3I003 : ALGORITHMIQUE

Mini Projet : Confitures

3 Décembre 2018

Par :

Hichem Rami AIT EL HARA

Celina HANOUTI

Table des matières

Introduction

Dans ce projet, on s'intéresse à un problème d'optimisation combinatoire qui modélise la mise en bocal d'une quantité de confiture donnée. L'objectif est de concevoir un algorithme permettant de remplir le moins possible de bocaux tout en garantissant que ces derniers seront remplis exactement à leurs capacités maximales.

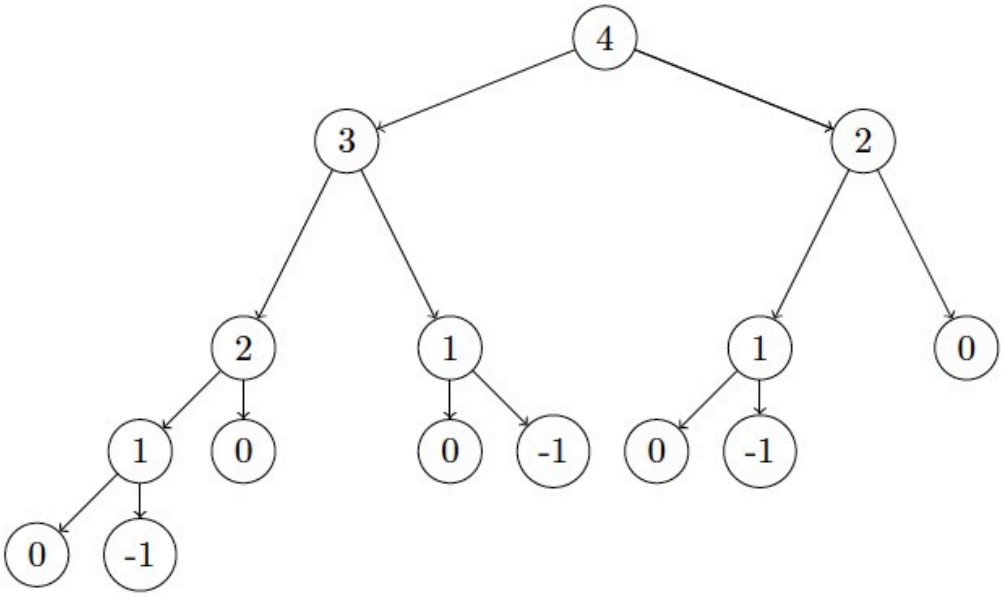
Pour répondre à cette problématique, nous étudierons plusieurs algorithmes ainsi que leurs complexités en faisant des analyses théoriques et expérimentales. Étant donnés un tableau V contenant k types de bocaux de capacités distinctes et une quantité S de confitures, ces algorithmes retourneront, d'une part, le nombre minimum de bocaux tels que la somme de leurs capacités soit égale à S, et d'autre part, un tableau A indiquant, pour chaque bocal, le nombre d'exemplaires à remplir pour chaque capacité.

2 Partie théorique

2.1 Algorithme I : Recherche exhaustive

Dans un premier temps, on s'intéresse à un algorithme dit « naïf » qui consiste à effectuer une recherche exhaustive des solutions. Remarquons d'abord qu'il existe toujours une solution en remplissant uniquement des bocaux de 1dg donc dans le pire des cas, l'algorithme retournera S bocaux. On initialise d'abord une variable *nbCont* à une valeur « grande » ou « pire cas » c'est à dire S. On parcourt ensuite le tableau de bocaux et on effectue une mise à jour de *nbCont* si un bocal nous permet d'avoir une valeur inférieure à la valeur courante. Cet algorithme est bien de type diviser pour régner étant donné que l'on a découpé le problème initial en sous-problèmes et qu'ensuite nous avons calculé une solution à ce problème initial à partir des solutions de ces sous-problèmes.

Soit l'arbre des appels récursifs pour l'algorithme RechercheExhaustive sur l'instance $V = [1,2]$, $k = 2$, $S = 4$.



Chaque noeud est étiqueté par la valeur de la quantité restante lors de l'appel récursif. On constate qu'il y'a plusieurs appels récursifs réalisés avec la même valeur de S, on peut alors conjecturer une complexité exponentielle.

Question 1

Montrons par récurrence forte sur la quantité s la validité et la terminaison de l'algorithme RechercheExhaustive.

Soit la propriété $P(r)$: « RechercheExhaustive(k, V, r) se termine et est valide i.e retourne le nombre minimum de bocaux nécessaires pour une quantité de confiture r »

Base : pour une quantité nulle, $s = 0$, RechercheExhaustive(k, V, s) se termine et renvoie *0* ce qui est trivialement vrai : pour une quantité nulle de confiture, on a besoin de 0 bocaux.

Étape inductive : Supposons $P(q)$ vraie pour toute quantité $q < s$ et montrons $P(s)$.

Considérons l'appel RechercheExhaustive(k, V, s). On sait qu'il existe toujours une solution en remplissant uniquement des bocaux de 1dg, cela se produit dans le cas où on disposerait uniquement de bocaux de capacité 1dg, d'où l'initialisation de *NbCont* à s. Pour tout $i \in \{1...k\}$, on effectue un appel récursif avec une quantité (s - V[i])$< s$ et par hypothèse d'induction RechercheExhaustive((k, V, s-V[i]) se termine et retourne le nombre minimum de bocaux nécessaire pour une quantité s – V[i]. Soit X_i la valeur retournée par RechercheExhaustive(k, V, s–V[i]) pour chaque $i \in \{1...k\}$. Remarquons que pour chaque i ∈ {1...k} et pour une quantité s, il faudra X_i bocaux + un bocal de capacité V[i] donc le nombre minimum de bocaux nécessaire pour une quantité s est définie par : min(X_i+1) pour $i \in \{1...k\}$.

Montrons par récurrence qu'à la i-ème itération on a bien :

$$NbCont = \min_{i \in 1..i} X_i + 1.$$

On initialise *NbCont* à s car dans le pire cas on aura recours à la solution qui consiste à remplir que des bocaux de 1dg.

Supposons la propriété vraie au rang i et montrons qu'elle l'est au rang $i+1$. À l'itération $i+1$, le test dans la boucle « pour » nous donne $NbCont = min(\min_{i' \in 1...i} X'_i + 1, X_{i+1} + 1) = \min_{i' \in 1...i+1} X'_i + 1$

Donc la propriété est vérifiée à l'itération $i+1$. Ceci est particulièrement vraie à l'issue de la dernière itération k, autrement dit, *NbCont* sera égale à $\min(X_i+1)$ pour $i \in \{1...k\}$.

Conclusion : RechercheExhaustive se termine et retourne bien le nombre minimum de bocaux nécessaires pour une quantité de confiture s.

Question 2

a.

Soit $a(s)$ le nombre d'appels récursifs effectués par RechercheExhaustive(2,V, s) avec V=[1,2] définie par :

$$a(s) = \begin{cases} 0 \text{ si } s \leq 0 \\ a(s-1) + a(s-2) + 2 \text{ sinon} \end{cases}$$

b.

Montrons par récurrence double sur s la propriété suivante :

P(s): «pour tout entier s ≥ 0, on a $b(s) \leq a(s) \leq c(s)$»

Base : pour s=0, la propriété est vraie car on a $a(0) = b(0) = c(0)$

pour s=1, la propriété est également vérifiée, en effet, on a

$a(1)=a(0)+a(-1)+2 = 2 \leq c(1) = 2c(0) + 2 = 2$

et $a(1) \geq b(1)=2$

Étape inductive : Montrons que si la propriété est vérifiée au rang $(s-2)$ et au rang $(s-1)$, alors elle l'est aussi au rang s.

