s][0] ← +∞    // Cas i = 0 : impossible sans bocaux
Pour i de 1 à k :   // Parcourir tous les types de bocaux
    Pour s de 0 à S : // Parcourir toutes les quantités
        Si s < V[i] :
            M[s][i] ← M[s][i-1]  // Impossible d'utiliser le bocal de type i
        Sinon :
            M[s][i] ← min(M[s][i-1], M[s-V[i]][i] + 1)  // Relation de récurrence
Retourner M[S][k] et M  // Résultat final : nombre minimum de bocaux nécessaires (et besoin du tableau dans son intégralité pour le premier paramètre de l’algorithme retour “backward”)
```

c)
Complexité temporelle :

On parcourt tous les types de bocaux i ∈ {1,…,k} et pour chaque valeur de i (chaque type de bocal), on parcourt toutes les quantités s∈{0,…,S} dans une boucle interne. Donc la boucle externe sur i est exécutée k fois et la boucle interne sur s est effectuée S+1 fois.
Le nombre total d’itérations est k×(S+1). Donc, le nombre total d’itérations est en O(k×S) car on ignore la constante additive 1.
De plus, chaque itération dans les boucles est en O(1) :

-vérification de la condition $s<V[i]$.
-calcul du minimum $\min(M[s][i-1],M[s-V[i]][i]+1)$.
-mise à jour d'une case dans le tableau par une affectation simple dans $M[s][i]$.
Donc, chaque itération étant en temps constant en $O(1)$, la complexité temporelle totale est $O(k \times S)$.

Complexité spatiale :

M est de dimensions $(S+1)\times(k+1)$ car nous avons $s \in \{0,\dots,S\}$ et $i \in \{0,\dots,k\}$. L'espace occupé par le tableau M est proportionnel au nombre de cases : il est donc de $O(k \times S)$.
On peut noter également que les variables (i,s,V[i],etc…) que l'algorithme utilise pour les calculs, prennent chacune en mémoire un espace en $O(1)$, que l'on néglige dans le calcul de complexité spatiale car M occupe un espace nettement supérieur.
Donc, la complexité spatiale totale est $O(k \times S)$.

Question 6 : a)
Les différentes cases de $M[s][i]$ ne contiendront plus uniquement un entier représentant le nombre minimal de bocaux nécessaires mais un tableau A de taille k où $A[j]$ représente le nombre de bocaux utilisés pour le type $V[j]$, avec $j \in \{1,\dots,k\}$ (comme illustré dans l'introduction du projet).

Complexité spatiale :

Chaque case $M[s][i]$ contient désormais un couple avec :
-un entier pour le nombre minimum de bocaux (déjà présent dans l'algorithme précédent) qui occupe $O(1)$ en espace.
-un tableau A de taille k (ajouté) qui occupe $O(k)$ en espace.

Les variables (s,i,V[i],etc…) occupent $O(1)$ espace, leur place occupée est négligeable en notation de Landau.
Sachant qu'il y a $(S+1)\times(k+1)$ cases dans M, cela donne donc la complexité totale de $O((S+1)\times(k+1)\times k) = O(S \times k^2)$, ce qui est cohérent car le facteur multiplicatif k par rapport à la complexité de l'algorithme considéré précédemment s'explique par les k nouvelles places occupées par chaque sous-tableau A pour chaque case $M[s][i]$ du tableau M.

b)
L'algorithme "retour" doit reconstituer le tableau A des bocaux utilisés, en utilisant les résultats stockés dans le tableau M. Il suffit de regarder quel est l'ordre des choix effectués pour obtenir la solution optimale, qui minimise le nombre de bocaux, donc :
-si $M[s,i] = M[s,i-1]$, cela signifie qu'on n'utilise pas le bocal de type i pour la solution optimale (ou bien on arrête de l'utiliser, puisque rien n'empêche d'utiliser plusieurs bocaux de même

capacité).
-si M[s,i] = M[s-V[i],i]+1, en revanche, cela signifie qu'on utilise le bocal de type i, ce qui revient à considérer maintenant la quantité restante de confiture (s-V[i]) car on remplit un bocal de capacité V[i] que l'on compte dans notre solution optimale.

Algorithme "retour" (Backward) :

Entrées :
M : tableau des résultats du premier algorithme (S+1) x (k+1)
S : quantité de confiture (entier)
k : nombre de types de bocaux (entier)
V : tableau des capacités des bocaux, indexé de 1 à k

Sortie :
Tableau A indiquant le nombre de bocaux pris pour chaque type

```
Initialiser un tableau A de taille k avec des valeurs à 0 (A[1..k] ← [0, ..., 0])
Initialiser une variable s ← S
Pour i de k à 1 :
   // Reconstituer la solution en partant de la dernière case du tableau M
   Tant que s > 0 et i > 0 :
      Si M[s, i] == M[s, i-1] :
         // Aucun bocal de type i n'a été utilisé
         Passer à i-1
      Sinon :
         // Un bocal de type i a été utilisé
         A[i] ← A[i] + 1  // Ajouter un bocal de type i
         s ← s - V[i]     // Décrémenter la quantité de confiture
Si s=0 :
   Retourner A

Sinon :
   Retourner "Impossible"   // nous avons toujours V[1]=1, donc, dans notre cas, l'algorithme
n'atteint jamais cette partie du code sauf si k=0 et S>0
```

Complexités de l'algorithme "retour" :

Complexité temporelle :

On commence par initialiser le tableau A (de taille k), ce qui fait O(k).
On boucle sur i allant de k jusqu'à 1, tous les types de bocaux sont ainsi parcourus et pour

chaque type de bocal on fait une comparaison (en O(1)) et une décrémentation de s (en O(1)).
Donc, l'algorithme "retour" a une complexité temporelle totale de O(k).

Complexité spatiale :

Le tableau M a une taille de (S+1)×(k+1), sa complexité spatiale est donc O(S×k). (Les deux constantes additives 1 ne changent pas le résultat dans la notation de Landau.)
Le tableau A, de taille k, est lui de complexité spatiale O(k).
Donc, l'algorithme "retour" a une complexité spatiale totale de O(S×k)+O(k) = O((S+1)×k) = O(S×k), car la constante additive est négligeable en notation de Landau.

Complexité temporelle totale (obtenue en enchaînant l'algorithme initial et l'algorithme "retour") :

L'algorithme initial a une complexité temporelle de O(S×k).
L'algorithme "retour" a quant-à-lui une complexité temporelle de O(k).
On calcule alors la somme :
O(S×k)+O(k)=O((S+1)×k)=O(S×k), qui est la complexité temporelle totale, O(k) étant négligeable en comparaison avec O(S×k) lorsque S est grand.

Complexité spatiale totale (obtenue en enchaînant l'algorithme initial et l'algorithme "retour") :

On évite donc de stocker inutilement à chaque fois le tableau A dans les cases M[s,i].
Donc, l'algorithme initial est en O(S×k).
L'espace requis pour A est géré par l'algorithme "retour", en O(S×k).
On calcule alors la somme des espaces nécessaires pour stocker les tableaux M et A :
O(S×k)+O(S×k)=2×O(S×k)=O(S×k), qui est la complexité spatiale totale, car le facteur multiplicatif 2 n'affecte pas la croissance asymptotique lorsque S et k deviennent très grands.

En conclusion, on remarque qu'avec cette nouvelle approche, on a traité deux sous-problèmes séparément, cela a permis d'améliorer la complexité totale, nous avions trouvé pour rappel une complexité spatiale totale de $O(S \times k^2)$ dans la première approche (avec le tableau A présent dans toutes les cases du tableau M) qui est maintenant réduite à O(S×k) avec l'algorithme "retour" séparé de notre "AlgoOptimisé".

Question 7 :
S et k sont nos paramètres d'entrée (quantité de confiture et nombre de types de bocaux).
O(S×k) est un produit linéaire en S et k, car on peut l'écrire par exemple comme O(n) où n=S×k, qui est bien polynomial. Si l'algorithme était de complexité exponentielle, on aurait une complexité comparable à O(2^n) ou O(n!), ce qui est beaucoup plus coûteux lorsque les paramètres d'entrée deviennent très grands.

Algorithme III : Cas particulier et algorithme glouton

Question 8 :
Algorithme “AlgoGlouton” :

Entrées :
S : quantité totale de confiture (entier positif)
k : nombre de types de bocaux
V : tableau des capacités des bocaux (V[1],]V[2], ..., V[k]) dans l'ordre croissant (V[1]<V[2]<...<V[k])

Sortie :

Tableau A indiquant le nombre de bocaux utilisés pour chaque capacité

```
A←[0,0,...,0]  //tableau de taille k
s←S

Pour chaque type de bocal i allant de k à 1 :
   Tant que s ≥ V[i] :
      A[i]←A[i]+1  //ajouter un bocal de ce type
      s←s−V[i]

Afficher A  // voir les bocaux utilisés

Si s = 0 :
   Retourner la somme des valeurs de A  //nombre de bocaux utilisés

Sinon :
   Retourner +∞          // nous avons toujours V[1]=1, donc, dans notre cas, l’algorithme n’atteint jamais cette partie du code sauf si k=0 et S>0
```

Complexité temporelle :

Dans la boucle principale de l’algorithme, pour chaque bocal i, on fait autant d’itérations que le nombre de bocaux de ce type que l’on utilise. Dans le pire des cas, aucun bocal hormis V[1] ne serait suffisamment petit pour stocker une partie de la confiture car ils seraient tous de capacité strictement supérieure à S, la quantité totale de confiture, il faudrait alors parcourir décigramme par décigramme la confiture jusqu’à S, ce qui donne une complexité maximale de O(S).
