Mini-Projet d'Algorithmique

Confitures

3i003

Licence d'informatique

Sorbonne Université

Année universitaire 2018-2019

Réalisé par :

Merrouche Aymen

Sidhom Imad

Table des matières:

1 Introduction:

Pour un problème algorithmique donné il existe souvent plusieurs algorithmes permettant de le résoudre mais parmi toutes ces solutions possibles il y en a certaines qui sont plus efficaces que d'autres en temps et/ou en espace, selon le cas. Afin de comparer l'efficacité de plusieurs algorithmes qui résolvent tous un même problème, on se propose de traiter algorithmiquement le problème suivant :

« Étant donné une quantité de S décigrammes de confiture, un ensemble de bocaux de plusieurs capacités que l'on classe en k types de bocaux différents : chaque type de bocal correspondant à une capacité distincte, quel est le nombre minimum de bocaux tels que la somme de leur capacité est exactement égale à S. »

Nous allons-nous intéresser dans ce qui va suivre à trois différents algorithmes pour traiter ce problème, en les étudiant d'abord théoriquement, en analysant les valeurs obtenues expérimentalement en fonction des résultats théoriques et enfin en comparant les performances de ces trois algorithmes.

2 Partie théorique:

2.1 Algorithme I : Recherche exhaustive

Réponse à la question 1 :

Validité : (par récurrence forte sur s)

Montrons par récurrence forte sur s que RechercheExhaustive(k,V,s) retourne le nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à s (valide) :

Cas de base :

- Si $s < 0$: l'algorithme retourne $+\infty$,car il est impossible de déterminer ce nombre pour une quantité de confiture négative donc le résultat retourné dans ce cas est valide (il retourne $+\infty$ pour les besoins de l'algorithme).
- Si $s = 0$: l'algorithme retourne 0, pour une quantité nulle de confiture, aucun bocal n'est nécessaire donc le résultat retourné dans ce cas est valide.

Induction :

Supposons que H1 : « l'algorithme est valide pour toutes les quantités de 0 à s », montrons qu'il est valide pour $ś = s + 1$ (i.e. RechercheExhaustive(k,V,$ś$) retourne le nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à $ś$)

-Puisque $ś > 0$ ($s \geq 0$ et $ś = s + 1$)(s et $ś$ sont des entiers) :

- Au début NbCont vaut $ś$,ce qui correspond à $ś$ bocal de capacité $V[1] = 1$ (cas où on utilise le plus de bocaux)
- On itère sur i allant de 1 au nombre de capacités k, à chaque itération (correspondant à une capacité $V[i]$) on appel récursivement RechercheExhaustive(k,V,$ś - V[i]$)
  Or : $ś - V[i] = s + 1 - V[i]$… (1)

$$V[i] \geq 1 \text{ (D'après l'énoncé)}$$
$$\Leftrightarrow -V[i] \leq -1$$
$$\Leftrightarrow s + 1 - V[i] \leq s + 1 - 1$$
$$\Leftrightarrow s + 1 - V[i] \leq s$$

$\overset{(1)}{\Leftrightarrow} \acute{s} - V[i] \leq s$

Puisque $\acute{s} - V[i] \leq s$ et d'après H1 à chaque itération, x contient le nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à $\acute{s} - V[i]$.
Conclusion 1 : pour toutes les itérations de 1 à k, RechercheExhaustive(k,V,$\acute{s} - V[i]$) est valide.

-Montrons par récurrence sur i la propriété P_i : « A l'issue de l'itération i ,$NbCont_i$ contient la meilleure solution parmi (La solution optimale qui utilise au moins un bocal de capacité $V[1]$, La solution optimale qui utilise au moins un bocal de capacité $V[2]$,…, La solution optimale qui utilise au moins un bocal de capacité $V[i-1]$, La solution optimale qui utilise au moins un bocal de capacité $V[i]$) »

Notons la meilleure solution parmi (La solution optimale qui utilise au moins un bocal de capacité $V[1]$, La solution optimale qui utilise au moins un bocal de capacité $V[2]$,…, La solution optimale qui utilise au moins un bocal de capacité $V[i-1]$, La solution optimale qui utilise au moins un bocal de capacité $V[i]$) » m_i.

Avant la boucle, NbCont est initialisé au pire cas, cas où on utilise uniquement des bocaux de capacité $V[1]$ ($NbCont_0 := s$)

Cas de base (i=1) :

- x Contient RechercheExaustive(k,V,$s - V[1]$), ce qui correspond d'après conclusion1 au nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à $s - v[1]$.
- On compare $x + 1$ (+1 correspond à la quantité $V[1]$ qui a été retranchée dans l'appel de RechercheExaustive(k,V,$s - V[1]$) et donc à un bocal de capacité $V[1]$ c'est-à-dire on force l'utilisation d'un bocal de capacité $V[1]$, d'où $x + 1$ représente la solution optimale qui utilise au moins un bocal de capacité $V[1]$) à $NbCont_0$,si il est strictement inferieur (i.e. la solution optimale qui utilise au moins un bocal de capacité $V[1]$ est meilleure que le pire cas) $NbCont_1 := x + 1$, sinon $NbCont_1 := NbCont_0$, au final on a bien que à l'issue de l'itération 1 $NbCont_1$ contient la meilleure solution parmi (La solution optimale qui utilise au moins un bocal de capacité $V[1]$) donc ok.

Induction :

Supposons que H1 : P_i et montrons que P_{i+1}.

- x Contient RechercheExaustive(k,V,$s - V[i+1]$),ce qui correspond d'après conclusion1 au nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à $s - v[i+1]$.
- On compare $x + 1$ (+1 correspond à la quantité $V[i+1]$ qui a été retranchée dans l'appel de RechercheExaustive(k,V,$s - V[i+1]$) et donc à un bocal de capacité $V[i+1]$ c'est-à-dire on force l'utilisation d'un bocal de capacité $V[i+1]$, d'où $x + 1$ représente la solution optimale qui utilise au moins un bocal de capacité $V[\mathrm{i}+1]$) à $NbCont_i$ qui représente d'après H1 m_i,si il est strictement inferieur (i.e. la solution optimale qui utilise au moins un bocal de capacité $V[i+1]$ est meilleure que m_i) $NbCont_{i+1} := x + 1$, sinon $NbCont_{i+1} := NbCont_i$, au final on a bien que à l'issue de l'itération i+1 $NbCont_{i+1}$ contient m_{i+1} donc ok.

Conclusion 2 :

Pour tout $i\epsilon\{1, \dots, k\}$, A l'issue de l'itération i ,$NbCont_i$ contient la meilleure solution parmi (La solution optimale qui utilise au moins un bocal de capacité $V[1]$, La solution optimale qui utilise au moins un bocal de capacité $V[2]$,…, La solution optimale qui utilise au moins un bocal de capacité $V[i-1]$, La solution optimale qui utilise au moins un bocal de capacité $V[i]$) .

En particulier à l'issue de l'itération k, $NbCont_k$ contient la meilleure solution parmi (La solution optimale qui utilise au moins un bocal de capacité $V[1]$, La solution optimale qui utilise au moins un bocal de capacité $V[2]$,…, La solution optimale qui utilise au moins un bocal de capacité $V[k-1]$, La solution optimale qui utilise au moins un bocal de capacité $V[k]$) , ce qui correspond exactement à la solution optimale.

Plus en détails : La solution optimale qui utilise au moins un bocal de capacité $V[i]$ représente la meilleure solution parmi toutes les solutions qui utilisent au moins un bocal de capacité $V[i]$ donc $NbCont_k$ contient la meilleure solution parmi (Les solution qui utilise au moins un bocal de capacité $V[1]$, Les solution qui utilise au moins un bocal de capacité $V[2]$,…, Les solution qui utilise au moins un bocal de capacité $V[k-1]$, Les solution qui utilise au moins un bocal de capacité $V[k]$) donc $NbCont_k$ contient la meilleure solution parmi (toutes les solutions possibles) donc $NbCont_k$ contient la solution optimale.

Terminaison : (par récurrence forte sur s)

Montrons par récurrence forte sur s que RechercheExaustive(k,V,s) se termine :

Cas de base :

- Si $s < 0$: l'algorithme retourne $+\infty$, et donc se termine
- Si $s = 0$: l'algorithme retourne 0, et donc se termine.