Supposons $P(s-1)$ et $P(s-2)$ et montrons $P(s)$.

D'un côté nous avons :

$$a(s) = a(s-1) + a(s-2) + 2$$
$$a(s) \leq c(s-1) + c(s-2) + 2 \quad (HR(s\text{-}2) \text{ et } HR(s\text{-}1))$$
$$a(s) \leq 2c(s-1) + 2 \quad (c(s) \text{ est croissante})$$

$$a(s) \leq c(s)$$

Donc on a $a(s) \leq c(s)$.

D'un autre coté nous avons :

$$\begin{aligned}
&a(\mathrm{s}) = a(s\text{ - }1) + a(s\text{ - }2) + 2 \\
&a(\mathrm{s}) \geq b(s\text{ - }1) + b(s\text{ - }2) + 2 && (HR(s\text{-}2) \text{ et } HR(s\text{-}1)) \\
&a(\mathrm{s}) \geq 2b(s\text{ - }2) + 2 && (b(s) \text{ est croissante}) \\
&a(\mathrm{s}) \geq b(\mathrm{s})
\end{aligned}$$

et on a bien $b(s) \leq a(s)$
Nous avons montré que si $P(s\text{-}1)$ et $P(s\text{-}2)$ vraies alors $P(s)$ est vraie.
Conclusion : D'après le principe de récurrence la propriété $P(s)$ est vérifiée, autrement dit, pour tout entier s $\geq$0, on a $b(\mathrm{s}) \leq a(\mathrm{s}) \leq c(\mathrm{s})$».

c) Pour retrouver le terme général de la suite c(s), on procède par substitution :

$$\begin{aligned}
&c(s) = 2c(s\text{ - }1) + 2 = 2(2c(s\text{ - }2) + 2) + 2 \\
&c(s) = 2^2 c(s\text{ - }2) + 2^2 + 2 \\
&c(s) = 2^3 \mathrm{c}(\mathrm{s}\text{ - }3) + 2^3 + 2^2 + 2 \\
&c(s) = 2^4 c(s\text{ - }4) + 2^4 + 2^3 + 2^2 + 2 \\
&\ldots \\
&c(s) = 2^s c(0) + \sum_{i=1}^{s} 2^i = \sum_{i=1}^{s} 2^i
\end{aligned}$$

$$\text{Donc } c(\mathrm{s}) = \sum_{i=1}^{s} 2^i = \frac{2(2^s - 1)}{2 - 1} = 2^{s+1} - 2$$

d) Montrons par récurrence forte que $b(s) = c(\left\lceil \frac{s}{2} \right\rceil)$

Base : c'est vrai pour s=0, car on a $b(0) = 0$ et $c(\frac{0}{2}) = 2^1 - 2 = 0$

Étape Inductive : Supposons la propriété vraie pour tout k<s et montrons que ça l'est au rang s :

$$b(s) = 2b(s-2) + 2 = 2(2^{\frac{s-2}{2}+1} - 2) + 2$$
$$b(s) = 2^{\frac{s-2}{2}+2} - 4 + 2 = 2^{\frac{s-2+4}{2}} - 2$$
$$b(s) = 2^{\frac{s+2}{2}} - 2$$
$$b(s) = 2^{\frac{s}{2}+1} - 2 = c(\left\lceil \frac{s}{2} \right\rceil)$$

Conclusion : $b(s) = c(\left\lceil \frac{s}{2} \right\rceil)$

e) D'après le résultat de la question précédente, on a :

$$b(s) = c(\frac{s}{2}) = 2^{\frac{s}{2}+1} - 2$$

et d'après les résultats de la question b) on a :

$$b(s) \leq a(s) \leq c(s) \Leftrightarrow 2^{\frac{s}{2}+1} - 2 \leq a(s) \leq 2^{s+1} - 2$$

ce qui veut dire que $a(s) \in O(2^s)$ et $a(s) \in \Omega(2^s)$.
On en déduit que a(s) $\in \Theta(2^s)$.
On conclut que la complexité temporelle de l'algorithme RechercheExhaustive pour un système de capacités avec k=2 et V= {1,2} est exponentielle et elle est en $\Theta(2^s)$.
Pour une instance particulière nous avons montré que l'algorithme est en $\Theta(2^s)$ donc on peut déduire que la complexité de l'algorithme RechercheExhaustive pour une instance quelconque est en $\Omega(2^s)$ pour $k \geq 2$ et $s > 0$.

2.2 Algorithme II : Programmation Dynamique

Pour résoudre notre problème d'optimisation, on recourt désormais à une autre méthode algorithmique dite de programmation dynamique qui, comme la méthode diviser pour régner, consiste à résoudre un problème en combinant des solutions de sous-problèmes autrement dit, on commence par résoudre les plus petits sous-problèmes et on mémorise (mémorisation) les solutions de ces derniers, on utilise ensuite ces solutions pour calculer les solutions de sous-problèmes de plus en plus grands, jusqu'à obtenir la solution du problème global.

Soit *m(S)* le nombre minimum de bocaux pour S dg de confiture et un tableau de capacités *V*. On définit un ensemble de sous problèmes dont les solutions nous aiderons à retrouver la solution du problème global, en l'occurrence *m(S)*: Étant donnés un entier s et un entier $i \in \{1,\dots,k\}$, on note *m(s,i)* le nombre minimum de bocaux nécessaires pour une quantité totale s en ne choisissant que des bocaux dans le système de capacités $V[1], V[2], \dots, V[i]$.

Question 3

a) Étant donné que *m(S)* est le nombre minimum de bocaux pour une quantité *S* dg de confiture en ne choisissant que des bocaux dans le système de capacités $V[1], V[2], \dots, V[k]$, Donc :

$$m(S) = m(S,k)$$

b) Montrons la relation de récurrence suivante pour tout $i \in \{1,\dots,k\}$:

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

$m(s,i)$ est le nombre minimum de bocaux nécessaires pour une quantité totale s en ne choisissant que des bocaux de capacités V[1],V[2], … , V[i]. si V[i]<=S, on peut choisir d'utiliser le bocal V[i], auquel cas le nombre minimum de bocaux à utiliser pour une quantité s sera égal à $m(s-V[i],i)+1$ ou on peut choisir de ne pas utiliser le bocal V[i], auquel cas le nombre minimum de bocaux à utiliser sera égal à $m(s,i-1)$. Comme on souhaite minimiser le nombre de bocaux, on choisit le minimum des résultats des deux cas, d'où la relation de récurrence ci-dessus.