La complexité temporelle de l’algorithme est donc de O(S).

Question 9 :
Comme précisé dans l'énoncé, un système de capacité est dit glouton-compatible si, pour ce système de capacité, l'algorithme glouton produit la solution optimale quelle que soit la quantité totale S, et l'algorithme glouton choisit toujours le bocal de plus grande capacité possible, sans chercher à anticiper les éventuels problèmes que cela peut engendrer.

Pour vérifier si un système de capacité possède cette propriété, on simplement vérifier si la condition de divisibilité est respectée, qui est la suivante :
Chaque capacité V[i] est un multiple de toutes les capacités suivantes (V[i+1],V[i+2],…).
Cette propriété est très utile car cela garantit que de choisir le bocal de plus grande capacité restante à chaque étape, n'empêchera pas de pouvoir remplir exactement la quantité totale S de confiture de manière optimale.
Remarque : il est possible qu'un système de capacité n'ait pas cette propriété mais soit quand même glouton-compatible, mais tout système qui satisfait cette propriété le sera à coup sûr.

Exemple pour illustrer qu'il existe des systèmes de capacité qui ne sont pas glouton-compatibles (l'exemple fourni dans la question 1-a fonctionne bien) :

V=[1,6,10] et S=13.
Le plus grand bocal possible est V[3]=10.
On utilise 1 bocal de 10, ce qui laisse S=13−10=3.
Avec S=3, on ne peut utiliser que V[1]=1.
On utilise 3 bocaux de 1, ce qui complète S=3−1−1−1=0.
La solution gloutonne donne 1×10+3×1=4 bocaux au total.

L'algorithme optimal (programmation dynamique) va rechercher le nombre minimum de bocaux qu'il est possible d'atteindre (s'il trouve 4 comme l'algorithme glouton, alors le système de capacité est glouton-compatible, sinon il ne l'est pas).
m(13,3)=min(m(13,2), 1+m(3,3)).
m(13,2) calcule le nombre minimum de bocaux en utilisant V=[1,6].
1+m(3,3) signifie qu'on utilise V[3]=10 puis que l'on poursuit avec S=3.
Calcul de chaque terme :
m(13,2) : En utilisant V=[1,6], on peut choisir deux bocaux de 6 et un bocal de 1 pour avoir 13. Donc m(13,2)=2+1=3.
1+m(3,3) : Si on utilise un bocal de 10, il reste S=3. On utilise forcément V[1]=1 car c'est le seul bocal pouvant être utilisé à partir de S=3. On en utilise 3, m(3,3)=3.
Donc 1+m(3,3)=1+3=4. Et m(13,3)=min(3,4)=3. La solution optimale est 2×6+1, soit 3 bocaux au total.
Le système de capacité V=[1,6,10] est donc non glouton-compatible, car l'algorithme glouton ne produit pas la solution optimale. Absence de la propriété de divisibilité dans V, car 10 n'est pas multiple de 6.

Question 10 :
En utilisant la propriété mentionnée sur les algorithmes glouton-compatibles, si k=2, nous avons donc un système de capacités V de taille 2 et sachant que V[1]=1, si l'on pose V[2]=x, on peut simplement constater que x×1=x (x est un entier, donc, est multiple de 1). Donc, tout système de capacité V avec k=2 est glouton-compatible.
On peut s'en assurer :
Si S est divisible par V[2] (c'est-à-dire que S peut être complètement rempli en utilisant seulement des bocaux de taille V[2]), l'algorithme glouton choisira simplement S/V[2] bocaux de taille V[2], ce qui est optimal.
Si S n'est pas divisible par V[2], l'algorithme glouton choisira d'abord autant de bocaux de taille V[2] que possible, et la quantité restante sera exactement S mod V[2] (résultat compris entre 1 et x-1), puis l'algorithme choisira des bocaux de taille V[1]=1 pour remplir ce qu'il reste, ce qui est également optimal car on minimise ainsi le nombre de bocaux V[1]=1.

Question 11 :
La condition $k \geq 3$ est en O(1).
S varie de V[3]+2 à V[k−1]+V[k]−1, ce qui donne comme nombre de valeurs prises par S :
dernier terme - premier + 1 = (V[k−1]+V[k]−1)−(V[3]+2)+1 = V[k−1]+V[k]−V[3]−2 valeurs.
j allant de 1 à k, parcourt k valeurs. On a donc k exécutions par valeur de S.
Dans chaque boucle, on vérifie la condition V[j]<S, qui s'exécute en O(1), puis 2 appels à "AlgoGlouton", dont chaque appel à une complexité de O(S).
Le nombre de valeurs prises par S peut s'évaluer en O(V[k−1]+V[k]−V[3]), où l'on néglige la constante additive -2 qui ne change pas la complexité totale.

Avec le nombre d'appels de l'algorithme "AlgoGlouton" en fonction du nombre de couples (S,k) considérés (2 appels à l'algorithme par couple), et sachant que 2×O(S) se simplifie en O(S), la complexité temporelle de l'algorithme "TestGloutonCompatible" est donc de O(k×(V[k−1]+V[k]−V[3])×S).

La différence majeure entre une complexité polynomiale et une complexité pseudo-polynomiale réside dans le fait que le temps d'exécution d'un algorithme dépend directement de la taille en nombre de bits de l'entrée, qui dans notre cas est S, (complexité polynomiale) ou s'il dépend de sa valeur numérique (complexité pseudo-polynomiale). Dans "TestGloutonCompatible", la complexité est O(S) pour "AlgoGlouton" et dépend directement de la valeur S, pas de sa taille en bits. En application, un algorithme pseudo-polynomial comme celui-ci, ne sera pas efficace pour une grande valeur d'entrée S, même si cette entrée est représentée sur peu de bits. En revanche, un algorithme polynomial serait beaucoup plus rapide à s'exécuter pour la même valeur S. L'algorithme "TestGloutonCompatible" est donc de complexité pseudo-polynomiale.

En conclusion, si notre algorithme "TestGloutonCompatible" renvoie vrai, nous indiquant que le système de capacités V utilisé est glouton-compatible, on peut utiliser l'algorithme glouton, de complexité polynomiale, dont l'exécution sera donc très efficace pour obtenir le nombre

minimum de bocaux qu'il est possible d'utiliser en fonction de S, V et k, mais seulement nous avons V et k fixés.
S'il renvoie faux, en revanche, on doit utiliser l'algorithme de programmation dynamique pour ce faire car l'algorithme glouton peut, dans ce cas, renvoyer un nombre de bocaux qui n'est pas minimal.

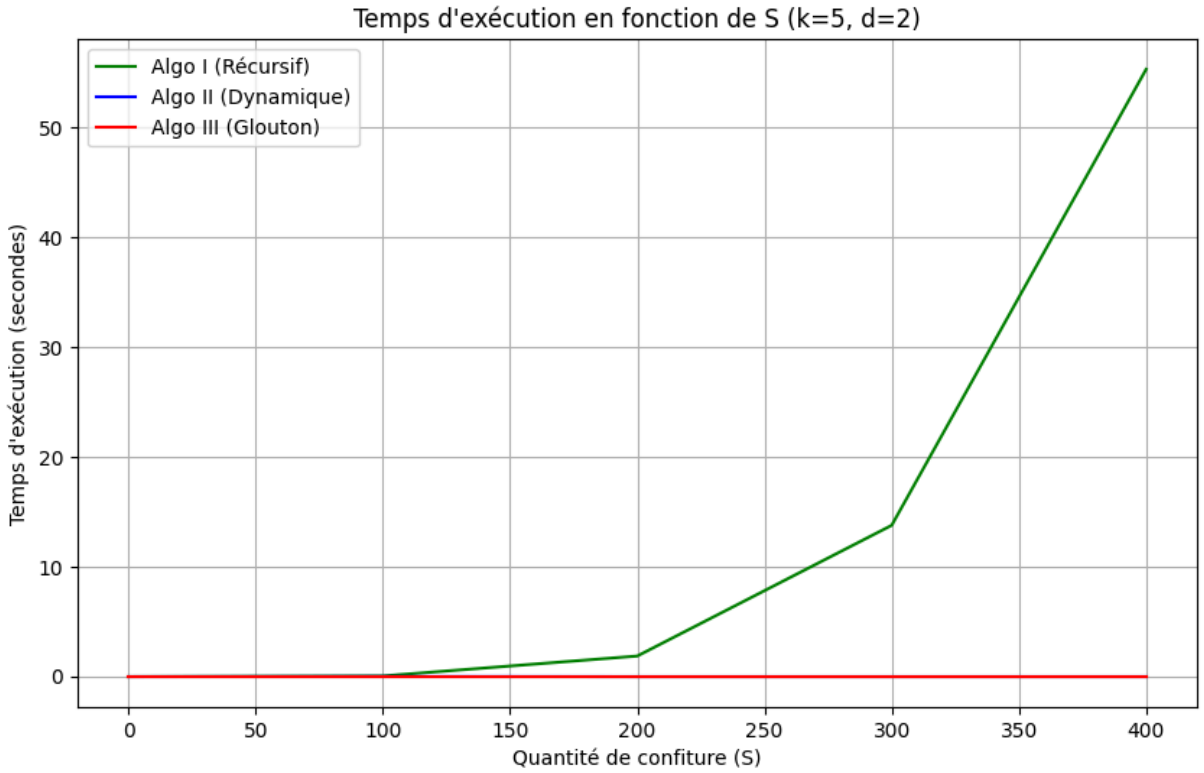
Mise en œuvre :

Question 12 :
Pour rappel, nous avons les complexités temporelles théoriques suivantes :
Algorithme 1 (récursif) : O(2^k).