Induction :

Supposons que H1 : « l'algorithme se termine pour toutes les quantités de 0 à s », montrons qu'il se termine pour $\acute{s} = s + 1$

-Puisque $\acute{s} > 0$ ($s \geq 0$ et $\acute{s} = s + 1$)(s et $\acute{s}$ sont des entiers) :

- La première instruction est une affectation qui se termine
- On itère sur i allant de 1 au nombre de capacités k,la boucle pour se termine dès que $i > k$, i s'incrémente de 1 à chaque itération (d'après la construction de la structure de contrôle pour) et à part cette incrémentation il ne subit aucune autre modification, donc i deviendra $> k$ après exactement k itérations.
  Conclusion 1 : il y'a exactement k tour de boucle (nombre fini).
  -montrons que chaque itération se termine :
  - À chaque itération (correspondant à une capacité $V[i]$) on appel récursivement RechercheExaustive(k,V,$\acute{s} - V[i]$)

$$\text{Or :} \quad \acute{s} - V[i] = s + 1 - V[i] \ldots (1)$$
$$V[i] \geq 1 \text{ (D'après l'énoncé)}$$
$$\Leftrightarrow -V[i] \leq -1$$
$$\Leftrightarrow s + 1 - V[i] \leq s + 1 - 1$$
$$\Leftrightarrow s + 1 - V[i] \leq s$$
$$\overset{(1)}{\Leftrightarrow} \acute{s} - V[i] \leq s$$

  Puisque $\acute{s} - V[i] \leq s$ et d'après H1 RechercheExaustive(k,V,$\acute{s} - V[i]$) se termine.
  - L'autre instruction est une structure de contrôle si qui contient une affectation et donc se termine.

  Conclusion 2 : pour i allant de 1 au nombre de capacités k, chaque itération se termine.

On conclut d'après conclusion1 et conclusion2 que la boucle pour se termine après exactement k itérations.

- L'algorithme retourne NbCont et se termine.

Conclusion : Pour tout $s \in N$,RechercheExaustive(k,V,s) se termine.

Réponse à la question 2 :

Seuls le bocaux de capacités 1dg et 2dg sont disponibles, $a(s)$ représente le nombre d'appels récursifs effectues par RechercheExaustive(2,[1,2],s)

a) Exprimons $a(s)$ sous forme d'une suite récursive :

Par déroulement de l'algorithme :

$Pour\ s = 0\ , a(0) = 0$

$Pour\ s = 1\ , a(1) = 2$

$Pour\ s = 2\ , a(2) = 4$

$Pour\ s = 3\ , a(3) = 8$

$Pour\ s = 4\ , a(4) = 14$

Alors on a :

- $Pour\ s = 0$: l'algorithme retourne 0 sans aucun appel récursif donc $a(0) = 0$.
- $Pour\ s < 0$: l'algorithme retourne $+\infty$ sans aucun appel récursif donc $a(+\infty) = 0$
- $Pour\ s > 0$: on a que $k = 2$,d'après l'algorithme on va avoir deux itérations, durant la première itération on a un appel récursif de RechercheExaustive(2,[1,2],$s-1$) et durant la deuxième on a un appel récursif de RechercheExaustive(2,[1,2],$s-2$) et donc $a(s) = 2 + a(s-1) + a(s-2)$.

On en conclut que :

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

b) Montrons par récurrence forte sur s que $b(s) \leq a(s) \leq c(s)$ $pour\ tout\ entier\ s \geq 0$:

Cas de base :

- $s = 0$

$$\begin{cases} a(0) = 0 \\ b(0) = 0 \\ c(0) = 0 \end{cases} \Rightarrow \quad b(0) \leq a(0) \leq c(0)$$

- $s = 1$

$$\begin{cases} a(1) = 2 + a(0) + a(-1) = 2 + 0 + 0 = 2 \\ b(1) = 2 \\ c(1) = 2.c(0) + 2 = 2.0 + 2 = 2 \end{cases} \Rightarrow \quad b(1) \leq a(1) \leq c(1)$$

Induction :

<u>a)-Pour $b(s) \leq a(s)$: $(s > 1)$</u>

Supposons que H1 : « $b(k) \leq a(k)$ pour k(entier) allant de 2 à s-1 » et montrons que $b(s) \leq a(s)$

D'après H1 : $b(s-2) \leq a(s-2)$ … (1)

-Montrons que : $a(s-1) \geq a(s-2)$

$$s > 1 \Leftrightarrow s - 1 > 0 \text{ Donc } a(s-1) = 2 + a(s-2) + a(s-3)$$

$$\text{Or } a(s-3) + 2 > 0$$

$$\text{Donc } a(s-1) \geq a(s-2) \ldots (2)$$

D'après (1) et (2) : $a(s-1) \geq a(s-2) \geq b(s-2)$

$$\Rightarrow a(s-1) \geq b(s-2)$$

On somme : $a(s-2) + a(s-1) \geq 2.b(s-2)$

$$\Leftrightarrow a(s-2) + a(s-1) + 2 \geq 2.b(s-2) + 2$$

$$\Leftrightarrow a(s) \geq b(s) \quad \text{CQFD}$$

b)-Pour $a(s) \leq c(s)$: $(s > 1)$

Supposons que H2 : « $a(k) \leq c(k)$ pour k(entier)allant de 2 à s-1 » et montrons que $a(s) \leq c(s)$

D'après H1 : $a(s-1) \leq c(s-1)$ … (1)

-Montrons que : $a(s-1) \geq a(s-2)$

$$s > 1 \Leftrightarrow s - 1 > 0 \text{ Donc } a(s-1) = 2 + a(s-2) + a(s-3)$$

$$\text{Or } a(s-3) + 2 > 0$$

$$\text{Donc } a(s-1) \geq a(s-2) \ldots (2)$$

D'après (1) et (2) : $a(s-2) \leq a(s-1) \leq c(s-1)$

$$\Rightarrow a(s-2) \leq c(s-1)$$

On somme : $a(s-2) + a(s-1) \leq 2.c(s-1)$

$$\Leftrightarrow a(s-2) + a(s-1) + 2 \leq 2.c(s-1) + 2$$

$$\Leftrightarrow a(s) \leq b(s) \quad \text{CQFD}$$

Donc de a) et b) on obtient que : $b(s) \leq a(s) \leq c(s)$

Conclusion :

Pour tout entier $s \geq 0$,on a que $b(s) \leq a(s) \leq c(s)$.

c) Déterminons le terme général de la suite c(s) :

En appliquant la définition récursive de la suite c(s) on trouve :

$Pour\ s = 0\ , c(0) = 0$

$Pour\ s = 1\ , c(1) = 2.c(0) + 2 = 2$

$Pour\ s = 2\ , c(2) = 2.c(1) + 2 = 2.2 + 2 = 6$

$Pour\ s = 3\ , c(3) = 2.c(2) + 2 = 2.6 + 2 = 14$

$Pour\ s = 4\ , c(4) = 2.c(3) + 2 = 2.14 + 2 = 30$

$Pour\ s = 5\ , c(5) = 2.c(4) + 2 = 2.30 + 2 = 62$

Montrons que $c(s) = 2^{s+1} - 2 = 2.(2^s - 1)$:

-Par récurrence sur s :

Cas de base ($s = 0$) :

D'après la définition récursive de la suite c(s), $c(0) = 0 = 2^{0+1} - 2 = 2 - 2 = 0$ $donc\ ok$

Induction :

Supposons que H1 : « $c(s) = 2^{s+1} - 2$ » et montrons que $c(s+1) = 2^{s+2} - 2$

Puisque $s + 1 \geq 1$ ($car\ s \geq 0\ et\ s\ entier$) :

$c(s+1) = 2.c(s) + 2$ Par définition de c(s)

$= 2.(2^{s+1} - 2) + 2$ D'après H1

$= 2^{s+2} - 4 + 2 = 2^{s+2} - 2$ CQFD

Conclusion :

Pour tout $s \geq 0$,$c(s) = 2^{s+1} - 2$

d) Montrons par récurrence sur s que $b(s) = c(\lceil \frac{s}{2} \rceil)$:

Si $s = 2.k$ (cas des s pairs) ($k \geq 0\ et\ k\ entier$) :

Montrons par récurrence sur s que $b(s) = c\left(\left\lceil \frac{2.k}{2} \right\rceil\right) = c(\lceil k \rceil) = c(k) = 2^{k+1} - 2$:

Cas de base (s=0) :

$b(0) = 0$ Par définition récursive de la suite b(s).

$c\left(\left\lceil \frac{0}{2} \right\rceil\right) = c(0) = 2^{0+1} - 2 = 2 - 2 = 0 = b(0)$ Donc ok

Induction ($s > 0$) :

$$c\left(\left\lceil \frac{s-2}{2} \right\rceil\right) = c\left(\left\lceil \frac{2.k-2}{2} \right\rceil\right) = c\left(\left\lceil \frac{2.(k-1)}{2} \right\rceil\right) = c(k-1) = 2^k - 2$$

Supposant que H1 : « $b(s-2) = c\left(\left\lceil \frac{s-2}{2} \right\rceil\right) = 2^k - 2$ » (le pas est égal à deux car nous nous intéressons ici qu'au nombre entiers pairs uniquement) et montrons que $b(s) = c\left(\left\lceil \frac{2.k}{2} \right\rceil\right) = c(\lceil k \rceil) = c(k) = 2^{k+1} - 2$.

$b(s) = 2.b(s-2) + 2$ D'après la définition récursive de la suite b(s) ($s > 0$ et pair $\Rightarrow s \geq 2$).