Question 4

a)

D'après la relation de récurrence ci-dessus, on constate que la case du tableau destinée à recevoir la valeur *m(s,i)* ne peut être remplie qu'après les cases destinées à recevoir les valeurs *m(s,i-1)* et *m(s-V[i],i).*

Donc l'ordre de remplissage des cases du tableau M est un parcours ligne par ligne, de gauche à droite, ou colonne par colonne, de haut en bas, ce qui permet de respecter les dépendances dans la relation de récurrence.

b)

Algorithm 1 AlgoProgDyn (k: entier, V: tableau de k entiers, S: entier)

```
M : S × k Tableau rempli à 0
for i = 0 to S do
    for j = 0 to k do
        M[i][j] ← GetMatrix(j,V,i,M)
    end for
end for
return M[S][k]
```

Algorithm 1 GetMatrix (k: entier, V: tableau de k entiers, S: entier, M: Tableau $S \times k$)

```
if S = 0 then

    return 0
else if k = 0 || S < 0 then

    return ∞
else if M[S][k] ≠ 0 then

    return M[S][k]
else

    return min(GetMatrix(k - 1, V, S, M), GetMatrix(k, V, S - V[k], M))
end if
```

c) La complexité spatiale de cet algorithme est en $\mathrm{O}(kS)$ autrement dit la taille du tableau M. Sa complexité temporelle est également en $\mathrm{O}(kS)$, car il suffit d'un nombre constant d'opérations pour calculer la valeur de chaque case du tableau, de plus, dans cet algorithme on utilise une technique de mémoïsation qui nous permet d'éviter de calculer la même valeur plusieurs fois (En effet, ces valeurs sont stockées dans M) Donc les appels récursifs mémoïsés se feront en $\Theta(1)$.

Question 5

a) En modifiant l'algorithme précédent, il est possible de récupérer un tableau A indiquant le nombre de bocaux pris pour chacune des capacités. Pour cela, il suffit de modifier la fonction GetMatrix, toujours en se basant sur la relation de récurrence :

Algorithm 1 AlgoProgDynV2 (k: entier, V: tableau de k entiers, S: entier)

$M : S \times k$ Tableau rempli à nil
for $i = 0$ to S **do**
  for $j = 0$ to k **do**
    $M[i][j] \leftarrow GetMatrixV2(j,V,i,M)$
  end for
end for
return $M[S][k]$

Algorithm 1 GetMatrixV2 (k: entier, V: tableau de k entiers, S: entier, M: Tableau $S \times k$)

A : Tableau de k cases
if $S = 0$ **then**
  $A[1,...,k] \leftarrow [0...0]$
else if $k = 0$ **then**
  $A[1,...,k] \leftarrow []$
else if $S > 0$ and $k > 0$ **then**
  $left \leftarrow getMsk(k-1, V, S, M)$
  $up \leftarrow getMsk(k, V, S - V[k], M)$
  if $left < up$ **then**
    $A[1,...,k-1] \leftarrow M[S][k-1]$
    $A[k] \leftarrow 0$
  else
    $A[1, ..., k] \leftarrow M[S - V[k]][k]$
    $A[k] \leftarrow A[k] + 1$
  end if
end if
return A

Algorithm 1 getMsk (k: entier, V: tableau de k entiers, S: entier, M: Tableau $S \times k$)

if $S = 0$ **then**

  return 0
else if $k = 0 \,||\, S < 0$ **then**

  return ∞
else if $M[S][k] \neq nil$ **then**
  $A \leftarrow M[S][k]$

  return

$$\sum_{i=0}^{k} A[i]$$

end if

Complexité spatiale : Si on considère que le tableau M est représenté en mémoire par un tableau de tableaux, On aura donc un tableau de S+1 cases, dans chaque case, se trouve un tableau de k+1 cases.
$\forall i \in \{0,\ldots,S\}$, chaque case $M[i][j]$ contient un tableau de taille $j \in \{0,\ldots,k\}$.
Soit $T(S,k)$ la taille du tableau M.
On a bien : $T(S,k) = (S+1) + (S+1)(k+1) + (S+1)\frac{k(k+1)}{2} \in O(Sk^2)$

b) Remarquons qu'il est impossible d'afficher le tableau A (qui indique le nombre de bocaux utilisés pour chaque capacité) au fur et à mesure du remplissage de M car pendant la construction de ce dernier, on ne peut pas savoir quel « chemin » à travers ce tableau correspondra à la solution optimale. On doit donc remplir entièrement le tableau M avant de remplir le tableau A.
L'algorithme-retour est décrit ci-dessous :

Algorithm 1 AlgoProgDyn (k: entier, V: tableau de k entiers, S: entier)

```
M : S × k Tableau rempli à 0
for i = 0 to S do
    for j = 0 to k do
        M[i][j] ← GetMatrix(j,V,i,M)
    end for
end for
return M[S][k],Backward(k, V, S, M)
```

Algorithm 1 Backward (k: entier, V: tableau de k entiers, S: entier, M:tableau $k \times S$ d'entiers)

```
A :Tableau de taille k rempli à 0
q ← S
i ← k
while q ≠ 0 do
  if i > 0 and M[q][i] = M[q][i − 1] then
    i ← i-1
  else
    A[i − 1] ← A[i − 1] + 1
    q ← q - V[i − 1]
  end if
end while
return A
```

La complexité temporelle de cet algorithme est en $\mathrm{O}(kS)$. On calcule les valeurs de la matrice M en $\mathrm{O}(kS)$ et l'algorithme Backward est en $\mathrm{O}(S+k)$ car dans les pires cas, c'est à dire, les cas où on utilise que des bocaux de taille $V[1] = 1$, on effectuera S itérations (q sera décrémenté de 1 à chaque itération de la boucle) et k itérations pour mettre à jour le tableau A. La complexité spatiale reste inchangée étant donné que la complexité spatiale de l'algorithme Backward est en $\mathrm{O}(k)$ (La taille du tableau A).

c) La complexité que nous avons trouvée précédemment dépend de deux paramètres k et S. On sait qu'en théorie de la complexité, la complexité d'un algorithme est calculée en fonction de la longueur de l'entrée et non pas de sa valeur donc on ne peut pas conclure à un temps de calcul polynomial car en effet la taille de S est $\log_2 S$ et la taille de k est $\log_2 k$. Or, $S = 2^{\log_2 S}$ et $k = 2^{\log_2 k}$. Donc la complexité de l'algorithme est exponentielle par rapport à la taille des données d'entrées et on peut également dire qu'elle est pseudo-polynomiale.

2.3 Algorithme III : Cas particulier et algorithme glouton

Dans cette partie, on considère une nouvelle implémentation de l'algorithme via une approche gloutonne. Cette méthode consiste à effectuer une succession de choix qu'on ne remettra pas en cause dans les étapes suivantes. L'algorithme peut être implémenté récursivement et itérativement, nous avons fait le choix d'une implémentation itérative pour éviter les débordements de la pile de récursion.

Question 7

Algorithm 1 AlgoGlouton (k: entier, $V[0, ..., k-1]$: tableau de k entiers, S: entier)

```
nbBocaux ← 0
s ← S
L[1, ..., k] : Tableau contenant k "0"
while s ≠ 0 do
    i ← 0
    imax ← 0
    while V[i] <= s and i<k do
        if V[i] > V[imax] then
            imax ← i
        end if
        i ← i + 1
    end while
    s ← s - V[imax]
    L[imax] ← L[imax] + 1
    nbBocaux ← nbBocaux + 1
end while
return nbBocaux, L
```

Dans le pire cas, c'est à dire, quand $k = 1$, la première boucle "while" effectuera S tours de boucle étant donné qu'on diminuera s de 1 à chaque itération, la deuxième boucle "while" effectuera une seule itération à chaque fois. On a donc une complexité en $\mathrm{O}(S)$.

Question 8

Un système de capacité est dit glouton-compatible si l'algorithme Glouton renvoie bien la solution optimale pour ce système de capacités quelle que soit la quantité totale S.

Il existe cependant des systèmes de capacités qui ne sont pas glouton-compatibles, en voici deux exemples :

$V_1 = [1,100,101]$ et $S_1 = 200dg$

L'algorithme glouton retournera la solution $g = [99,0,1]$ or cette solution n'est pas optimale car pour une quantité de $200dg$ il suffit de prendre 2 bocaux de taille 100 ce qui nous donne une solution optimale $o = [0,2,0]$.