Algorithme 2 (dynamique) : O(k×S).
Algorithme 3 (glouton) : O(S).
Les résultats pratiques doivent se justifier par ces résultats calculatoires.

Plusieurs paramètres modifient le temps d'exécution des différents algorithmes :
-S : pour la quantité de confiture à mettre dans les bocaux (le temps d'exécution augmente quand S augmente).
-k : le nombre de bocaux (le temps d'exécution augmente quand k augmente).
-d : le système de capacité Expo (le temps d'exécution diminue quand d augmente).
Rappel : Le système Expo est construit à partir d'une valeur entière d ≥ 2 : V [1] = 1, V [2] = d, V [3] = d^2, …, V [k] = d^k−1, et pour tout entier d ≥ 2, ce système de capacités Expo est glouton-compatible.

Nous utilisons dans les premiers graphiques des valeurs S et k assez faibles. Dans notre code, nous avons fait une étude du temps d'exécution en fonction de S et une deuxième étude en fonction de k, à chaque fois l'autre paramètre reste fixe pour identifier correctement les impacts sur le temps d'exécution du programme.

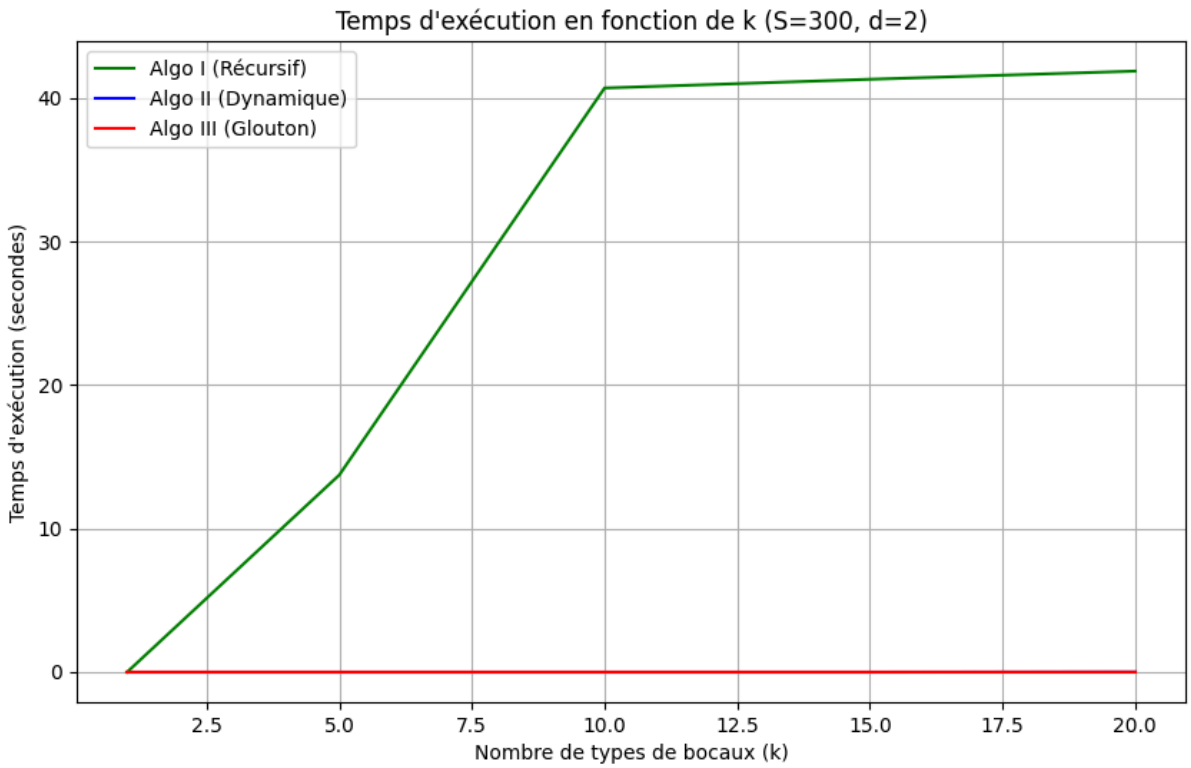
Les 6 premiers graphiques ont pour but de mettre en avant l'algorithme récursif (1), qui, même avec des paramètres faibles, a un temps d'exécution nettement supérieur aux autres (complexité exponentielle).



On devine que l'algorithme dynamique (bleu) s'exécute également instantanément (est caché derrière la droite rouge).
L'algorithme récursif semble avoir un temps d'exécution qui augmente de plus en plus vite avec S qui augmente de façon linéaire. Cela fait penser à une courbe exponentielle.

Affichage du terminal :
Tests pour d = 2
Test pour d=2, S=0, k=5
Test pour d=2, S=100, k=5
Test pour d=2, S=200, k=5
Test pour d=2, S=300, k=5
Test pour d=2, S=400, k=5
Test pour d=2, S=500, k=5
Algo I a dépassé 60s pour d=2, S=500, k=5

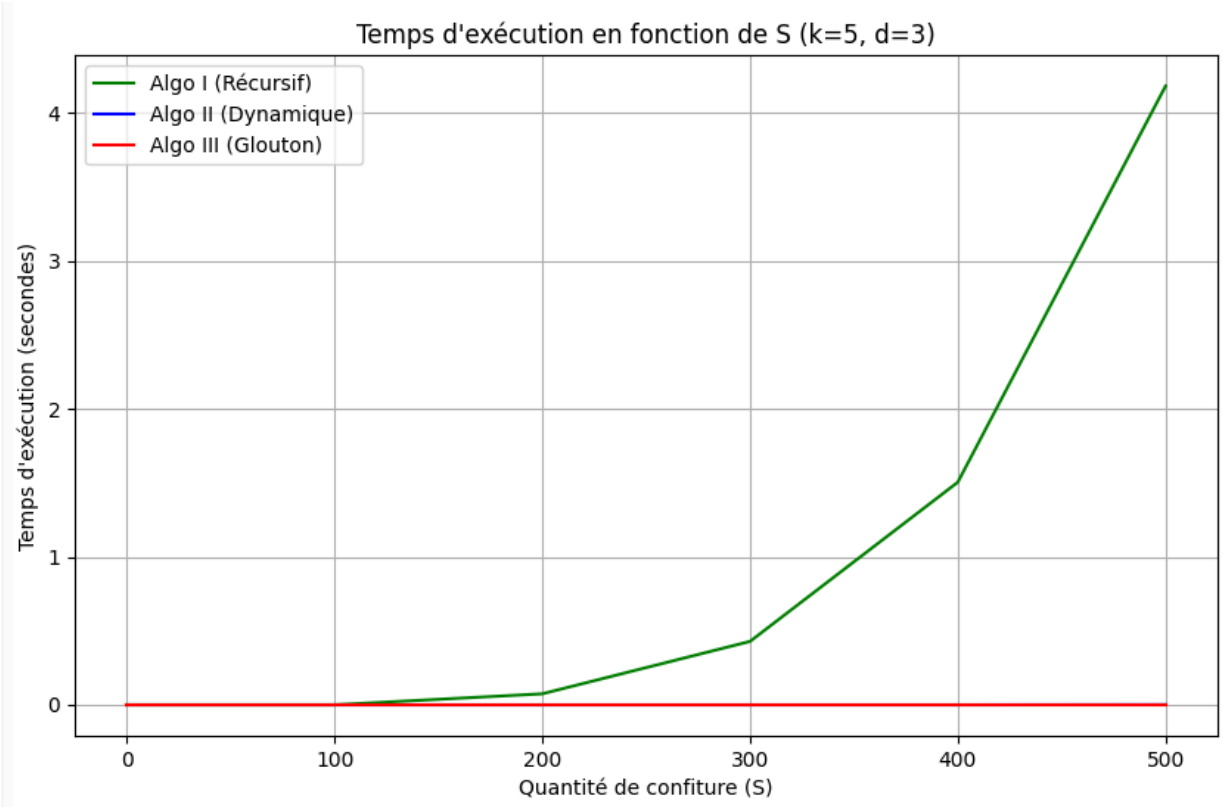
Les quantités S testées ici sont : 0, 100, 200, 300, 400, 500 dg.