$= 2.c\left(\left\lceil \frac{s-2}{2} \right\rceil\right) + 2 = 2.(2^k - 2) + 2$ D'après H1

$= 2^{k+1} - 4 + 2 = 2^{k+1} - 2$ CQFD

Conclusion 1 :

Pour tout $s \geq 0$ pair, $b(s) = c(\lceil \frac{s}{2} \rceil)$

Si $s = 2.k + 1$ (cas des s impairs) ($k \geq 0\ et\ k\ entier$) :

Montrons par récurrence sur s que $b(s) = c\left(\left\lceil\frac{2.k+1}{2}\right\rceil\right) = c\left(\left\lceil k+\frac{1}{2}\right\rceil\right) = c(k+1) = 2^{k+2} - 2$ (le plus petit entier supérieur ou égale à $k+\frac{1}{2}$ est $k+1$ d'où $\left\lceil k+\frac{1}{2}\right\rceil = k+1$) :

Cas de base (s=1) :

$b(1) = 2$ Par définition récursive de la suite b(s).

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

Induction :

$c\left(\left\lceil\frac{s-2}{2}\right\rceil\right) = c\left(\left\lceil\frac{2.k+1-2}{2}\right\rceil\right) = c\left(\left\lceil\frac{2.k-1}{2}\right\rceil\right) = c\left(\left\lceil k-\frac{1}{2}\right\rceil\right) = c(k) = 2^{k+1} - 2$ (Le plus petit entier supérieur ou égale à $k-\frac{1}{2}$ est k d'où $\left\lceil k-\frac{1}{2}\right\rceil = k$)

Supposant que H1 : « $b(s-2) = c\left(\left\lceil\frac{s-2}{2}\right\rceil\right) = 2^{k+1} - 2$ » (le pas est égal à deux car nous nous intéressons ici qu'au nombre entiers impairs uniquement) et montrons que $b(s) = c\left(\left\lceil\frac{2.k+1}{2}\right\rceil\right) = c\left(\left\lceil k+\frac{1}{2}\right\rceil\right) = c(k+1) = 2^{k+2} - 2.$

$b(s) = 2.b(s-2) + 2$ D'après la définition récursive de la suite b(s) ($s > 1$ et impair $\Rightarrow s \geq 3$).

$= 2.c\left(\left\lceil\frac{s-2}{2}\right\rceil\right) + 2 = 2.(2^{k+1} - 2) + 2$ D'après H1

$= 2^{k+2} - 4 + 2 = 2^{k+2} - 2$ CQFD

Conclusion 2 :

Pour tout $s \geq 0$ impair, $b(s) = c(\left\lceil\frac{s}{2}\right\rceil)$

Conclusion :

D'après conclusion1 et conclusion2, Pour tout $s \geq 0$ $b(s) = c(\left\lceil\frac{s}{2}\right\rceil)$.

e) :

Terme général de la suite b(s) :

Pour tout $s \geq 0$, $b(s) = c(\left\lceil\frac{s}{2}\right\rceil)$ D'après la question d).

$= 2(2^{\left\lceil\frac{s}{2}\right\rceil} - 1)$ D'après la question c).

$$= \begin{cases} 2.(2^{k} - 1)\ si\ s = 2.k \\ 2.(2^{k+1} - 1)\ si\ s = 2.k+1 \end{cases}$$

Encadrement du nombre d'appels récursifs réalises par RechercheExaustive(2,[1, 2],*s*) :

Pour tout entier $s \geq 0$, $b(s) \leq a(s) \leq c(s)$ D'après la question b)

$\Leftrightarrow c(\left\lceil\frac{s}{2}\right\rceil) \leq a(s) \leq c(s)$ D'après la question d)

$\Leftrightarrow 2(2^{\left\lceil\frac{s}{2}\right\rceil} - 1) \leq a(s) \leq 2(2^{s} - 1)$ D'après la question c)

Si $s = 2.k$ (cas des s pairs) ($k \geq 0$ *et* k *entier*) :

$$2(2^k - 2) \leq a(s) \leq 2(2^{2.k} - 2)$$

Si $s = 2.k + 1$ (cas des s impairs) ($k \geq 0$ *et* k *entier*) :

$$2(2^{k+1} - 2) \leq a(s) \leq 2(2^{2.k+1} - 2)$$

Complexité temporelle de l'algorithme RechercheExaustive :

-Complexité de RechercheExaustive(2,[1,2],s) :

Si $s = 2.k$ (cas des s pairs) ($k \geq 0$ *et* k *entier*) :

$$2(2^{\frac{s}{2}} - 2) \leq a(s) \leq 2(2^s - 2)$$

Si $s = 2.k + 1$ (cas des s impairs) ($k \geq 0$ *et* k *entier*) :

$$2(2^{\lceil \frac{s}{2} \rceil} - 2) \leq a(s) \leq 2(2^s - 2)$$

On peut en conclure que :
$a(s) \in O(c(s)) \iff a(s) \in O(2^S)$

$a(s) \in \Omega(b(s)) \iff a(s) \in \Omega(2^{\lceil \frac{s}{2} \rceil})$

-exprimons $a(s)$ dans le cas général :

- $Pour\ s = 0$: l'algorithme retourne 0 sans aucun appel récursif donc $a_g(0) = 0$.
- $Pour\ s < 0$: l'algorithme retourne $+\infty$ sans aucun appel récursif donc $a_g(+\infty) = 0$
- $Pour\ s > 0$: d'après l'algorithme on va avoir k itérations, durant chaque itération i on a un appel récursif de RechercheExaustive(k,V,$s - V[i]$) et donc $a_g(s) = k + a(s - V[1]) + a(s - V[2]) + \cdots + a(s - V[k])$.

On en conclut que :

$$a_g(s) = \begin{cases} 0\ si\ s \leq 0 \\ k + a(s - V[1]) + a(s - V[2]) + \cdots + a(s - V[k])\ sinon \end{cases}$$

On remarque que :

Si $s \leq 0$: $a_g(s) \geq a(s)$

Si $s > 0$: $a_g(s) = k + a(s - V[1]) + a(s - V[2]) + \cdots + a(s - V[k]) \geq k + a(s - V[1]) + a(s - V[2]) \geq 2 + a(s - 1) + a(s - 2)$ (car $k \geq 2\ et\ V[2] \geq 2\ et\ V[1] \geq 1$)

Et puisque $a(s) \in O(2^S)$ alors on déduit que la complexité de l'algorithme RechercheExaustive est exponentielle.

2.2 Algorithme II: Programmation dynamique

Réponse à la question 3 :

a) La valeur de $m(s)$ en fonction des valeurs $m(s,i)$ définies dans la section précédente :

$$m(s) = \underbrace{\min}_{i \epsilon \{1,\ldots,k\}} \big(m(s,i)\big) = m(s,k)$$

b) :

Soit $i\epsilon\{1, \dots, k\}$:

- Si $s = 0$ alors $m(0, i) = 0$, aucun bocal n'est nécessaire pour une quantité nulle de confiture.
- Sinon : nous disposons de i types de bocaux de capacités différentes $(V[1], \dots, V[i])$,pour avoir le nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à s en utilisant seulement ces capacités :
  - Soit on n'utilise aucun bocal de capacité maximale $V[i]$ ce qui correspond à $m(s, i-1)$.
  - Soit on utilise au moins un bocal de capacité maximale $V[i]$ ce qui correspond à $m(s - V[i], i) + 1$ (on retranche la capacité $V[i]$ du bocal qui a été rempli de la quantité à stocker et on reconsidère à nouveau cette quantité ce qui correspond à $m(s - V[i], i)$, le $+1$ correspond au bocal de capacité $V[i]$ qui a été rempli)

  On retient alors la meilleure des deux possibilités d'où $\min(m(s, i - 1), m(s - V[i], i) + 1)$

On en conclut que :

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

Réponse à la question 4 :

a) :

On veut calculer m(s,k) :

A chaque appel récursif engendré $m(s, i)$ on doit d'abord calculer $m(s, i - 1)$ puis $m(s - V[i], i) + 1$ pour enfin déduire $m(s, i)$,donc les cases sont remplies suivant un parcours postfixé de l'arbre des appels récursifs.