$V_2 = [1,20,30]$ et $S_2 = 40dg$

Pareil, l'algorithme glouton retournera une solution $g = [10,0,1]$ alors que la solution optimale est $o = [0,2,0]$.

Question 9

Considérons un système dit système Expo tel que les capacités ont pour valeurs $V[1] = 1, V[2] = d, V[3] = d^2, V[4] = d^3, \ldots, V[k] = d^{k-1}$ pour $d \geq 2$.

Montrons que ce système est glouton-compatible. Pour montrer l'optimalité de l'algorithme glouton pour ce système de capacités, nous allons procéder à une preuve classique dite « par échange ».

On considère une instance du problème, et pour cette instance soit $g = (g_1, g_2, \ldots, g_k)$ la solution produite par l'algorithme Glouton et soit $o = (o_1, o_2, \ldots, o_k)$ une solution optimale tel que $o \neq g$.

a) Montrons qu'il existe un plus grand indice $j \in \{1,\ldots,k\}$ tel que $o_j < g_j$:

Par construction de l'algorithme glouton et vu la définition de l'heuristique on a nécessairement $o_j < g_j$. En effet, l'algorithme glouton utilise autant de fois que possible le bocal de taille maximale et comme la solution o différente de g, la solution o fera un choix "non glouton", il existe donc forcément un type de bocal j que la solution gloutonne choisira autant de fois que possible mais ça ne sera pas le cas pour la solution optimale o, on a

donc $g_j \geq o_j + 1$ (on a $g_j = o_j + 1$ si l'algorithme glouton choisit le bocal j une fois de plus que l'algorithme qui renvoie la solution optimale).
Étant donné que $g_j \geq o_j + 1$ on a donc $g_j > o_j$ pour un plus grand indice $j \in \{1,\ldots,k\}$.
b) On a de plus :

$$\sum_{i=1}^{k} V[i]g_i = \sum_{i=1}^{k} V[i]o_i \qquad (1)$$

On sait qu'il n'existe pas de $t > j$ tel que $o_t > g_t$ car cela est contradictoire avec la construction de l'algorithme glouton. De plus, il n'existe pas de $t' > j$ tel que $o_{t'} < g_{t'}$ car cela contredit le résultat de la question a) : j est le plus grand indice tel que $o_j < g_j$. Donc :
$g_i = o_i$ pour i supérieur à j, on a donc :

$$\sum_{i=j+1}^{k} V[i]g_i = \sum_{i=j+1}^{k} V[i]o_i \qquad (2)$$

Par (1) et (2) on déduit que :

$$\sum_{i=1}^{j} V[i]g_i = \sum_{i=1}^{j} V[i]o_i$$

c) On a aussi :

$$\sum_{i=1}^{j-1} V[i]o_i + V[j]o_j = \sum_{i=1}^{j-1} V[i]g_i + V[j]g_j$$

Donc :

$$\sum_{i=1}^{j-1} V[i]o_i + V[j]o_j \geq V[j]g_j$$

$$\sum_{i=1}^{j-1} V[i]g_i \geq \quad V\lfloor j \rfloor g_j - V\lfloor j \rfloor o_j$$

$$\sum_{i=1}^{j-1} V[i]o_i \geq \quad V\lfloor j \rfloor (g_j - o_j)$$

et comme $o_j < g_j$ on a :

$$\sum_{i=1}^{j-1} V[i]o_i \geq \quad V\lfloor j \rfloor$$

avec $V\lfloor j \rfloor = d^{j-1}$

Interprétation « en français » : Cette propriété nous fait remarquer que la solution o fait un choix « non glouton » à une étape donné, autrement dit, o ne fait pas le choix de prendre le bocal de plus grande capacité malgré le fait que la quantité restante soit supérieure à la capacité de ce bocal. En trouvant une contradiction à cette propriété, on pourra montrer que l'algorithme glouton fait le bon choix lors de cette étape.

d) Nous avons :

$$\sum_{i=1}^{j-1} V[i] = \sum_{i=1}^{j-1} d^{i-1} = \sum_{i=1}^{j-2} d^i = \frac{d^{j-1} - 1}{d-1}$$

$$(d-1)\sum_{i=1}^{j-1} V[i] = \quad d^{j-1} - 1$$

comme $o_i \leq d - 1$ on a alors :

$$\sum_{i=1}^{j-1} V[i]o_i \leq \sum_{i=1}^{j-1} (d-1)V[i]$$

$$\sum_{i=1}^{j-1} V[i]o_i \leq (d-1)\sum_{i=1}^{j-1} V[i]$$

$$\sum_{i=1}^{j-1} V[i]o_i \leq d^{j-1} - 1$$

$$\sum_{i=1}^{j-1} V[i]o_i < d^{j-1}$$

or d'après les résultat de la question c) , on a :

$$\sum_{i=1}^{j-1} V[i]o_i \geq d^{j-1}$$

CONTRADICTION.

e)

D'après les résultats de la question précédente on déduit qu'il existe un $l,\ 1 \leq l \leq j-1$ tel que $o_l \geq d$

Soit la suite o' telle que $o'_l = o_l - d$ et $o'_{l+1} = o_{l+1} + 1$ et $o'_i = o_i$ pour tout $i \neq l+1\ et\ i \neq l+1$.

Pour que o' soit une solution du problème il faut que :

$$\sum_{i=1}^{k} V[i]o'_i = S$$

Puisque o est une solution du problème on sait que :

$$\sum_{i=1}^{k} V[i]o_i = S$$

Prouvons que :

$$\sum_{i=1}^{k} V[i]o_i = \sum_{i=1}^{k} V[i]o'_i$$

On sait que :

$$\sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o'_i + V[l]o'_l + V[l+1]o'_{l+1} + \sum_{i=l+2}^{k} V[i]o'_i$$

D'après l'énoncé, nous avons :

$$\sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o_i + V[l]o'_l + V[l+1]o'_{l+1} + \sum_{i=l+2}^{k} V[i]o_i$$

Toujours d'après l'énoncé :

$$\sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o_i + V\big[l\big](o_l - d) + V[l+1](o_{l+1} + 1) + \sum_{i=l+2}^{k} V[i]o_i$$

or $V\big[l\big] = d^{l-1}$ et $V\big[l+1\big] = d^l$

Donc :

$$\sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o_i + d^{l-1}(o_l - d) + d^l\,(o_{l+1} + 1) + \sum_{i=l+2}^{k} V[i]o_i$$

$$\Leftrightarrow \sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o_i + d^{l-1}o_l - d^l + d^l\,o_{l+1} + d^l + \sum_{i=l+2}^{k} V[i]o_i$$

$$\Leftrightarrow \sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{l-1} V[i]o_i + d^{l-1}o_l + d^l\,o_{l+1} + \sum_{i=l+2}^{k} V[i]o_i$$

Donc :

$$\sum_{i=1}^{k} V[i]o'_i = \sum_{i=1}^{k} V[i]o_i$$

On en déduit que o' est bien une solution du problème et elle est meilleure que o car en effet :

$$o_l + o_{l+1} - d + 1 < o_l + o_{l+1} \Leftrightarrow o_l + o_{l+1} - (d-1) < o_l + o_{l+1}$$

Et comme $d \geq 2$ on a $d > 1$ donc :

$$o'_l + o'_{l+1} < o_l + o_{l+1}$$

et étant donné que $o'_l = o_l$ pour $i \neq l$ et $i \neq l+1$

On en déduit que :

$$\sum_{i=1}^{k} o'_i < \sum_{i=1}^{k} o_i$$

Donc o' est une solution qui utilise moins de bocaux que la solution o : o n'est pas optimale.