Même observation pour les algorithmes 2 et 3 qu'en faisant varier S.
L'algorithme récursif semble avoir un temps d'exécution qui augmente très rapidement au départ, le fait que cela se tasse ensuite s'explique simplement par le fait que S soit fixé à 300, ajouter des bocaux plus grands que S n'a aucune utilité car ils ne sont pas utilisés.

Affichage du terminal :
Test pour d=2, S=300, k=1
Test pour d=2, S=300, k=5
Test pour d=2, S=300, k=10
Test pour d=2, S=300, k=15
Test pour d=2, S=300, k=20

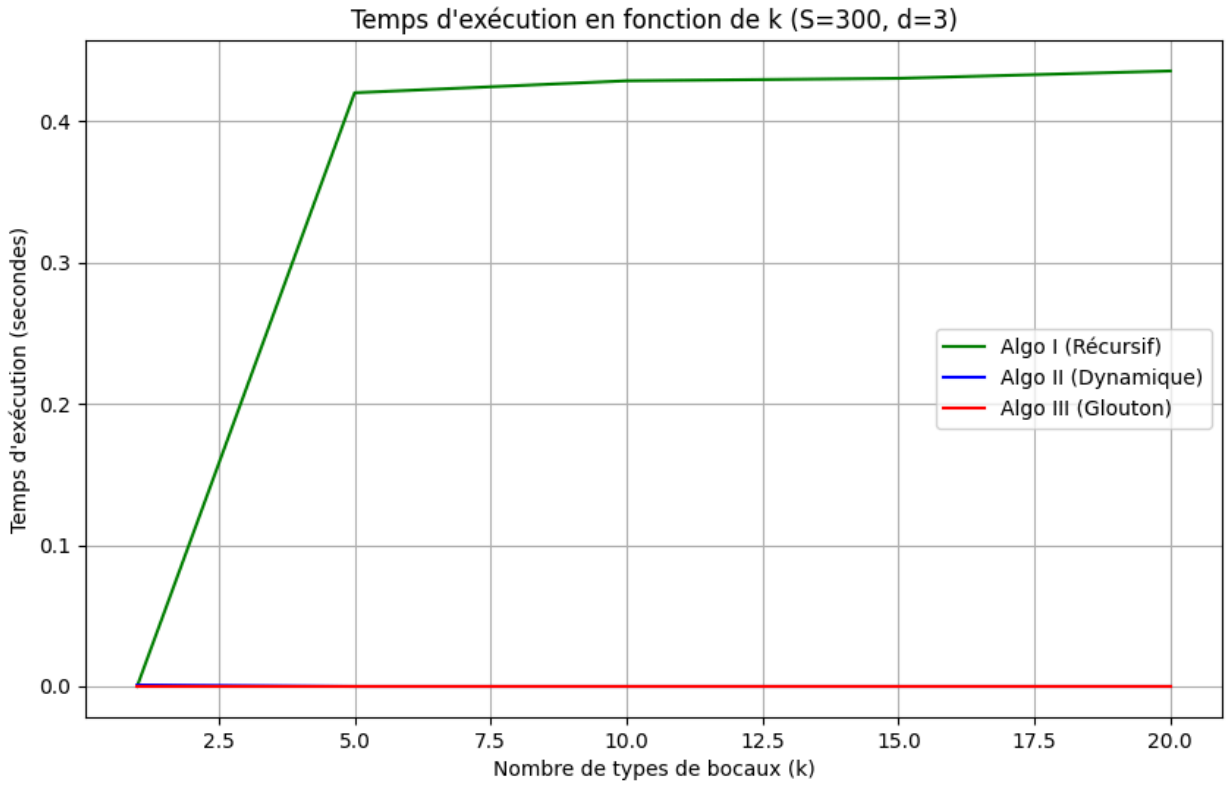
Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20.



Lorsque d=3, les capacités des bocaux (à partir de V[2]) augmentent, il y a moins de récursions à faire pour l'algorithme récursif, cela se voit sur le temps d'exécution environ 10 fois plus court. L'algorithme récursif semble avoir un temps d'exécution qui augmente de plus en plus vite avec S qui augmente de façon linéaire. On retrouve une courbe exponentielle.

Affichage du terminal :
Tests pour d = 3
Test pour d=3, S=0, k=5
Test pour d=3, S=100, k=5
Test pour d=3, S=200, k=5
Test pour d=3, S=300, k=5
Test pour d=3, S=400, k=5
Test pour d=3, S=500, k=5

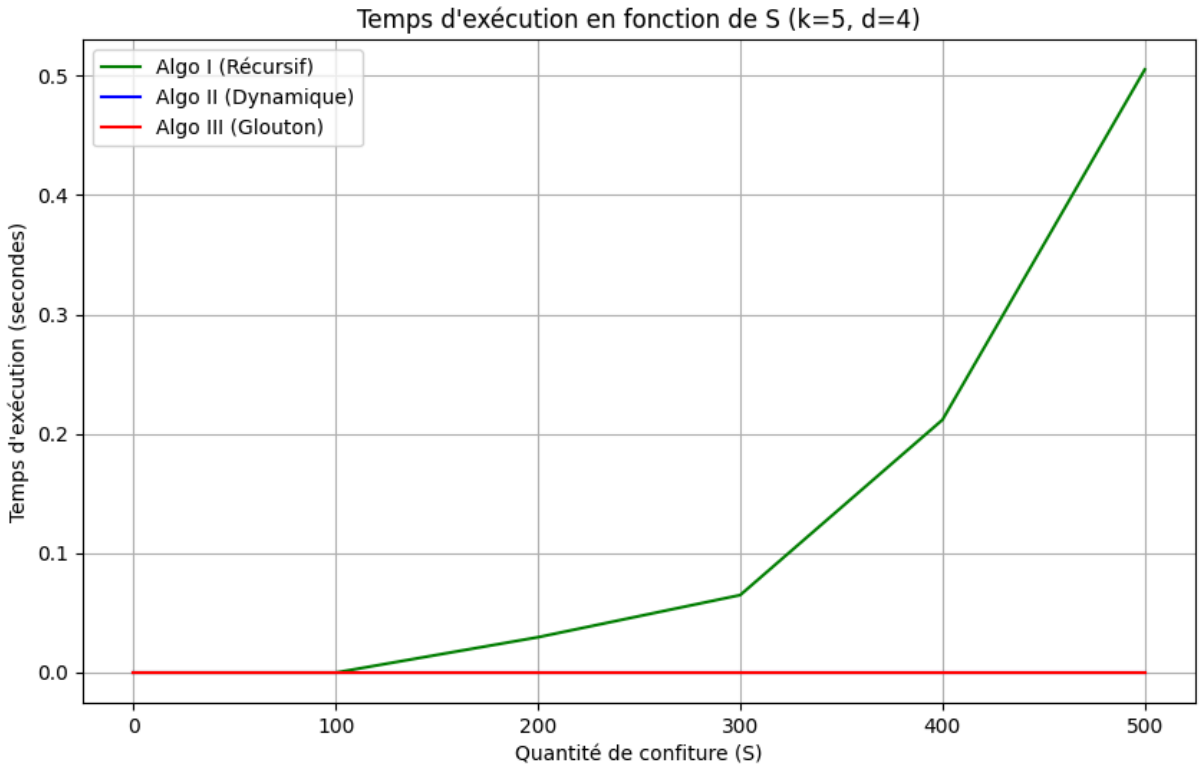
Les quantités S testées ici sont : 0, 100, 200, 300, 400, 500 dg.



Le temps d'exécution est également moindre avec d=3. Le même phénomène se répète à nouveau car l'algorithme récursif semble avoir un temps d'exécution qui augmente très rapidement au départ, le fait que cela se tasse ensuite s'explique simplement par le fait que S soit fixé à 300, ajouter des bocaux plus grands que S n'a aucune utilité car ils ne sont pas utilisés.