On en conclut que les cases du tableau sont remplies en utilisant l'approche « du bas vers le haut » (en commençant par les cas de bases) dans l'ordre suivant :

Pour i allant de 0 à s-1 :

Remplir $M(s - i.V[1], 0)$ // $M[s, 0], M[s - 1,0], M[s - 2,0], \dots, M[1,0]$

Ensuite

Pour i allant de 1 à k

Pour j allant de s à 0

Remplir $M(s - j.V[i], i)$

b) Déduction de l'algorithme AlgoProgDyn :

Variable globale $M[\quad][\quad] := -1$ partout

Algorithme AlgoProgDyn(s : entier,k : entier,V : tableau de k entiers) : entier

$op1, op2, min$:entiers

Si $s < 0$ retourner $+\infty$

Si $M[s,k] \geq 0$ retourner $M[s,k]$

Si $(s = 0)$

$M[s,k] := 0$

Retourner $M[s,k]$

Fin si

Si $(k = 0)$

$M[s,k] := +\infty$

Retourner $M[s,k]$

Fin si

$op1 := AlgoProgDyn(s, k-1, V)$

$op2 := AlgoProgDyn(s - V[k], k, V) + 1$

$min :=$ min$(op1, op2)$

$M[s,k] := min$

Retourner $M[s,k]$

c) Analyse de la complexité spatiale et temporelle de l'algorithme :

Complexité spatiale :

M est une matrice de s+1 lignes et k+1 colonnes, en tout $(s+1).(k+1)$ cases.

On en conclut que la complexité spatiale de cet algorithme est en $\Theta((s+1).(k+1)) \simeq \Theta(\mathrm{s.k})$.

Complexité temporelle :

Dans le pire des cas on aura $(s+1).(k+1)$ sous problèmes à résoudre $\{m(0,0), m(0,1), \dots, m(s+1, k+1)\}$ correspondant aux $(s+1).(k+1)$ cases de la matrice M.

On en conclut que la complexité de l'algorithme AlgoProgDyn est en $\mathrm{O}((s+1).(k+1)) \simeq \mathrm{O}(\mathrm{s.k})$

Réponse à la question 5 :

a) :

Les modifications à effectuer dans l'algorithme précèdent :

Il suffit de rajouter comme avant dernière instruction :

Si $op2 < op1$ alors $(M[s,k].A[k]) + +$

La complexité spatiale de cet algorithme :

-Chaque case $M[s,i]$ de la matrice M, contient en plus de la valeur $m(s,i)$ un tableau de taille i, si on note par c le nombre total de cases on obtient :

$$c = \underbrace{(s+1)(k+1)}_{la\ matrice} + \sum_{i=0}^{k}(s+1).i = (s+1)(k+1) + (s+1)\frac{k.(k+1)}{2} = (s+1)(k+1)(\frac{k}{2}+1)$$

On en conclut que la complexité spatiale de cet algorithme est en $\Theta(s.k^2)$

b) :

Description de l'algorithme-retour (backward) :

Algorithme backward(k : entier,V : teableau de k entiers,s : entier) : tableau d'entiers

cpt, i, j, se : entiers

A : tableau d'entiers

Pour $j\ allant\ de$ 1 à k

$A[i] := 0$

Fin pour

$cpt := 0$

$i := k$

$se := s$

Tant que $i > 1$

Si $M[se,i] < M[se,i-1]$

Tant que $((se - V[i] \geq 0)\ et\ (M[se - V[i], i] + 1 < M[se, i-1]))$

$A[i] + +$

$se := se - V[i]$

$cpt := cpt + V[i]$

Fin tant que

Fin si

$i - -$

Fin tant que

$A[1] := s - cpt$

Retourner A

Complexité spatiale de l'algorithme-retour :

M est une matrice de s+1 lignes et k+1 colonnes, en tout $(s+1).(k+1)$ cases.

A est un tableau de k cases.

En tout $(s+1)(k+1)+k = s.k+s+k+k+1$ cases.

On en conclut que la complexité spatiale de backward est $\Theta(s.k+2.k+s+1) \simeq O(s.k)$.

Complexité temporelle de l'algorithme retour :

- La première boucle pour est réalisée en $O(k)$.
- La complexité cumulée pire cas des deux boucles tant que imbriquées est en $O(k+s)$,car dans le pire des cas on utilise uniquement des bocaux de capacité $V[1]$,k itérations pour le tant que externe (i est décrémenté de k à 1), s exécutions du tant que interne pour mettre $A[1]$ à s.

On en conclut que la complexité temporelle de l'algorithme retour est en $O(2.k+s) \simeq O(k+s)$.

Réponse à la question 6 :

L'algorithme AlgoProgDyn : est en $O(s.k)$ (l'algorithme est polynomial)

L'algorithme backward : est en $O(k+s)$ (l'algorithme est polynomial)

L'exécution de l'algorithme AlgoProgDyn suivi de l'algorithme backward est en $O(k+s+s.k) \simeq O(s.k)$ d'où effectivement on peut dire que cet algorithme est polynomial.

2.3 Algorithme III: Cas particuliers et algorithme glouton

Réponse à la question 7 :

Ecrivons l'algorithme glouton en pseudo code :

```
Algorithme AlgoGlouton(k : entier,V : teableau de k entiers,s : entier) : (entier,tableau de k entiers)
    i : entier
    A : tableau de k entiers
    i := k
    cpt := 0
    Tant que s > 0 :
        A[i] := A[i] + (s div V[i])
        cpt := cpt + (s div V[i])
        s := s mod V[i]
        i := i − 1
    Fin tant que
    Retourner (cpt, A)
```

Complexité temporelle de l'algorithme dans le pire cas :

La complexité temporelle de cet algorithme dans le pire des cas est en $O(k)$.

Preuve :

- Les deux premières instructions sont des affectations réalisées en temps constant.
- La boucle tant que contient des instructions réalisées en temps constant, reste à déterminer le nombre de tours de boucle exécutées dans le pire des cas, ce qui va correspondre à la complexité temporelle de cet algorithme.
  La boucle tant que se termine dès que $s \leq 0$ (dans ce cas dès que $s = 0$), la seule instruction qui modifie la valeur de s est $s := s\ mod\ V[i]$, sachant que i est initialisée à k et décrémentée à chaque itération, à chaque itération s contient le reste de la division de s par $V[i]$ (i.e. le reste de confiture après remplissage de tous les bocaux de capacité $V[i]$ possibles en commençant par les plus grandes capacités (i est initialisée à k et décrémentée à chaque itération)) ,on en conclut que dans le pire des cas nous utiliserons des bocaux de capacité $V[1] = 1$ (au moins un), et donc dans ce cas à la kième itération on aura $i = 1, V[1] = 1\ et\ s\ mod\ V[1] = 0$, et donc la boucle tant que se termine après exactement k itérations, et c'est le maximum d'itérations possibles puisque s est un entier positif on aura donc toujours $s\ mod\ V[1] = s\ mod\ 1 = 0$.

Réponse à la question 8 :

En effet il existe des systèmes de capacités qui ne sont pas glouton-compatibles,

Exemple :

-pour un système de 4 capacités $V = (1,250,300,500)$

-pour une quantité totale de confiture $s = 550$

- L'algorithme glouton renvoie :
  - $n_g = 51$
  - $A_g = (50,0,0,1)$

  On a bel et bien $n_g = \sum_{i=1}^{k} A_g[i]$ et $s = \sum_{i=1}^{k} A_g[i].V[i]$
- La solution optimale renvoie :
  - $n_o = 2$
  - $A_o = (0,1,1,0)$

  On a bel et bien $n_o = \sum_{i=1}^{k} A_o[i]$ et $s = \sum_{i=1}^{k} A_o[i].V[i]$

Puisque $n_o < n_g$,donc l'algorithme glouton ne produit pas la solution optimale, ce système de capacités n'est pas glouton-compatible, et donc effectivement il existe des systèmes de capacités qui ne sont pas glouton-compatibles.

Réponse à la question 9 :

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

Puisque la solution o est optimale alors on a que :

$$\sum_{i=1}^{k} o_i \leq \sum_{i=1}^{k} g_i$$

Si on compare un à un les o_i et les g_i, il existe forcement au moins un g_i qui soit strictement supérieur à un o_i, on note par j l'indice de la plus grande valeur des g_i qui vérifient cette condition (i.e. dans le couple (o_i, g_i) considéré on a que $g_i > o_i$) (puisque l'ensemble des g_i est fini cette plus grande valeur existe).

Preuve : supposons par l'absurde que H1 : « pour tous les o_i et les g_i pris un à un $g_i \leq o_i$ (i.e. $\forall 1 \leq i \leq k, g_i \leq o_i$) »

$$\forall 1 \leq i \leq k, g_i \leq o_i \qquad \text{D'après H1}$$

$$\Rightarrow \sum_{i=1}^{k} g_i \leq \sum_{i=1}^{k} o_i$$

- Si $\sum_{i=1}^{k} g_i < \sum_{i=1}^{k} o_i$: la solution o n'est pas optimale, contradiction.
- Si $\sum_{i=1}^{k} g_i = \sum_{i=1}^{k} o_i$: puisque les solutions sont différentes (d'après l'énoncé) alors :
    - 1) Il existe au moins un $1 \leq t \leq k$ tel que $o_t < g_t$.
    - 2) Il existe au moins un $1 \leq \acute{t} \leq k$ tel que $o_{\acute{t}} > g_{\acute{t}}$ (pour équilibrer avec 1) et ainsi préserver l'égalité $\sum_{i=1}^{k} g_i = \sum_{i=1}^{k} o_i$)

  Contradiction de 1) avec H1.