On aboutit alors à une contradiction de l'hypothèse $o \neq g$.

Donc, on a nécessairement $o = g$, autrement dit, la solution retournée par l'algorithme glouton est bel et bien optimale pour le système Expo quelle que soit la valeur de S. On en déduit que le système Expo est glouton-compatible.

Question 10

Étant donné que $V[1] = 1$, tout système de capacités V avec $k = 2$ est composé d'un bocal $V[1] = 1$ et un bocal $V[2] = t > 1$ (car V est trié). On sait que l'algorithme Glouton, pour ce système de capacités, retournera pour toute quantité de confiture S une solution de la forme $g = \{a, b\}$, avec a le nombre de bocaux de taille 1 utilisés et b le nombre de bocaux de taille t utilisés, de façon évidente, la solution optimale consistera à prendre autant de fois que possible des bocaux de taille t, en effet, étant donné que $V[1] = 1$ et que 1 est un diviseur de tout entier naturel, donc en utilisant des bocaux de taille t jusqu'à ce que la quantité restante soit inférieur à t, on aboutit à un nombre de bocaux inférieur au nombre de bocaux retourné par une solution qui prendrait des bocaux de capacité 1 dg alors que la quantité restante de confiture est supérieur ou égale à t. Comme l'algorithme Glouton repose sur l'idée de prendre des bocaux de taille maximum autant de fois que possible, donc ce dernier retourne bien une solution optimale pour tout système de V avec $k = 2$, on en déduit donc que ce système est bien glouton-compatible.

Question 11

L'algorithme TestGloutonCompatible est composé de deux boucles imbriquées, on remarque que la première est bornée par $V[k]$ donc la première boucle est en $O(V\left[k\right])$ La deuxième itère de 1 à k donc elle est en $\Theta(k)$, à l'intérieur de cette boucle, on effectue un test dans lequel il y'a deux appels à AlgoGlouton, qui est en $\mathrm{O}(S)$, donc on effectue k fois 2 appels à AlgoGlouton, la complexité de la deuxième boucle « pour » est donc en $O(k*S)$. On en déduit que l'algorithme TestGloutonCompatible est en $O(V\left[k\right]*k*S)$.

Cependant, on ne peut pas dire que l'algorithme est polynomial, en effet, il est pseudo-polynomial car comme nous l'avons mentionné précédemment, la complexité d'un algorithme est calculée en fonction de la taille de codage de l'entrée et qui est logarithmique en la valeur de cette entrée. De plus, rien ne nous empêche d'avoir $V\left[k\right] \gg k$. Donc l'algorithme TestGloutonCompatible est pseudo-polynomial.

3 Mise en œuvre

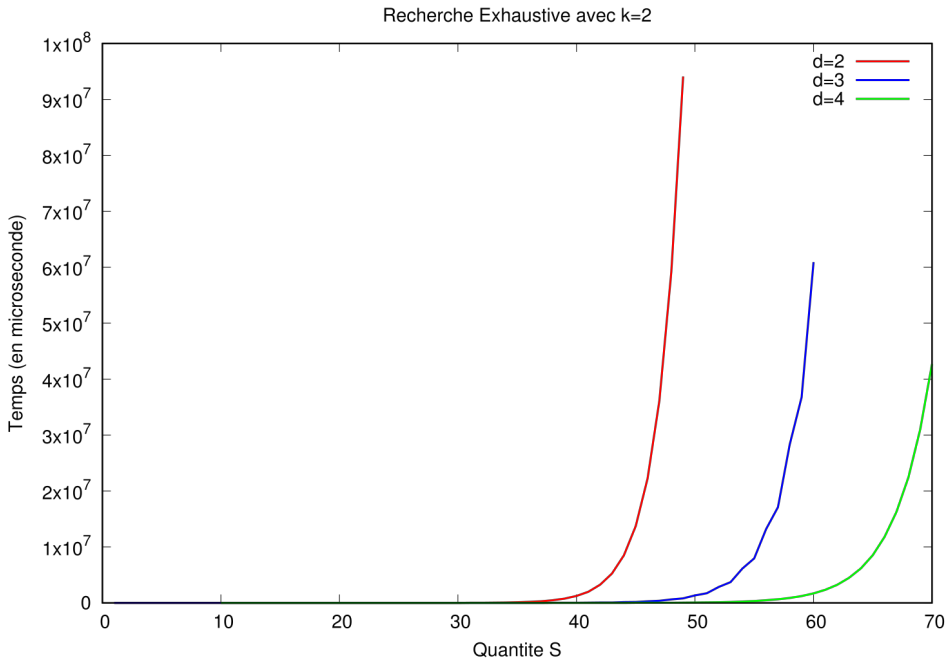
3.1 Implémentation et description du code

Les algorithmes ont été implémentés en Python et répartis en plusieurs fichiers *.py pour faciliter la lisibilité et la compréhension du code.
les fichiers RechExhaustive.py, AlgoDynamique.py et AlgoGlouton.py contiennent respectivement l'implémentation des algorithmes de recherche exhaustive, de programmation dynamique et l'algorithme glouton. Les fichiers Outils.py et TestFonctionnement.py contiennent le code qui permet de réaliser la partie 3.2 du projet. Les fichiers AnalyseComp.py et UtilisationAlgoGlouton.py contiennent les fonctions qui permettent de réaliser les parties 3.2 et 3.3. le fichier exec.py est un fichier de test qui contient en commentaire les différents scripts utilisés pour créer les fichiers de données qu'on a utilisés pour dessiner les graphes.