Affichage du terminal :
Test pour d=3, S=300, k=1
Test pour d=3, S=300, k=5
Test pour d=3, S=300, k=10
Test pour d=3, S=300, k=15
Test pour d=3, S=300, k=20

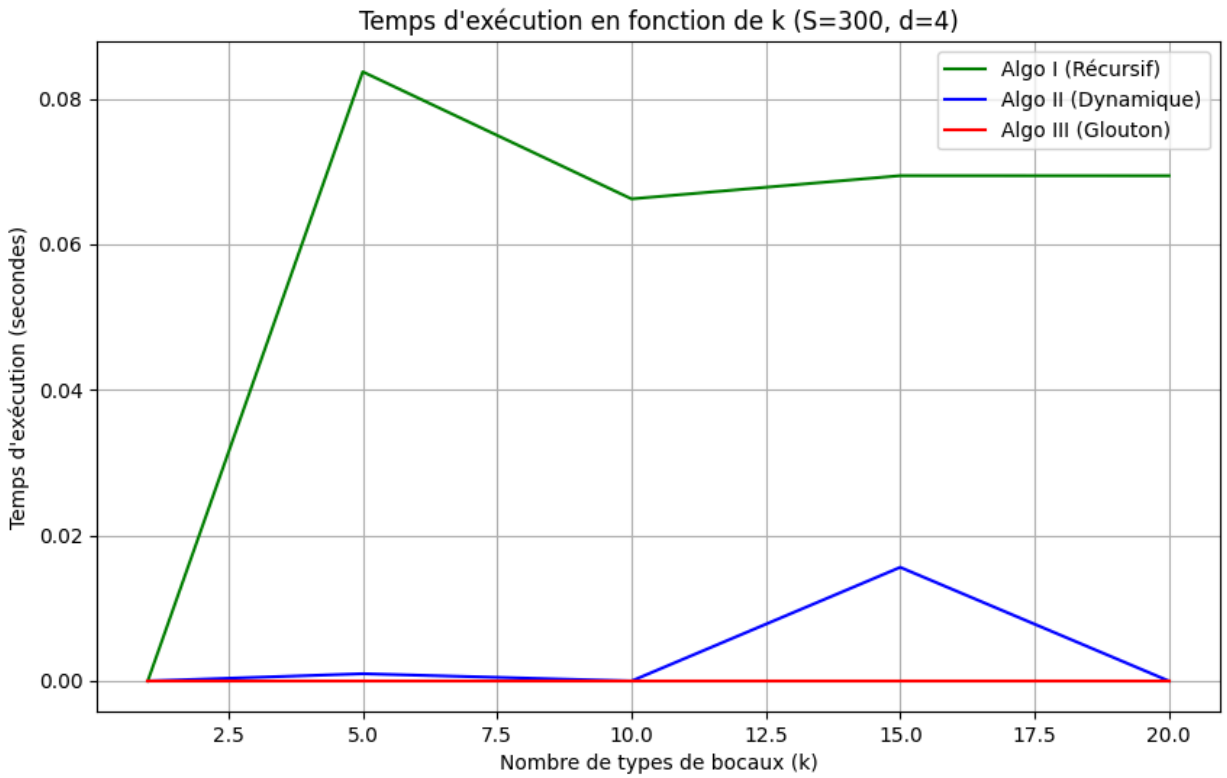
Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20.



Avec d=4, le temps d'exécution diminue encore par rapport à d=3. L'allure de la courbe de l'algorithme récursif est toujours identique, ressemble toujours à une courbe exponentielle.

Affichage du terminal :
Tests pour d = 4
Test pour d=4, S=0, k=5
Test pour d=4, S=100, k=5
Test pour d=4, S=200, k=5
Test pour d=4, S=400, k=5
Test pour d=4, S=500, k=5

Les quantités S testées ici sont : 0, 100, 200, 300, 400, 500 dg.



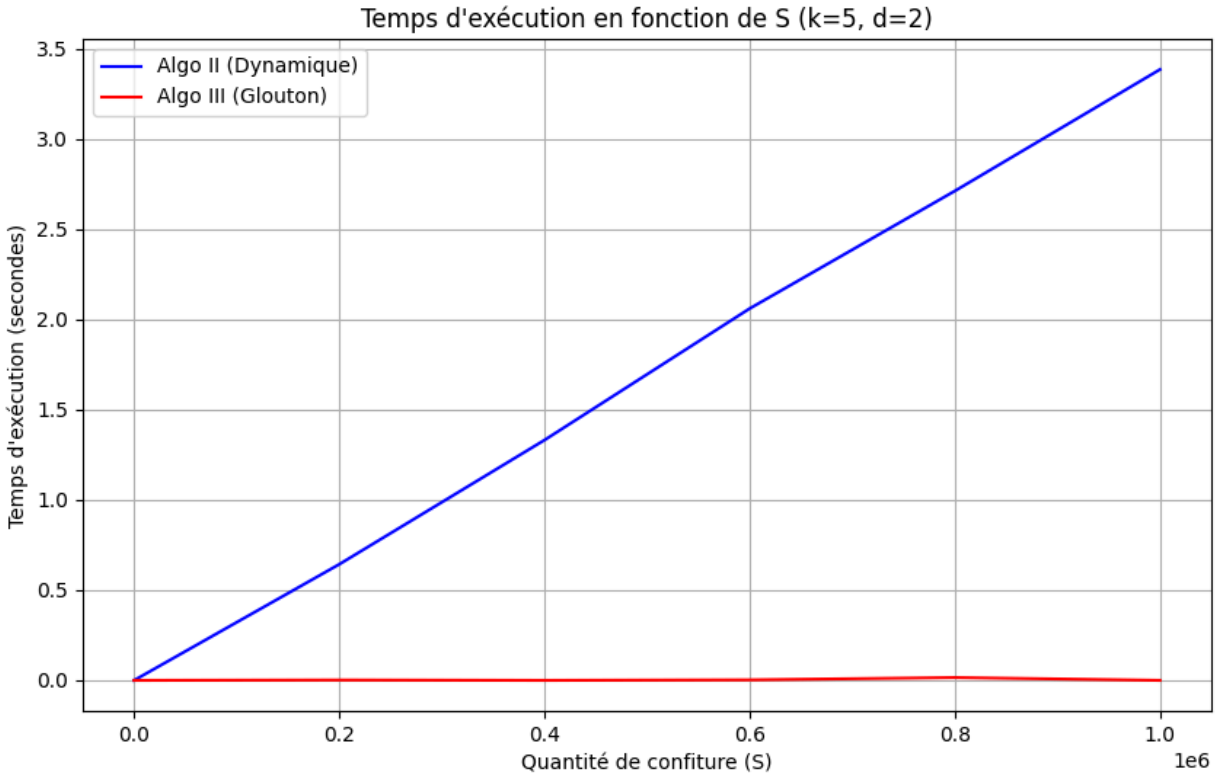
Affichage du terminal :
Test pour d=4, S=300, k=1
Test pour d=4, S=300, k=5
Test pour d=4, S=300, k=10
Test pour d=4, S=300, k=15
Test pour d=4, S=300, k=20

Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20.

A 0,01 seconde près, nous commençons à voir du changement, et l'algorithme dynamique ne reste plus à 0 seconde. En fait, le temps d'exécution des trois algorithmes est très court, cette fois-ci, l'algorithme récursif a aussi un temps d'exécution très faible. La précision sur le temps mis par les trois algorithmes étant élevée, cela peut se mêler à l'inconstance dans les calculs du CPU, qui ne fait pas ses calculs de façon constante, il y a des interférences et c'est normal. Il y a sûrement donc eu un tout petit ralentissement du CPU au niveau du calcul de k=15.
Même observation pour l'algorithme récursif que précédemment, le temps pris pour l'exécution se stabilise pour 10, 15 et 20 bocaux de taille différente car les plus grands bocaux ne sont pas utilisés.
La courbe verte aurait également l'allure d'une courbe exponentielle comme pour tous les autres cas si la valeur de S était suffisamment grande, or le temps de calcul lorsque d=2 dépassait la minute sur des tests plus gourmands (et nous devons arrêter le programme

lorsque cela se produit), la valeur fixée à 300 dg a été choisie de cette façon.
Également, la valeur fixée de k=5 pour les tests en fonction de S a été choisie pour éviter que l'algorithme récursif ne dépasse la minute, il y a énorméments d'appels récursifs sinon.
La complexité de l'algorithme récursif colle parfaitement avec les résultats.

Pour la suite, les tests effectués utilisent des valeurs nettement supérieures de S et k pour comparer en détail les algorithmes dynamique et glouton sur les temps d'exécutions, qui étaient quasi-nulles jusqu'ici.
Les deux algorithmes ont des complexités polynomiales et sont donc naturellement plus rapides à l'exécution.