Conclusion : il existe un plus grand indice j pris dans {1,…,k} tel que $o_j < g_j$.

b) Montrons que $\sum_{i=1}^{j} V[i].g_i = \sum_{i=1}^{j} V[i].o_i$:

Montrons que $\forall i \epsilon \{j+1, k\}\ o_i = g_i$:

Soit $i \epsilon \{j+1, k\}$,

- D'après la question précédente j est le plus grand indice pris dans {1,…,k} tel que $o_j < g_j$ et puisque $i > j$ alors $o_i \geq g_i$.
  Conclusion1 : $\forall i \epsilon \{j+1, k\}\ o_i \geq g_i$
- Supposons par l'absurde que $o_i > g_i$,alors il existe un plus grand indice t pris dans $\{j+1, k\}$ telle que $o_t > g_t$,donc $\forall l \epsilon \{t+1, k\}\ o_l \leq g_l$ en combinant ce résultat avec Conclusion1 on conclut que $\forall l \epsilon \{t+1, k\}\ o_l = g_l$, donc $\sum_{i=t+1}^{k} V[i].g_i = \sum_{i=t+1}^{k} V[i].o_i$ (i.e. il reste la même quantité de confiture dans les deux solution après utilisation des bocaux de capacités $\{V[t+1], V[k]\}$) d'après la construction de l'algorithme glouton g_t est maximale et donc $o_t \leq g_t$, contradiction.
  Conclusion2 : $\forall i \epsilon \{j+1, k\}\ o_i \leq g_i$

Conclusion3 : d'après Conclusion1 et Conclusion2 , $\forall i \epsilon \{j+1, k\}\ o_i = g_i$ et donc $\sum_{i=j+1}^{k} V[i].g_i = \sum_{i=j+1}^{k} V[i].o_i$.

On note $v = \sum_{i=j+1}^{k} V[i].g_i = \sum_{i=j+1}^{k} V[i].o_i$.

-Les deux solutions o et g permettent de stocker toutes la quantité s de confiture et donc :

- AlgoGlouton : $s = \sum_{i=1}^{k} V[i].g_i = \sum_{i=1}^{j} V[i].g_i + \sum_{i=j+1}^{k} V[i].g_i = \sum_{i=1}^{j} V[i].g_i + v$.
- Solution optimale : $s = \sum_{i=1}^{k} V[i].o_i = \sum_{i=1}^{j} V[i].o_i + \sum_{i=j+1}^{k} V[i].o_i = \sum_{i=1}^{j} V[i].o_i + v$

$$s = s$$

$$\Leftrightarrow \sum_{i=1}^{j} V[i].g_i + v = \sum_{i=1}^{j} V[i].o_i + v$$

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

CQFD

c) :

Déduisons que $\sum_{i=1}^{j-1} V[i].o_i \geq V[j]$:

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

Or $\sum_{i=1}^{j-1} V[i].g_i \geq 0$

d'où $\sum_{i=1}^{j} V[i].g_i \geq V[j].g_j$ …(1)

$g_j > o_j$ D'après la question a)

$\Rightarrow g_j \geq o_j + 1$ (g_j et o_j sont des entiers naturels).

$\Rightarrow V[j].g_j \geq V[j].(o_j + 1)$

$\Rightarrow \sum_{i=1}^{j} V[i].g_i \geq V[j].(o_j + 1)$ D'après (1)

$\Leftrightarrow \sum_{i=1}^{j} V[i].o_i \geq V[j].o_j + V[j]$ D'après b)

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

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

CQFD

Donnons une interprétation en français de cette propriété à partir du problème de choix de bocaux :

La quantité de confiture qui reste dans la solution optimale après que cette dernière a utilisé tous les bocaux de capacité $V[j]$ de sa configuration de bocaux (i.e. $\sum_{i=1}^{j-1} V[i].o_i$), cette quantité est supérieure à la capacité $V[j]$ d'un bocal de capacité $V[j]$, ce qui signifie que cette solution optimale aurait encore pu utiliser un bocal de capacité $V[j]$ pour dimunuer la quantite de confiture restante mais elle ne l'a pas fait , car ce choix n'est pas globalement optimal , alors que l'algorithme glouton n'aurait pas procédé de la même manière et aurais utilisé un autre bocal de capacité $V[j]$.

d) :

Soit $i\epsilon\{1, \dots, j-1\}$:

$V[i] = d^{i-1}$ D'après l'énoncé

$o_i \leq d-1$ D'après l'hypothèse

On multiplie $\Rightarrow V[i].o_i \leq d^{i-1}.(d-1)$

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

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

- $\sum_{i=1}^{j-1} d^i = d.\frac{1-d^{j-1}}{1-d}$ Somme d'une suite géométrique de raison d.
- $\sum_{i=1}^{j-1} d^{i-1} = \frac{1-d^{j-1}}{1-d}$ Somme d'une suite géométrique de raison d.

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

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

Puisque $\sum_{i=1}^{j-1} V[i].o_i$ et $V[j]-1$ sont des entiers naturels alors :

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

Ce qui constitue une contradiction avec le résultat de la question c) $(\neg\left(\sum_{i=1}^{j-1} V[i].o_i < V[j]\right) = \sum_{i=1}^{j-1} V[i].o_i \geq V[j])$

e) :

On a vu d'après la question précédente que $\forall i \epsilon \{1, \dots, j-1\}\ o_i \leq d-1$ generait une contradiction alors on a que :

$\neg(\forall i \epsilon \{1, \dots, j-1\}\ o_i \leq d-1)$

$\Leftrightarrow \exists i \epsilon \{1, \dots, j-1\}\ o_i > d-1$

$\Leftrightarrow \exists i \epsilon \{1, \dots, j-1\}\ o_i \geq d$ ($o_i\ et d-1$ sont des entiers naturels)

On note cet i par l.

Montrons que *ó* est une solution (i.e. $\sum_{i=1}^{k} ó_i.V[i] = s$) :

$$\sum_{i=1}^{k} ó_i.V[i] = \sum_{\substack{i=1 \\ i\neq l \\ i\neq l+1}}^{k} ó_i.V[i] + ó_l.V[l] + ó_{l+1}.V[l+1]$$

Or d'après l'énoncé $ó_i = o_i$ pour tout $i \neq l\ et\ i \neq l+1$ d'où $\sum_{\substack{i=0 \\ i\neq l \\ i\neq l+1}}^{k} ó_i.V[i] = \sum_{\substack{i=0 \\ i\neq l \\ i\neq l+1}}^{k} o_i.V[i]$

$$\Rightarrow \sum_{i=1}^{k} \acute{o}_i . V[i] = \sum_{\substack{i=1 \\ i\neq l \\ i\neq l+1}}^{k} o_i . V[i] + \acute{o}_l . V[l] + \acute{o}_{l+1} . V[l+1]$$

Or d'après l'énoncé $\acute{o}_l = o_l - d$ et $\acute{o}_{l+1} = o_{l+1} + 1$.

$$\Rightarrow \sum_{i=1}^{k} \acute{o}_i . V[i] = \sum_{\substack{i=1 \\ i\neq l \\ i\neq l+1}}^{k} o_i . V[i] + (o_l - d) . V[l] + (o_{l+1} + 1) . V[l+1]$$

$$= \sum_{\substack{i=1 \\ i\neq l \\ i\neq l+1}}^{k} o_i . V[i] + o_l . V[l] - d . V[l] + o_{l+1} . V[l+1] + V[l+1] = \sum_{i=1}^{k} o_i . V[i] - d . V[l] + V[l+1]$$

Or d'après l'énoncé $V[l+1] = d^l$ et $V[l] = d^{l-1}$.

$$\Rightarrow \sum_{i=1}^{k} \acute{o}_i . V[i] = \sum_{i=1}^{k} o_i . V[i] - d . d^{l-1} + d^l = \sum_{i=1}^{k} o_i . V[i] - d^l + d^l = \sum_{i=1}^{k} o_i . V[i] = s$$

$$\Rightarrow \sum_{i=1}^{k} \acute{o}_i . V[i] = s$$

Même dans le cas ou l n'est pas unique ce résultat reste valable (le raisonnement est le même).

Et donc $\acute{o}$ est une solution du problème.