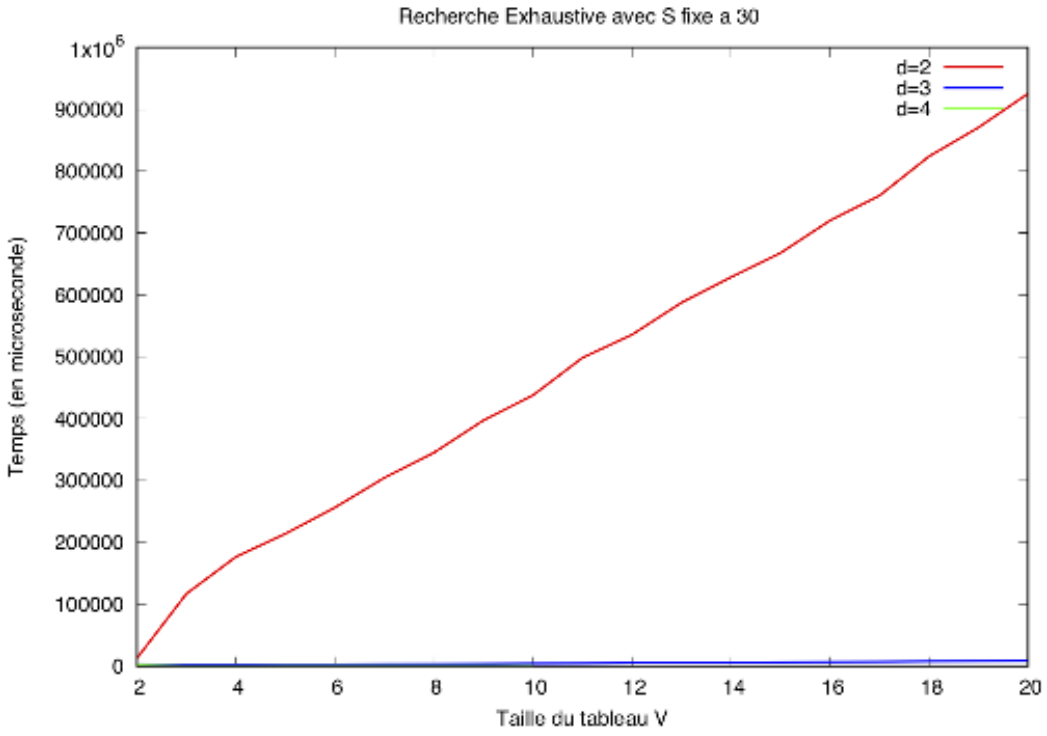
3.2 Analyse de complexité expérimentale

Question 12

1. Recherche Exhaustive



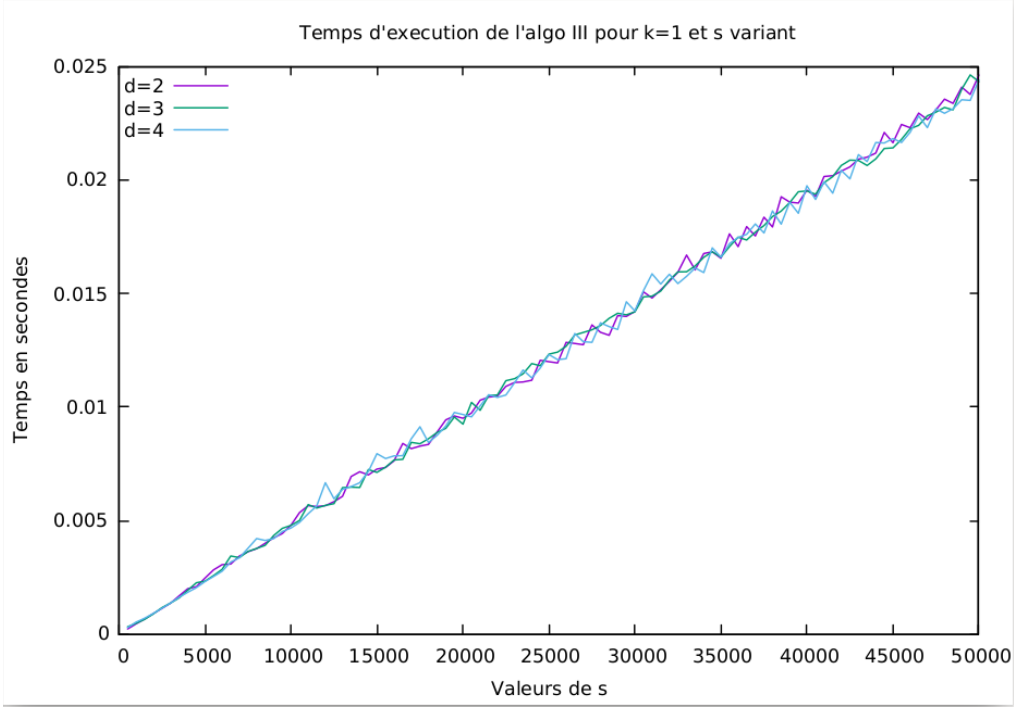
La figure ci-dessus représente le temps d'exécution de l'algorithme recherche exhaustive sur 3 systèmes de capacités de type Expo avec k=2, et S qui varie de 1 jusqu'à ce que le temps d'exécution atteigne 1 minute. La valeur de k a été choisie dans le but de différencier les différents systèmes de capacités sans que le programme prenne beaucoup de temps à s'exécuter. On remarque que la variation du temps d'exécution est bien exponentielle par rapport à S et que pour d=2, le programme commence à croître bien avant que pour les systèmes où d=3 et d=4, cela est dû au fait que la capacité maximum dans un système avec d=2 est inférieure à la capacité maximum dans un système avec d=3 ou d=4, et donc on aura forcément plus d'appels récursifs.
On peut en déduire que l'analyse expérimentale est en accord avec la complexité temporelle qu'on a trouvé dans la partie théorique, qui est de $\Omega(2^s)$.



Le graphique de la figure ci-dessus représente le temps d'exécution de l'algorithme recherche exhaustive pour des systèmes de capacités Expo avec d=2, d=3 et d=4, pour s=30 et k variant de 1 jusqu'à ce que le temps d'exécution atteigne 1minute, On remarque sur le graphique que la courbe qui représente le temps d'exécution de l'algorithme sur le système de capacité Expo avec d=2 augmente de façon monotone et plus rapide que les deux, qui elles, augmentent aussi de façon monotone mais plus lentement, cela est dû au fait que les valeurs des V[i] du système de capacités avec d=2 sont plus petites que les autres, ce qui fait que le nombre d'appels récursifs faits par l'algorithme de recherche exhaustive quand celui-ci est exécuté avec ce système de capacités est beaucoup plus grand que lorsqu'il est exécuté sur les deux autres, car dans l'algorithme de recherche exhaustive il y'a une boucle pour i allant de 1 à k, qui fait des appels récursifs avec s-V[i] et plus les valeurs de V[i] sont petites, plus on aura des valeurs s-V[i] qui seront

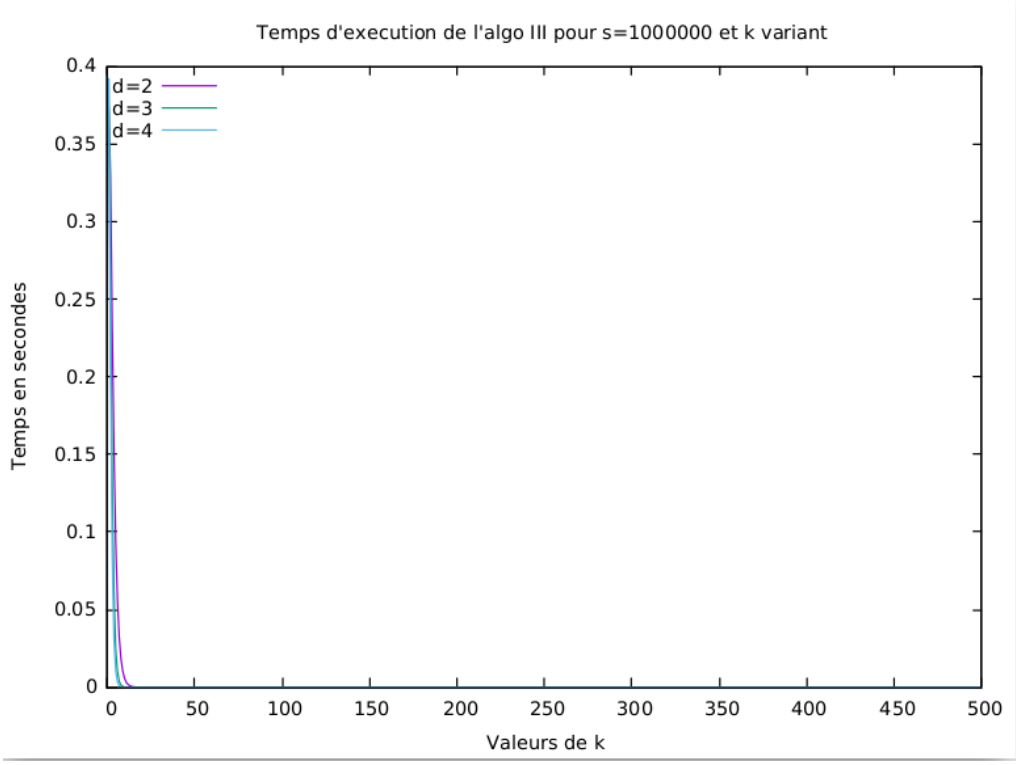
supérieurs ou égales à 0 et qui donc, généreront plus d'appels récursifs, par contre quand les valeurs des V[i] sont plus grandes (ce qui est le cas quand d=3 et d=4) on a plus de chances d'avoir des appels récursifs avec des valeurs s-V[i] qui seront négatives ou nulles et qui donc ne généreront pas autant d'appels récursifs, ce qui explique la différence entre les courbes qu'on voit sur le graphique. Étant donné que s est fixé et que k varie, on déduit que le temps d'exécution expérimentale est en accord avec la complexité temporelle qu'on a trouvé dans la partie théorique qui est de $\Omega(2^s)$.