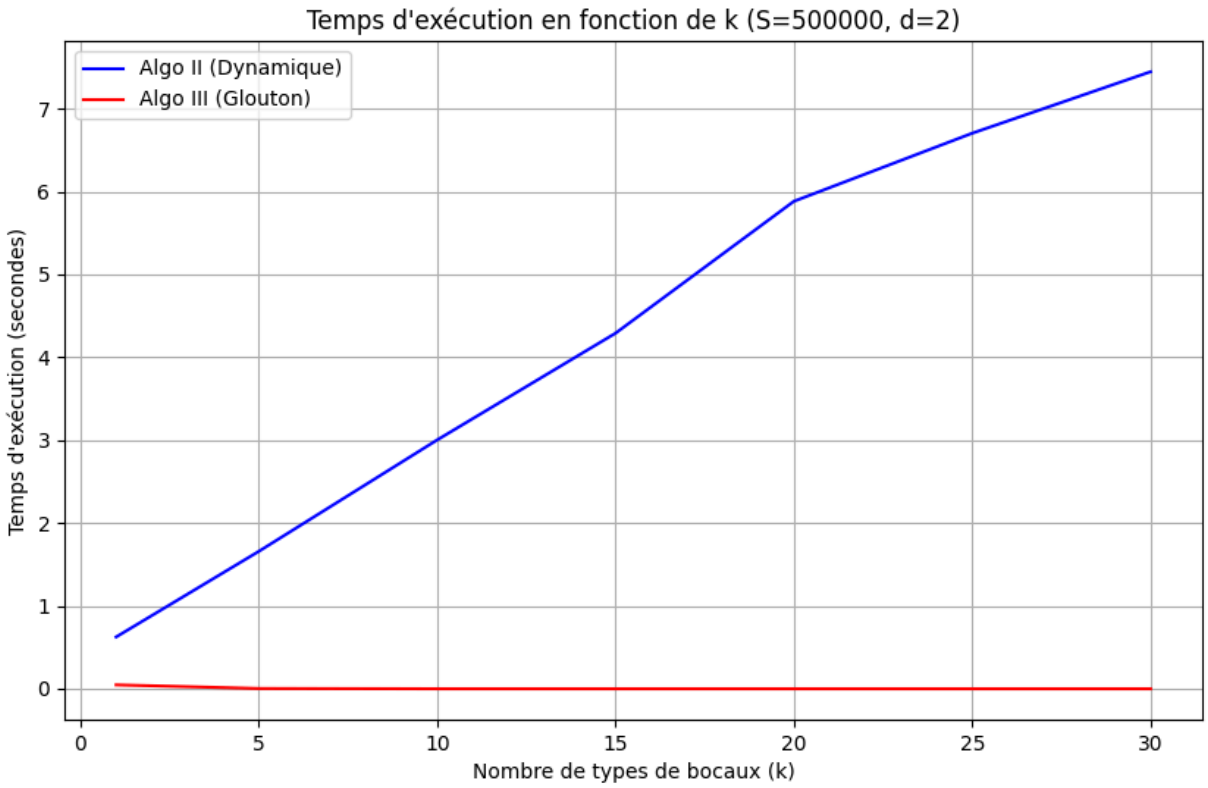


Nous avons conservé un système de capacité de 5 bocaux pour les tests suivants en fonction de S. Nous remarquons ici que nous avons une progression à peu de choses près rectiligne pour l'algorithme dynamique. La progression est quasi linéaire tout comme les intervalles de valeurs pour la quantité S (dg) de confiture qui va de 200000 en 200000 dg. Ce résultat est cohérent avec nos observations dans le calcul de la complexité de l'algorithme dynamique.
L'algorithme glouton est très efficace, son exécution est presque instantanée, on se rend compte qu'il a moins de calculs à effectuer, étant donné que le système Expo (ici avec d=2) est glouton-compatible. Il récupère les bocaux les plus grands possibles sans se soucier de la solution optimale (qu'il réalise puisque notre système est glouton-compatible).
L'algorithme dynamique, lui, fait les tests et son temps d'exécution augmente linéairement en

fonction du nombres de tests pour chercher la solution optimale au problème.

Affichage du terminal :
Tests pour d = 2
Test pour d=2, S=0, k=5
Test pour d=2, S=200000, k=5
Test pour d=2, S=400000, k=5
Test pour d=2, S=600000, k=5
Test pour d=2, S=800000, k=5
Test pour d=2, S=1000000, k=5

Les quantités S testées ici sont : 0, 200000, 400000, 600000, 800000, 1000000 dg.

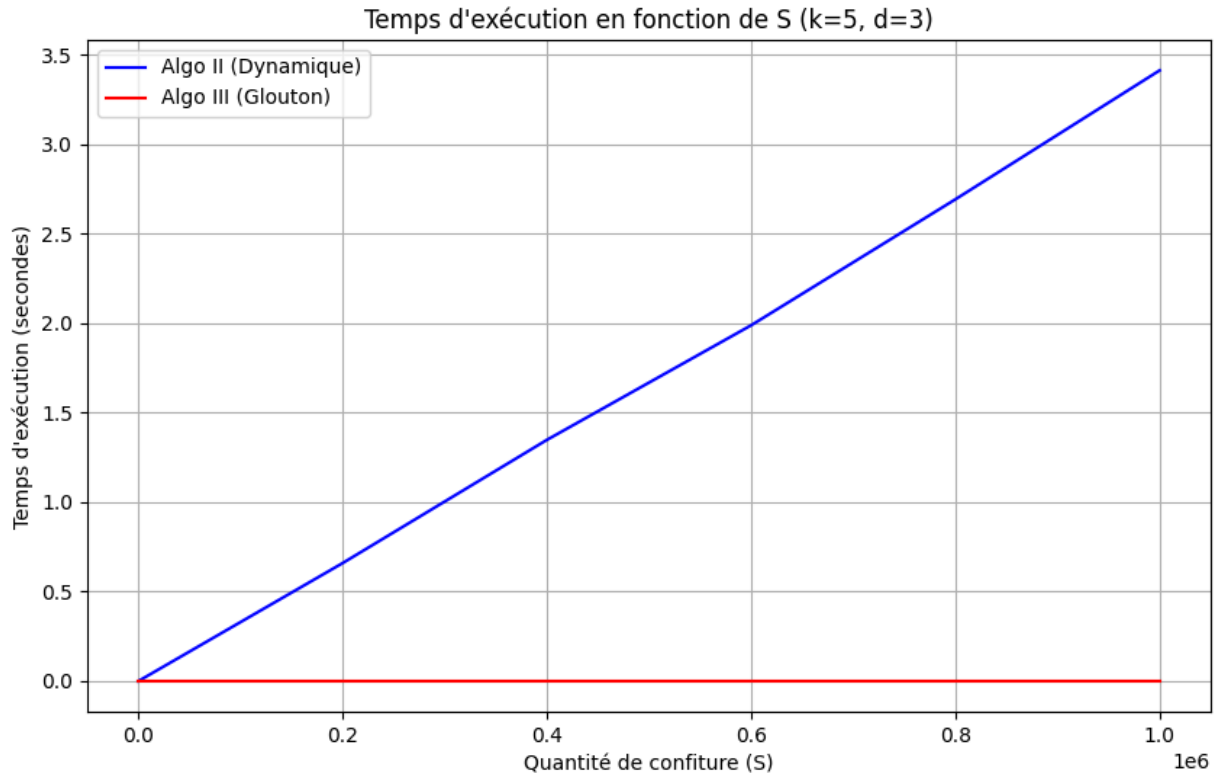


Nous avons ajouté des calculs avec 25 et 30 types de bocaux. Nos tests sont effectués sur une quantité de 500000 dg de confiture. Nous remarquons que comme pour l'évaluation en fonction de la quantité S de confiture, ici, pour l'évaluation du temps en fonction du nombres de types de bocaux disponibles, nous avons également une progression quasi linéaire sur la courbe, dont le temps d'exécution voit sa progression ralentir entre 20 et 30 bocaux. Comme dans les précédentes observations avec l'algorithme récursif, nous semblons avoir ici suffisamment de bocaux de grandes capacités. Ce ralentissement peut s'expliquer par le fait que les bocaux de plus grande taille sont inutiles car trop grands pour S. L'allure de la courbe est conforme à nos

attentes théoriques avec une complexité polynomiale. L'algorithme glouton, comme expliqué, est optimal pour le choix des bocaux et ne fait aucun test vérifiant s'il choisit bien le minimum possible de bocaux pour stocker toute cette confiture. Sa complexité étant meilleure que celle de l'algorithme dynamique, ces résultats sont tout-à-fait légitimes.

Affichage du terminal :
Test pour d=2, S=500000, k=1
Test pour d=2, S=500000, k=5
Test pour d=2, S=500000, k=10
Test pour d=2, S=500000, k=15
Test pour d=2, S=500000, k=20
Test pour d=2, S=500000, k=25
Test pour d=2, S=500000, k=30

Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20, 25, 30.