CQFD

Déduisons que le système Expo est glouton-compatible :

$$\exists l \epsilon \{1, \dots, j-1\}\ o_l \geq d$$

$$V[l] = d^{l-1}$$

$$\Rightarrow\ o_l . V[l] \geq d . d^{l-1} = d^l = V[l+1]$$

- Ce qui signifie qu'on aurait pu prendre un bocal de capacité $V[l+1]$au lieu de d bocal de capacité V$[l]$ (ce qui rejoint la logique de l'algorithme glouton), on effectue alors l'échange $\acute{o}_l = o_l - d$ et $\acute{o}_{l+1} = o_{l+1} + 1$(i.e. on remplace d bocaux de capacité V$[l]$ par un bocal de capacité V$[l+1]$,le nombre de bocaux de capacité V$[l]$ obtenu est $\acute{o}_l$ et le nombre de bocaux de capacité V$[l+1]$ est $\acute{o}_{l+1}$),(un échange peut engendrer un autre échange dans le cas par exemple ou $o_l \geq d$ et $o_{l+1} = d - 1$ on obtient après échange $\acute{o}_l = o_l - d$ et $\acute{o}_{l+1} = d$ et donc $o_{l+1}^{'} \geq d$) en suivant cette logique on va utiliser le plus grand nombre de bocaux de taille plus grande possible , ce qui renvoie exactement la même solution que l'algorithme glouton, reste à montrer que cette solution $\acute{o}$ est optimale .

  Montrons que $\acute{o}$ est optimale :

Supposons par l'absurde que H1 : « $\acute{o}$ n'est pas optimale (i.e. $\sum_{i=1}^{k} o_i < \sum_{i=1}^{k} \acute{o}_i$) »

D'après l'énoncé $\acute{o}_i = o_i$ pour tout $i \neq l\ et\ i \neq l+1$ d'où $\sum^{k}_{\substack{i=1 \\ i\neq l \\ i\neq l+1}} \acute{o}_i = \sum^{k}_{\substack{i=1 \\ i\neq l \\ i\neq l+1}} o_i \dots$ (1)

$$d \geq 2$$

$$o_l - d \leq o_l - 2$$

On a d'après l'énoncé : $\acute{o}_l = o_l - d$

$$\Rightarrow \acute{o}_l \leq o_l - 2$$

$$\Rightarrow \acute{o}_l + \acute{o_{l+1}} \leq o_l - 2 + \acute{o_{l+1}}$$

On a d'après l'énoncé : $\acute{o}_{l+1} = o_{l+1} + 1$

$$\Rightarrow \acute{o}_l + \acute{o_{l+1}} \leq o_l - 2 + o_{l+1} + 1$$

$$\Rightarrow \acute{o}_l + \acute{o_{l+1}} \leq o_l - 1 + o_{l+1}$$

$$\Rightarrow \acute{o}_l + \acute{o_{l+1}} + \sum_{\substack{i=0 \\ i \neq l \\ i \neq l+1}}^{k} \acute{o}_i \leq o_l - 1 + o_{l+1} + \sum_{\substack{i=0 \\ i \neq l \\ i \neq l+1}}^{k} o_i$$

$$\Rightarrow \sum_{i=1}^{k} \acute{o}_i \leq \sum_{i=1}^{k} o_i - 1$$

$\Rightarrow \sum_{i=1}^{k} \acute{o}_i < \sum_{i=1}^{k} o_i$ Car il s'agit d'entiers naturels

Contradiction avec H1

CQFD

D'où le système Expo est glouton-compatible.

Réponse à la question 10 :

Montrons que tout système de capacités V $avec$ $k = 2$ est glouton-compatible :

Supposons par l'absurde qu'Il existe un système de capacités V avec $k = 2$ qui soit non glouton-compatible c'est-à-dire le nombre n de bocaux renvoyé par AlgoGlouton n'est pas minimal, donc il existe une solution optimale qui renvoie $\acute{n}$ le nombre minimum de bocaux tels que la somme de leurs capacités est exactement égale à s et donc $\acute{n} < n$.

Notons par :

- $k_1 = A[1]$ De AlgoGlouton (le nombre de bocaux de capité $V[1]$ utilisés)
- $k_2 = A[2]$ De AlgoGlouton (le nombre de bocaux de capité $V[2]$ utilisés)
  Et donc $n = k_1 + k_2$.
- $\acute{k_1} = A[1]$ De la solution optimale (le nombre de bocaux de capité $V[1]$ utilisés)
- $\acute{k_2} = A[2]$ De la solution optimale (le nombre de bocaux de capité $V[2]$ utilisés)
  Et donc $\acute{n} = \acute{k_1} + \acute{k_2}$.

Cas 1 : $k_2 = \acute{k_2}$ ou $k_1 = \acute{k_1}$:

- $k_2 = \acute{k_2}$:
  Il reste la même quantité de confiture dans les deux solutions après remplissage des bocaux de capacité $V[2]$, elle est égale à $s - V[2].k_2 = s - V[2].\acute{k_2}$, donc on aura forcément le même nombre de bocaux de capacite $V[1]$ dans les deux solutions (i.e. $k_1 = \acute{k_1}$)

$$k_1 = \acute{k_1}$$

$$k_2 = \acute{k_2}$$
$$\Rightarrow k_1 + k_2 = \acute{k_1} + \acute{k_2}$$
$$\Rightarrow n = \acute{n}$$
Contradiction

- $\underline{k_1 = \acute{k_1}}$:
  Il reste la même quantité de confiture dans les deux solutions après remplissage des bocaux de capacité $V[1]$, elle est égale à $s - \mathrm{V}[1].\mathrm{k}_1 = s - \mathrm{V}[1].\acute{\mathrm{k}}_1$, donc on aura forcément le même nombre de bocaux de capacite $V[2]$ dans les deux solutions (i.e. $k_2 = \acute{k_2}$)

$$k_1 = \acute{k_1}$$
$$k_2 = \acute{k_2}$$
$$\Rightarrow k_1 + k_2 = \acute{k_1} + \acute{k_2}$$
$$\Rightarrow n = \acute{n}$$
Contradiction

$\underline{\text{Cas 2 : } \mathrm{k}_2 < \acute{\mathrm{k}}_2}$

D'après la construction de l'algorithme glouton (aucun bocal n'a été sélectionné il reste la même quantité de confiture dans les deux solutions) le nombre de bocaux de taille $V[2]$ sélectionnés (i.e. k_2) est maximale et donc $\mathrm{k}_2 \geq \acute{\mathrm{k}}_2$,contradiction.

$\underline{\text{Cas 3 : } \mathrm{k}_2 > \acute{\mathrm{k}}_2 \; et \; \mathrm{k}_1 > \acute{\mathrm{k}}_1}$:

$$\mathrm{k}_2 > \acute{\mathrm{k}}_2$$
$$\Rightarrow V[2].\mathrm{k}_2 > \mathrm{V}[2].\acute{\mathrm{k}}_2 \ldots (1)$$
$$\mathrm{k}_1 > \acute{\mathrm{k}}_1$$
$$\Rightarrow V[1].\mathrm{k}_1 > \mathrm{V}[1].\acute{\mathrm{k}}_1 \ldots (2)$$
$$(1) et (2) \Rightarrow V[1].\mathrm{k}_1 + V[2].\mathrm{k}_2 > \mathrm{V}[1].\acute{\mathrm{k}}_1 + \mathrm{V}[2].\acute{\mathrm{k}}_2$$
$$\Leftrightarrow s > s$$
Contradiction

$\underline{\text{Cas 4 : } \mathrm{k}_2 > \acute{\mathrm{k}}_2 \; et \; \mathrm{k}_1 < \acute{\mathrm{k}}_1}$:

- Algorithme glouton : après utilisation des bocaux de capacité $\mathrm{V}[2]$ il reste la quantité $s - \mathrm{V}[2].\mathrm{k}_2$.
- Solution optimale : après utilisation des bocaux de capacité $\mathrm{V}[2]$ il reste la quantité $s - \mathrm{V}[2].\acute{\mathrm{k}}_2$.

D'où :

$\left(s - \mathrm{V}[2].\acute{\mathrm{k}}_2\right) - (s - \mathrm{V}[2].\mathrm{k}_2) = \mathrm{V}[2].\mathrm{k}_2 - \mathrm{V}[2].\acute{\mathrm{k}}_2 = \left(\mathrm{k}_2 - \acute{\mathrm{k}}_2\right).V[2]$ Représente le nombre de bocaux de capacité $\mathrm{V}[1] = 1$ que la solution optimale a de plus que la solution gloutonne et c'est aussi égal a $\acute{\mathrm{k}}_1 - k_1 \ldots$ (1)

- $\mathrm{k}_2 - \acute{\mathrm{k}}_2$ Représente le nombre de bocaux de capacité $\mathrm{V}[2]$ que la solution gloutonne a de plus que la solution optimale … (2)

D'après (1) et (2) :

$$n = \acute{n} + \left(\mathrm{k}_2 - \acute{\mathrm{k}}_2\right) - \left(\acute{k_1} - k_1\right) = \acute{n} + \left(\mathrm{k}_2 - \acute{\mathrm{k}}_2\right) - \left(\mathrm{k}_2 - \acute{\mathrm{k}}_2\right).V[2] = \acute{n} + \left(\mathrm{k}_2 - \acute{\mathrm{k}}_2\right).(1 - V[2])$$

Or :

$$V[2] > 1$$

$$\Rightarrow -V[2] < -1$$

$$\Rightarrow 1 - V[2] < 0$$

Et $\mathrm{k}_2 > \acute{\mathrm{k}_2} \Rightarrow \mathrm{k}_2 - \acute{\mathrm{k}_2} > 0$

$$\Rightarrow (\mathrm{k}_2 - \acute{\mathrm{k}_2}).(1 - V[2]) < 0$$

$$\Rightarrow \acute{n} > n$$

Contradiction avec l'hypothèse $\acute{n} < n$

Dans tous les cas possibles on trouve une contradiction, donc on peut conclure que tout système de capacités $V\ avec\ k = 2$ est glouton-compatible.