2. Algorithme Glouton



Sur la figure ci-dessus, nous avons la courbe qui représente le temps d'exécution de l'algorithme glouton en fonction de la valeur de s (la

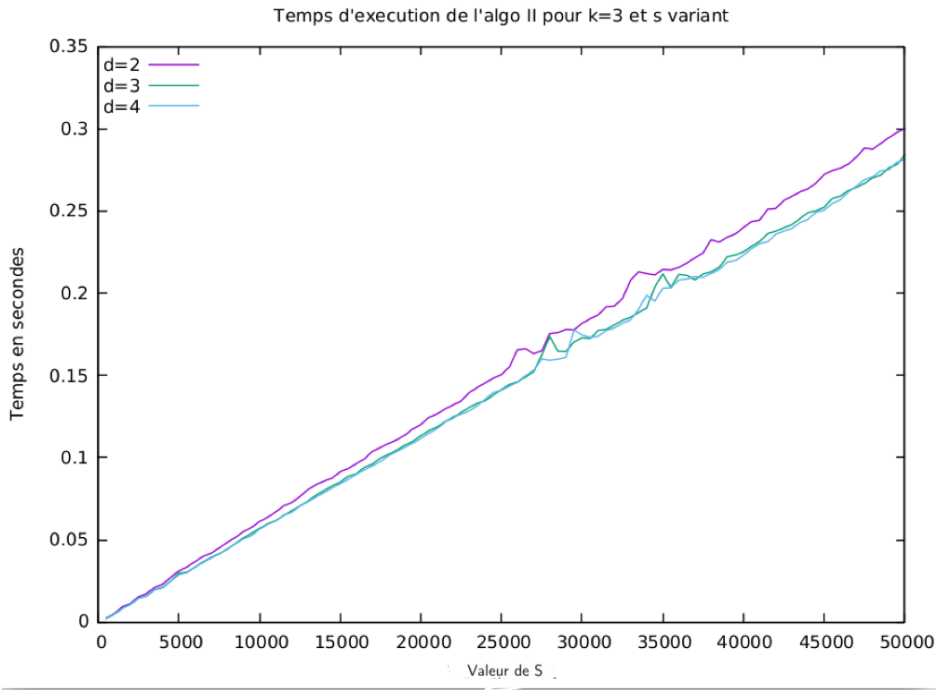
quantité de confiture), sur 3 systèmes de capacités Expo (décrit dans la question 9) pour une valeur k (nombre de bocaux) fixée à 1, la valeur de k a été choisie pour avoir le temps d'exécution dans le pire cas, et les valeurs de s varient entre 500 et 50000 d'un pas de 500. Le taux de variation et la valeur initiale de s ont était choisis de sorte qu'ils nous permettent de voir l'effet de la variation de s sur le temps d'exécution du programme sans que celui-ci prenne trop de temps à s'exécuter et sans qu'il nous donne, non plus, des valeurs non significatives. On remarque sur le graphe que les temps d'exécution de l'algorithme glouton pour les 3 systèmes de capacités sont au maximum quand s est à sa plus grande valeur, et ils sont au minimum quand s est à sa plus petite valeur, on remarque également que le temps d'exécution augmente de façon quasi-monotone avec s. Les pics et les creux qui apparaissent sur le graphique sont dûs à des facteurs liés à l'exécution du programme et apparaissent peu importe les valeurs de s et de k choisies, et pas sur les mêmes endroits de la courbe quand on exécute le programme plusieurs fois, ce qui nous permet de supposer qu'ils sont dûs au processeur, qui probablement, exécute autre chose en arrière plan et étant donné que les temps d'exécution ici sont négligeables, le processeur peut éventuellement affecter ces derniers.
L'augmentation du temps d'exécution de façon monotone avec s est bel et bien compatible avec ce qu'on a trouvé dans la partie théorique, qui est que l'algorithme glouton à une complexité en O(s).



La figure ci-dessus représente le temps d'exécution de l'algorithme glouton en fonction de k sur 3 systèmes de capacités Expo pour une valeur de s fixée à 1000000 et des valeurs de k qui varient de 1 à 500 et qui augmentent d'un pas de 1. La valeur de s a été choisie de sorte qu'elle nous permette de voir de quelle manière la variation de k affecte le temps d'exécution, si on avait choisi une valeur plus petite, on aurait eu du mal à distinguer le pire cas ($k = 1$) des autres cas (k > 1). Les valeurs de k ont été choisies pour la même raison, c'est à dire, dans le but de voir d'une part, comment le temps d'exécution est affecté par des valeurs de k variantes et d'une autre part, comment le choix de k peut nous donner un pire cas pour cet algorithme. On remarque que la valeur maximale du temps d'exécution de l'algorithme sur les 3 systèmes de capacités est atteinte quand k=1, ce qui confirme l'analyse théorique que nous avons effectuée. La valeur minimale est atteinte entre k=5 et k=15. On peut remarquer aussi que la courbe pour $d = 4$ décroît plus vite que les courbes pour $d = 3$ et $d = 2$, cela est dû au

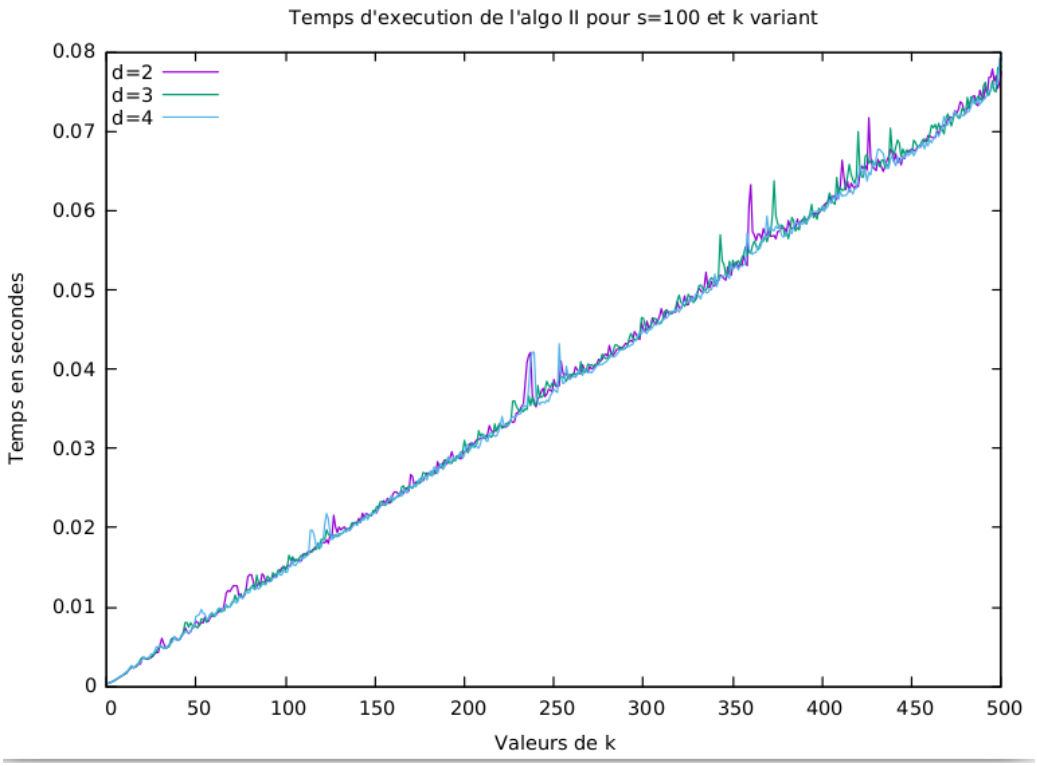
fait que plus d est grand, plus la valeur $V[k]<=S$ sera grande et une fois ce bocal de taille $V[k]$ utilisé, il est évident que la quantité S sera diminuée considérablement et donc l'algorithme prendra moins de temps pour la quantité restante. Cela explique donc la raison pour laquelle nous avons un temps d'exécution négligeable à partir de $k >= i$ avec $V[i]$ la capacité maximum inférieure ou égale à S.
L'analyse expérimentale de cet algorithme correspond bien à l'analyse théorique, où la complexité était en O(S) et le pire cas était atteint pour $k = 1$.