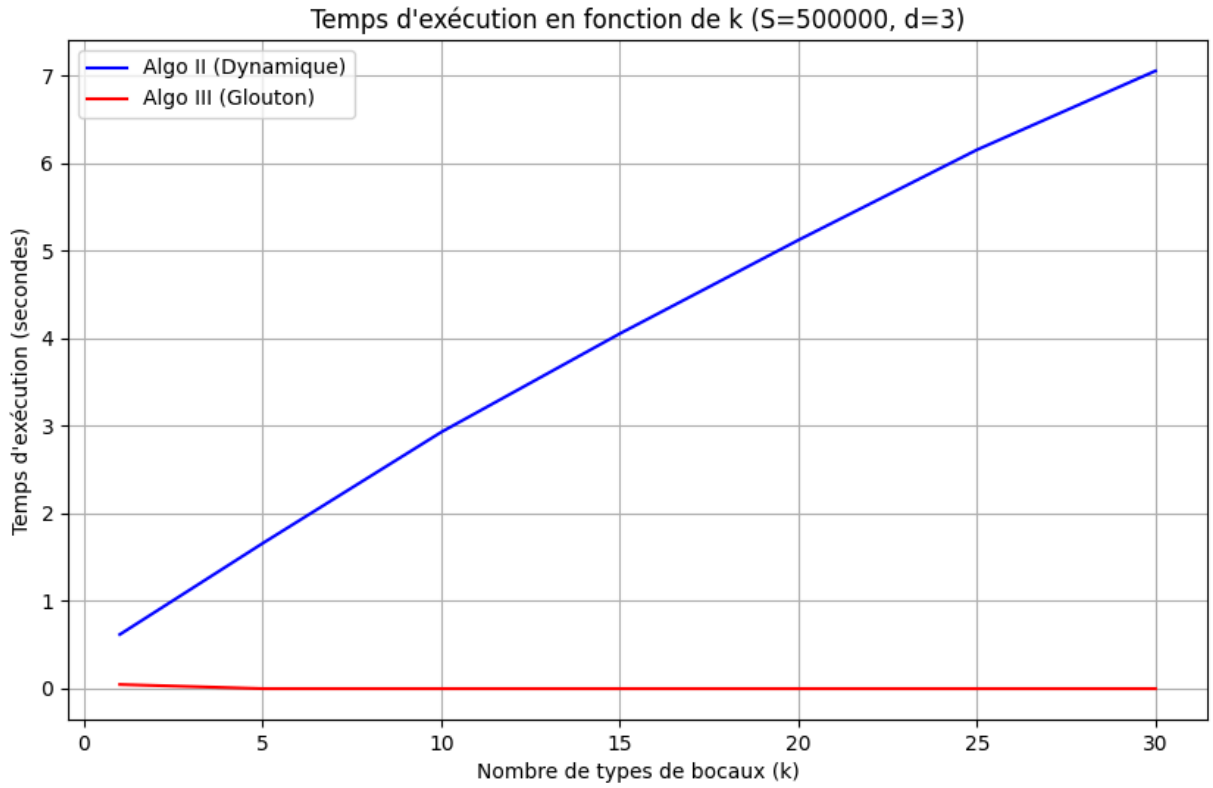


Avec d=3, le temps d'exécution semble tout aussi rapide, les capacités des différents bocaux sont plus grandes mais l'impact est non significatif par rapport à l'algorithme récursif. Les conclusions sont les mêmes que pour d=2.

Affichage du terminal :
Tests pour d = 3

Test pour d=3, S=0, k=5
Test pour d=3, S=200000, k=5
Test pour d=3, S=400000, k=5
Test pour d=3, S=600000, k=5
Test pour d=3, S=800000, k=5
Test pour d=3, S=1000000, k=5

Les quantités S testées ici sont : 0, 200000, 400000, 600000, 800000, 1000000 dg.

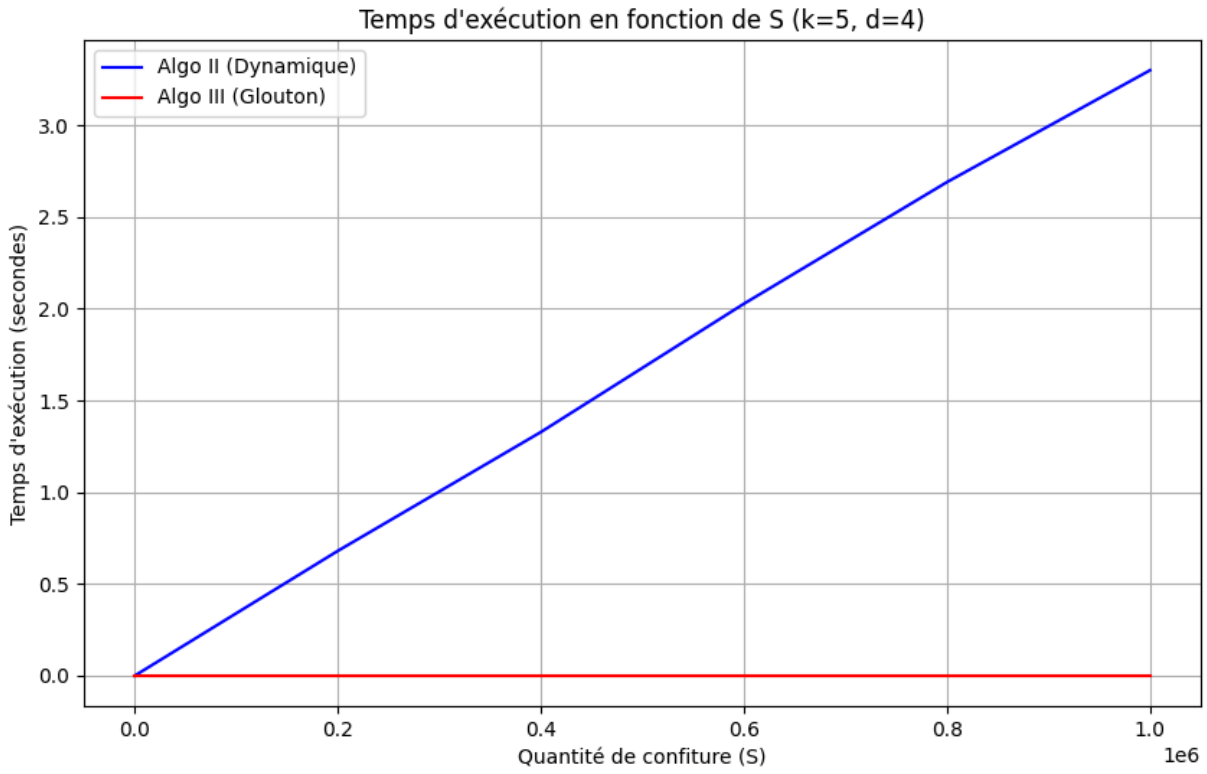


Avec d=3, on gagne environ 0,5 seconde en considérant 30 types de bocaux par rapport au système Expo où d=2. Le temps d'exécution semble ralentir tout du long et non seulement pour les plus grands nombres de types de bocaux différents. Cela se justifie par le fait que les bocaux (à partir de V[2]) sont de taille supérieure, il nécessite d'utiliser moins de types de bocaux différents pour notre quantité S qui reste fixe. Nous arrivons donc plus rapidement au total S.

Affichage du terminal :
Test pour d=3, S=500000, k=1
Test pour d=3, S=500000, k=5
Test pour d=3, S=500000, k=10
Test pour d=3, S=500000, k=15

Test pour d=3, S=500000, k=20
Test pour d=3, S=500000, k=25
Test pour d=3, S=500000, k=30

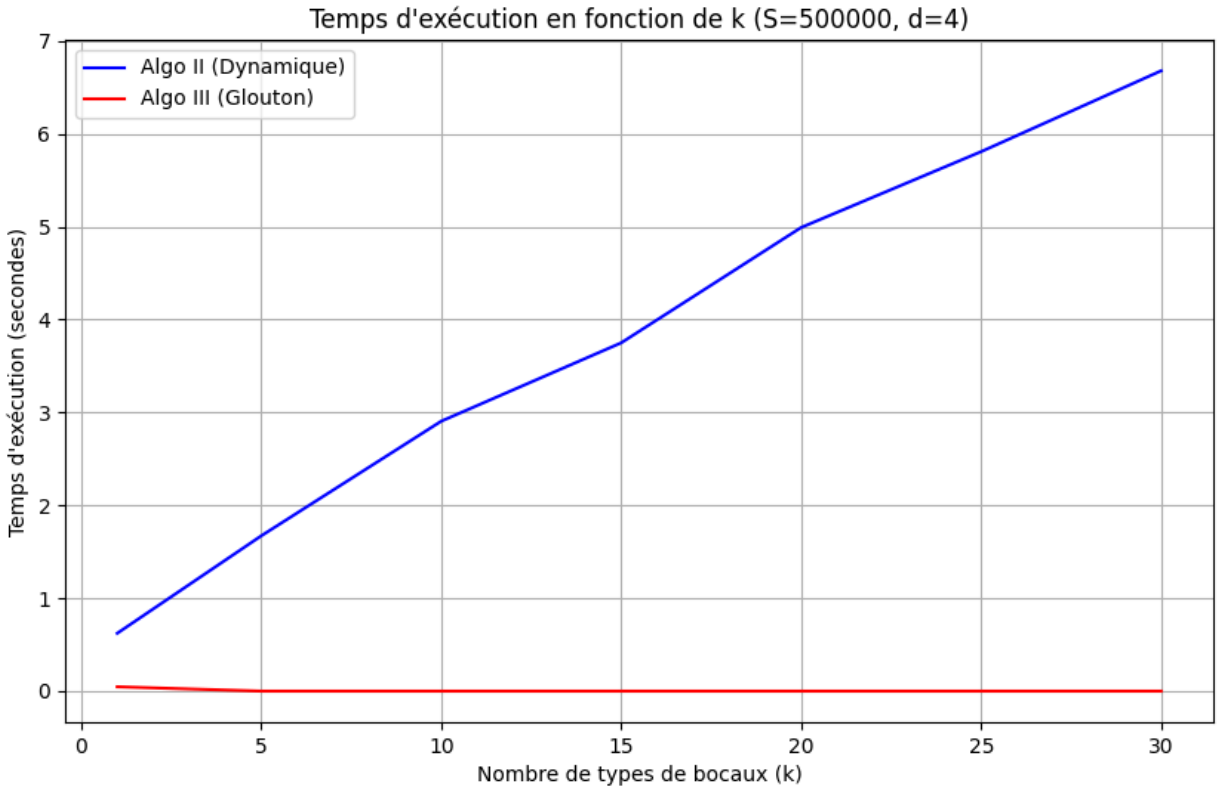
Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20, 25, 30.