Réponse à la question 11 :

Prouvons que cet algorithme de test est polynomial en donnant sa complexité :

s Varie de $V[3] + 2$ à $V[k-1] + V[k] - 1$ et pour chaque valeur de s :

j Varie de 1 à k et pour chaque valeur de j :

- Test 1 : $V[j] < s$ est réalisé en temps constant.
- Test 2 : $AlgoGlouton(s) > 1 + AlgoGlouton(s - V[j])$:
  - $AlgoGlouton(s)$ Qui est réalisé en $\mathrm{O}(k)$ (d'après question7)
  - $1 + AlgoGlouton(s - V[j])$ Qui est réalisé en $\mathrm{O}(k)$ (d'après question7)

On en déduit que la complexité temporelle pire cas de cet algorithme est en :

$$\mathrm{O}\Big(\sum_{s=V[3]+2}^{V[k-1]+V[k]-1} \sum_{j=1}^{k} k\Big)$$

Calculons $\sum_{s=V[3]+2}^{V[k-1]+V[k]-1} \sum_{j=1}^{k} k$:

$$\sum_{j=1}^{k} k = k^2$$

$$\sum_{s=V[3]+2}^{V[k-1]+V[k]-1} k^2 = (V[k-1] + V[k] - V[3] - 2)k^2$$

D'où : $\sum_{s=V[3]+2}^{V[k-1]+V[k]-1} \sum_{j=1}^{k} k = (V[k-1] + V[k] - V[3] - 2)k^2 = V[k-1].k^2 + V[k].k^2 - V[3].k^2 - 2.k^2$.

On en conclut que cet algorithme est en $\mathrm{O}(V[k].k^2)$ et donc il est bel et bien polynomial.

3 Mise en œuvre:

3.1 Implémentation

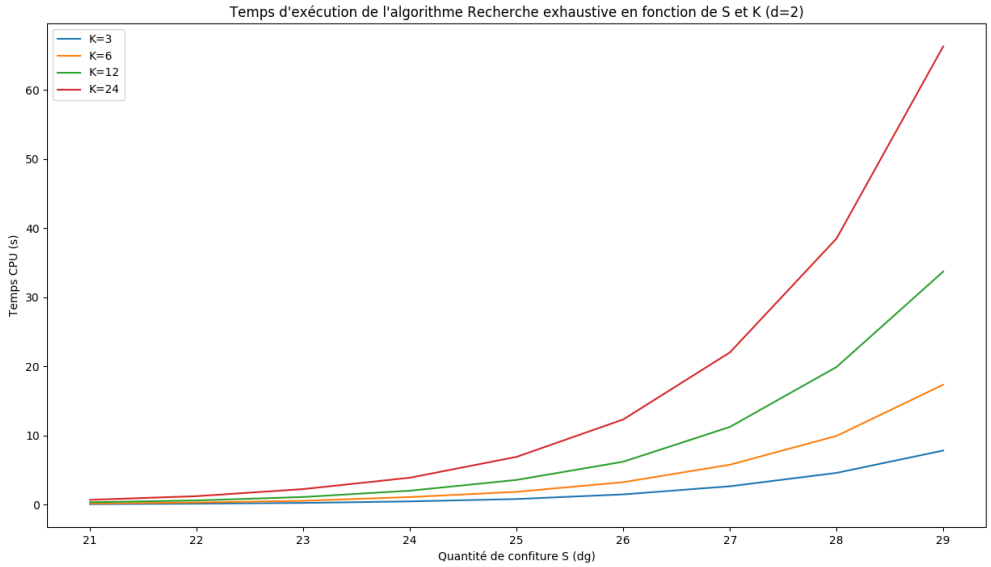
Voir les fichiers de code.

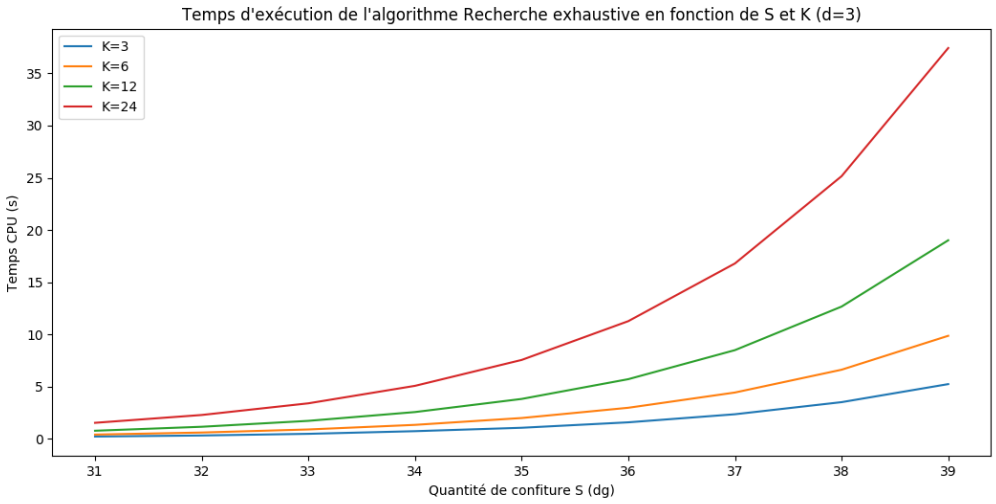
3.2 Analyse de complexité expérimentale :

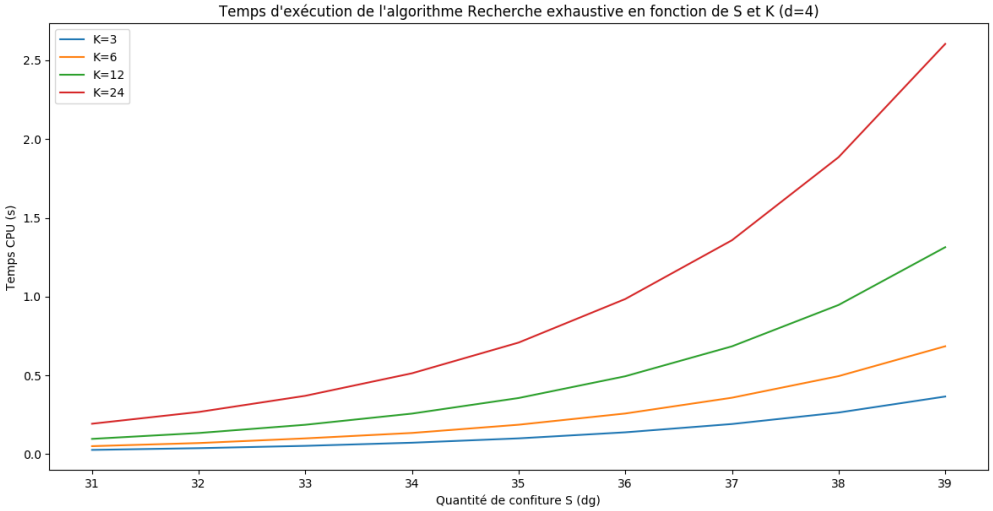
Réponse à la question 12 :

a) Analysons de façon critique la valeur expérimentale obtenue en fonction des complexités théoriques dans les trois cas :

1-Recherche exhaustive :