3. Programmation Dynamique



La figure ci-dessus représente le temps d'exécution de l'algorithme de programmation dynamique sur des systèmes de capacités Expo avec d=2, d=3 et d=4, avec S qui varie de 500 à 50000 avec un pas de 500. La valeur de k a été choisie pour nous permettre de voir la différence

en temps d'exécution avec l'algorithme glouton. Les valeurs de s ont été choisies de sorte qu'on puisse voir comment celles-ci affectent le temps de d'exécution du programme. On Remarque que le programme qui s'exécute sur un système de capacité Expo avec d=2 prend plus de temps que pour les deux autres car les valeurs de $V[i]$ avec i entre 1 et k pour ce système sont significativement plus petite que les deux autres. On remarque que le temps d'exécution augmente de façon presque monotone avec l'augmentation de s, les pics et creux qui apparaissent sur le graphe sont probablement dûs aux raisons qu'on a cité précédemment.
On peut dire que l'analyse expérimentale correspond à l'analyse théorique de cet algorithme, la complexité est bel et bien en O(Sk).



La figure ci-dessus représente le temps d'exécution de l'algorithme de programmation dynamique sur des systèmes de capacités Expo avec d allant de 2 à 4, avec k qui varie entre 1 et 500avec un pas de 1. La

valeur de s a été choisie de sorte qu'elle soit significatif dans le temps d'exécution.
On remarque que le temps d'exécution augmente de façon presque monotone avec l'augmentation de k, les pics et creux qui apparaissent sur le graphe sont probablement dûs aux mêmes raisons qu'on a cité précédemment. On peut conclure que l'analyse expérimentale correspond à l'analyse théorique de cet algorithme, et donc la complexité est bien en O(Sk).

3.3 Utilisation de l'algorithme glouton

Dans cette partie, on souhaite tester expérimentalement plusieurs aspects dont la fréquence d'apparition de systèmes de capacités glouton-compatibles et également évaluer pour un système non glouton-compatible l'écart relatif entre la valeur de la solution fournie par l'algorithme de programmation dynamique et la valeur de la solution fournie par l'algorithme glouton. Pour cela, nous avons généré plusieurs .txt contenant des statistiques pour un nombre de systèmes, un p_{max} et un f donnés.

Question 13
Avec $p_{max} = 100$ et en variant le nombre de systèmes de capacités générés, nous avons obtenu les résultats ci-dessous :

Nombre de sytèmes de capacités générés	Pourcentage de systèmes glouton-compatibles
50	4 %
100	3 %
200	0.5 %
400	2 %
800	2.25 %

1000	2.1 %
1500	2.53 %
2000	1.45 %
4000	1.9 %
10000	1.95 %
50000	1.73 %
100000	1.81 %

On remarque que la proportion de systèmes glouton-compatibles est petite et donc la fréquence d'apparition des systèmes glouton-compatibles est extrêmement faible. Les systèmes gloutons compatibles sont en fait des systèmes canoniques. Comme nos systèmes de capacités sont tirés aléatoirement, ils n'ont aucune raison d'être canonique.

Question 14

Sur un échantillon de 100 systèmes de capacités que l'on suppose représentatif, et pour chaque système détecté comme étant non glouton-compatible, nous avons calculé le pire écart et l'écart moyen entre le résultat de l'algorithme Glouton et celui de l'algorithme de programmation dynamique pour toutes les valeurs de S entre p_{max} et f^*p_{max}. Nous avons obtenu les statistiques ci-dessous:

Fichier	p_{max}	f	*pire écart sur l'ensemble des systèmes*	*écart minimum sur l'ensemble des systèmes*	*Nombre de systèmes glouton-compatibles*
GloutonCompatible_50_100_2	50	2	28	2	3
GloutonCompatible_50_100_6	50	6	38	2	22
GloutonCompatible_100_100_4	100	4	79	2	17
GloutonCompatible_100_100_6	100	6	80	3	21
GloutonCompatible_100_200_2	100	8	72	3	14

On peut constater que l'écart entre les résultats des deux algorithmes II et III est non négligeable, de plus, les systèmes glouton-compatibles sont de faible fréquence (comme on l'a pu le constater dans la Question 13).

Ces statistiques nous permettent de conclure sur la faisabilité de l'algorithme glouton en pratique, en effet, ce dernier est clairement de complexité moindre que l'algorithme de programmation dynamique et il est également plus simple à implémenter. Cependant, combien de temps d'exécution gagne-t-on pour une perte donnée en optimalité? Dans le cadre de ce problème, il serait plus judicieux de favoriser l'optimalité au temps d'exécution, et donc préférer la technique de programmation dynamique à l'approche gloutonne étant donné, que les systèmes de capacités sont rarement canoniques.

Il est certes possible de tester si un système est glouton-compatible ou non en temps polynomial et, en cas de réponse positive, on fait appel à l'algorithme glouton, sinon, on préférera l'algorithme de programmation dynamique, cependant, comme on l'a pu le remarquer dans la Question 13, les systèmes glouton-compatibles sont extrêmement rares dans le cadre de ce problème donc on n'y gagne pas énormément en temps d'exécution.

Conclusion

Afin de répondre à la problématique de la mise en bocal d'une quantité donnée de confiture, nous avons étudié plusieurs algorithmes dont l'algorithme glouton qui est très efficace mais optimal uniquement pour des systèmes canoniques (glouton-compatibles) et l'algorithme de programmation dynamique qui est toujours optimal, mais moins rapide que l'algorithme glouton. L'étude expérimentale nous a permis de conclure sur la faisabilité de l'algorithme glouton, la fréquence d'apparition des systèmes canoniques est quasiment négligeable, donc, dans le cadre de ce problème, il serait plus judicieux d'avoir recours à l'algorithme de programmation dynamique plutôt qu'à l'algorithme glouton et en contrepartie, compromettre le temps d'exécution. Il existe cependant des problèmes qui reposent sur des systèmes qui sont toujours glouton-compatibles comme les systèmes monétaires.