Pour d=4, l'algorithme dynamique gagne environ 0,1 seconde lorsque S=1000000. Les résultats sont très similaires pour chaque valeur de d. L'algorithme Glouton, bien qu'il est difficile de le remarquer ici (il faudrait pouvoir atteindre une valeur de S nettement supérieure), se rapproche de l'exécution instantanée par rapport à d=2 et d=3. Il y a moins de bocaux de capacité inférieure à S donc moins de calculs pour le remplissage.

Affichage du terminal :
Tests pour d = 4
Test pour d=4, S=0, k=5
Test pour d=4, S=200000, k=5
Test pour d=4, S=400000, k=5
Test pour d=4, S=600000, k=5
Test pour d=4, S=800000, k=5
Test pour d=4, S=1000000, k=5

Les quantités S testées ici sont : 0, 200000, 400000, 600000, 800000, 1000000 dg.



Avec d=4, nous gagnons encore 0,5 seconde environ pour k=30, soit 1 seconde de moins que pour d=2. Il y a moins de bocaux de capacité inférieure à S donc moins de calculs pour le remplissage. L'augmentation du temps semble toujours ralentir peu à peu. Avec 15 bocaux différents, le calcul a été certainement un peu plus rapide dû au CPU car il n'y a pas de raison pour que le temps d'exécution augmente petit à petit de moins en moins vite.

Affichage du terminal :
Test pour d=4, S=500000, k=1
Test pour d=4, S=500000, k=5
Test pour d=4, S=500000, k=10
Test pour d=4, S=500000, k=15
Test pour d=4, S=500000, k=20
Test pour d=4, S=500000, k=25
Test pour d=4, S=500000, k=30

Les nombres de bocaux k testés ici sont : 1, 5, 10, 15, 20, 25, 30.

Les temps d'exécution sont donc cohérents et suivent les résultats théoriques obtenus sur l'étude des complexités respectives des algorithmes récursif, dynamique et glouton.

temps glouton ≤ temps dynamique ≤ temps récursif, tout comme O(S) est de complexité inférieure ou égale à O(k×S) qui est strictement inférieure à O(2^k).

Question 13 :
Si le système n'est pas compatible, la différence entre la solution glouton et la solution optimale est calculée à l'aide de l'algorithme dynamique (2) et de l'algorithme glouton (3).
Pour le calcul des écarts relatifs, pour chaque système non compatible, l'écart relatif entre la solution glouton et la solution optimale se calcule ainsi :

$$\text{Écart relatif} = \frac{\text{Solution glouton} - \text{Solution optimale}}{\text{Solution optimale}}$$

Par exemple, avec k=10 et pmax=100, donc les bocaux V[2] à V[10] ont des capacités aléatoires entre 2 et 100 dg :
La fréquence d'apparition de systèmes de capacités glouton-compatibles a été mesurée à partir de 1000 tests (générations aléatoires de systèmes de capacités respectant les paramètres k et pmax).

Résultats :
Compatibilité glouton : 0.00%
Non compatibilité glouton : 100.00%
Écart relatif moyen : 0.8112
Écart relatif maximal : 13.5000

Cela indique que tous les systèmes générés ne sont pas glouton-compatibles. Avec de nombreux autres essais, il est rare d'obtenir une compatibilité glouton différente de 0% en utilisant ces paramètres, simplement car les capacités générées sont trop nombreuses et ont un intervalle de valeurs possibles trop large. Cela empêche l'algorithme glouton de donner une solution optimale dans chaque cas.
L'écart relatif moyen indique que la solution glouton est 81% moins efficace que la solution optimale en moyenne sur l'ensemble des 1000 générations aléatoires réalisées sur cet exemple.
L'écart relatif maximal montre que dans le(s) pire(s) système(s) parmi des 1000 générés, l'algorithme glouton a été jusqu'à 13.5 fois moins efficace que la solution optimale de l'algorithme dynamique.
L'algorithme glouton n'est donc pas une bonne solution si les valeurs du système de capacités sont nombreuses avec une capacité qui peut varier selon énormément de possibilités. (et il n'y a ici que 100-1=99 capacités possibles pour les 9 bocaux, avec plus, il serait encore moins probable de tomber par hasard sur un système de capacité ayant la propriété d'être glouton-compatible).

Pour k=1 et k=2, indépendamment de la valeur de pmax, nous obtenons toujours à l'exécution :
Compatibilité glouton : 100.00%
Non compatibilité glouton : 0.00%
Aucun écart relatif calculé (aucun système non glouton-compatible testé avec succès).

Avec 1 ou 2 types de bocaux différents, nous avons toujours un système de capacité glouton-compatible (comme vu dans une partie précédente) donc le résultat obtenu est cohérent.

Un autre exemple avec k=3 et pmax=100, donc les bocaux V[2] à V[3] ont des capacités aléatoires entre 2 et 100 dg :
La fréquence d'apparition de systèmes de capacités glouton-compatibles a été mesurée à partir de 1000 tests.

Résultats :
Compatibilité glouton : 24.10%
Non compatibilité glouton : 75.90%
Écart relatif moyen : 0.2672
Écart relatif maximal : 27.0000

Il y a donc environ ici ¼ chance que l'algorithme glouton donne la solution optimale selon les résultats. Mais nous remarquons aussi qu'il a utilisé 27 fois plus de bocaux que l'algorithme dynamique dans le pire cas parmi les 1000 tirages, très peu efficace dans ce cas.

Un dernier exemple intéressant avec k=x, pmax=x. Dans ce cas, nous obtenons avec 100% de probabilité le résultat suivant :
Compatibilité glouton : 100.00%
Non compatibilité glouton : 0.00%
Aucun écart relatif calculé (aucun système non glouton-compatible testé avec succès).

Si k=pmax alors notre système de capacité V est forcément glouton-compatible et dans ce cas il est préférable d'utiliser l'algorithme glouton que l'algorithme dynamique qui met plus de temps à s'exécuter (ici, tous les bocaux de toutes les capacités disponibles peuvent être utilisés pour remplir la confiture, le résultat est donc évident).

En règle générale, plus k est petit, moins il y a de types de bocaux disponibles donc il est plus probable que les systèmes soient glouton-compatibles. Plus pmax est grand, plus il y a de possibilités différentes pour les capacités de chaque bocal, donc moins il y a de chance qu'un système tiré aléatoirement soit glouton-compatible.

Question 14 :
En reprenant le premier exemple avec k=10 et pmax=100, donc les bocaux V[2] à V[10] ont des capacités aléatoires entre 2 et 100 dg :
La fréquence d'apparition de systèmes de capacités glouton-compatibles a été mesurée à partir de 100 tests (générations aléatoires de systèmes de capacités respectant les paramètres k et pmax). Nous avons choisi f=3, donc on fait les calculs pour S=100 à S=300 à chaque test.

Résultats :
Compatibilité glouton : 0.00%
Non compatibilité glouton : 100.00%
Écart relatif moyen : 1.1531
Écart relatif maximal : 19.5000

Tous les systèmes générés ne sont pas glouton-compatibles, comme précédemment.
L'écart relatif moyen indique en moyenne, par rapport à la solution optimale, quel est le surcoût causé par l'utilisation de la méthode gloutonne. Ici, en moyenne, la solution gloutonne a coûté environ 1.15 fois plus que la solution optimale (115% plus de bocaux utilisés).
L'écart maximal est très élevé, ce qui suggère qu'il existe des cas où la solution gloutonne est nettement moins optimale que la solution exacte, avec un surcoût allant jusqu'à 19.5 fois la solution optimale donnée par l'algorithme dynamique.

Ici on a mesuré des statistiques sur certaines valeurs de S, sur des systèmes de capacité qui ne sont pas gloutons-compatibles, à la différence de la question précédente où nous ne tenions pas compte de f et donc des différentes valeurs de S (le système de capacité n'est pas glouton-compatible s'il existe des valeurs de S qui ne donneront pas la solution optimale avec l'algorithme glouton).