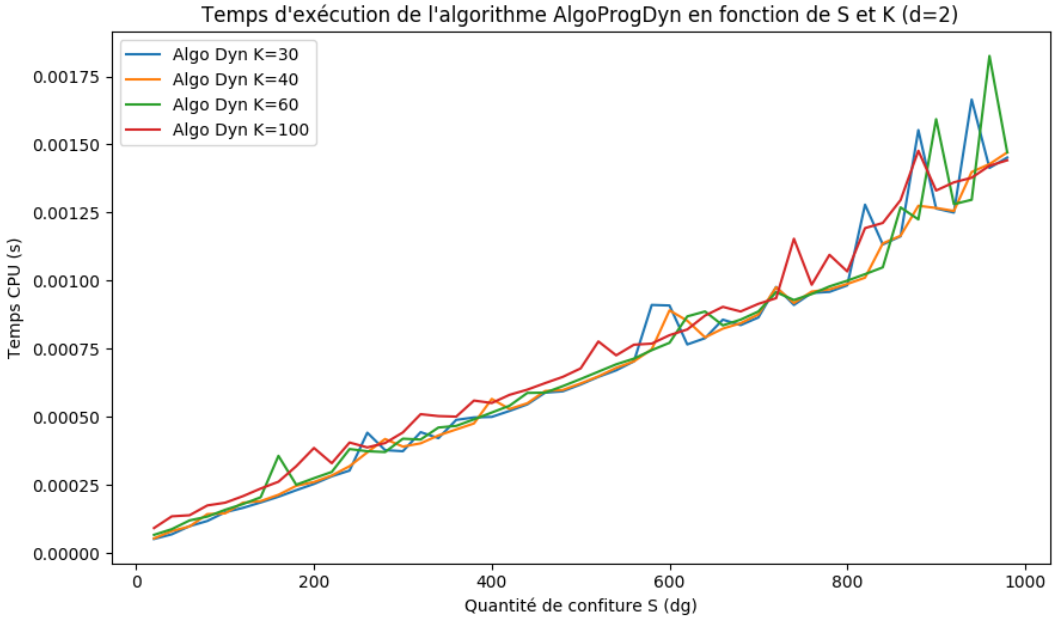


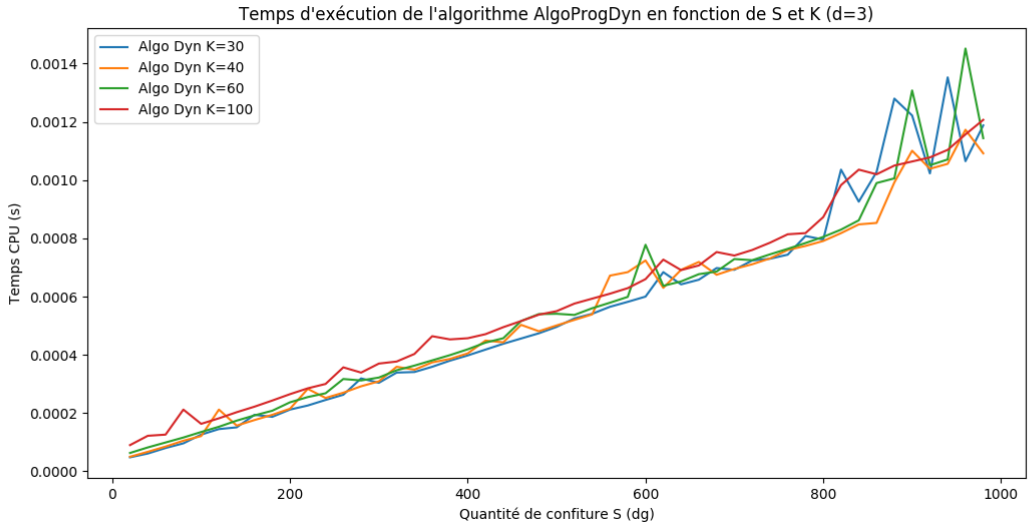


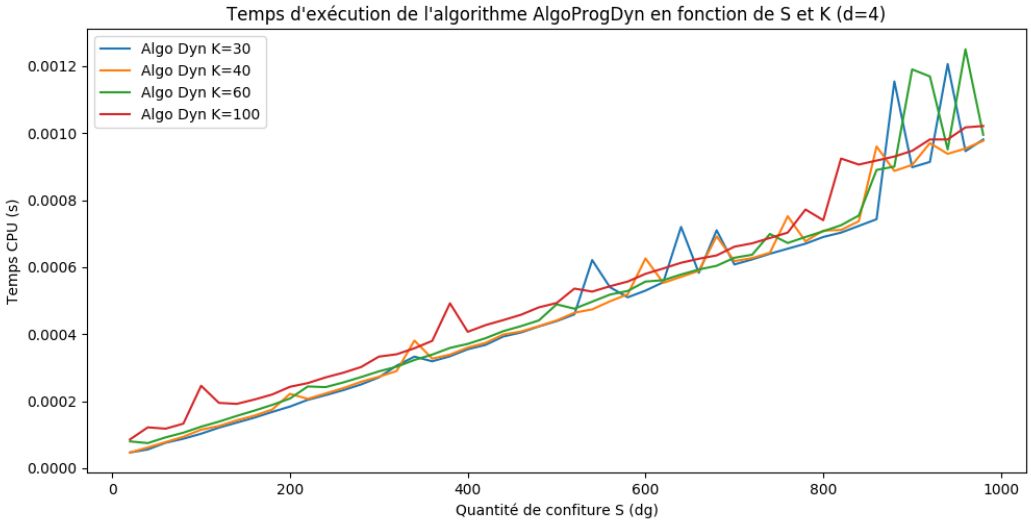


Les trois graphes ci-dessus représentent l'évolution du temps cpu en secondes de l'exécution de RechercheExaustive en fonction de la quantité de confiture S et ce pour des valeurs différentes de k (le nombre de capacités différentes de bocaux) , on remarque que le temps cpu croit exponentiellement en fonction de la quantité de confiture s , on remarque aussi que la croissance exponentielle est plus rapide pour les grandes valeurs de k, ce qui confirme le résultat théorique obtenu dans la partie e) de la question 2 (la complexité de l'algorithme RechercheExaustive est exponentielle).

2-Programmation dynamique :

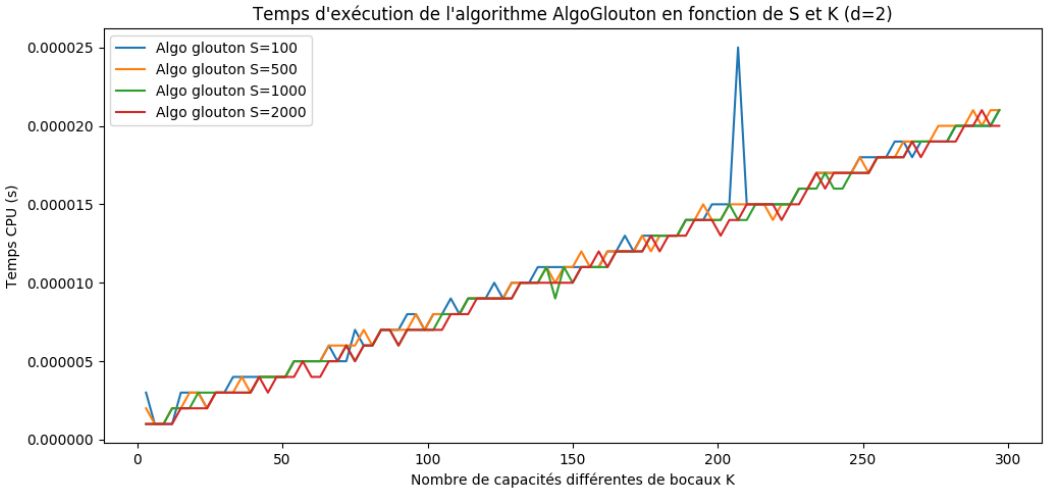


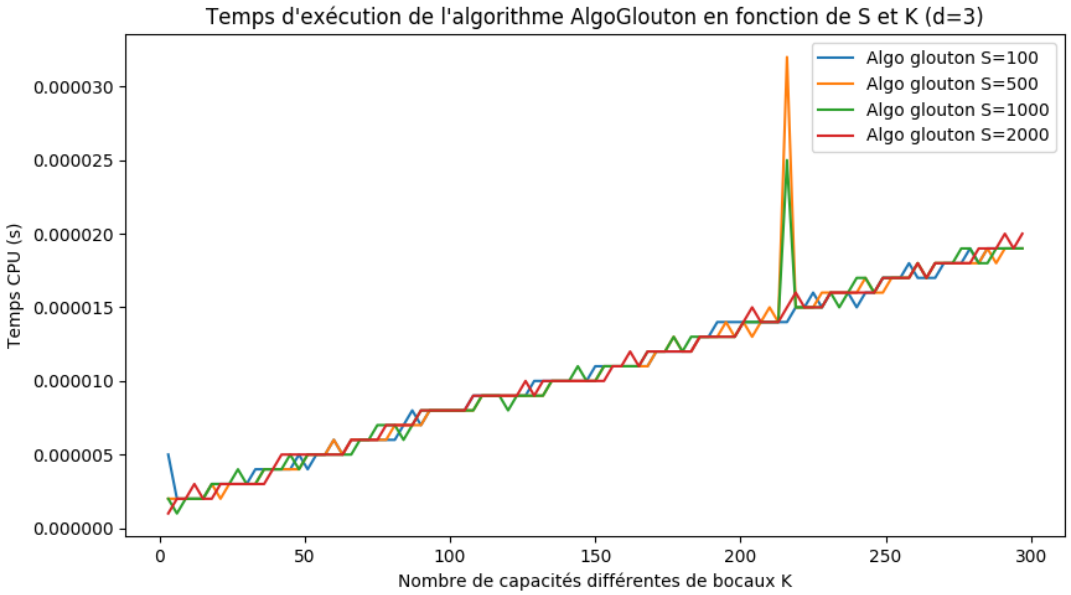


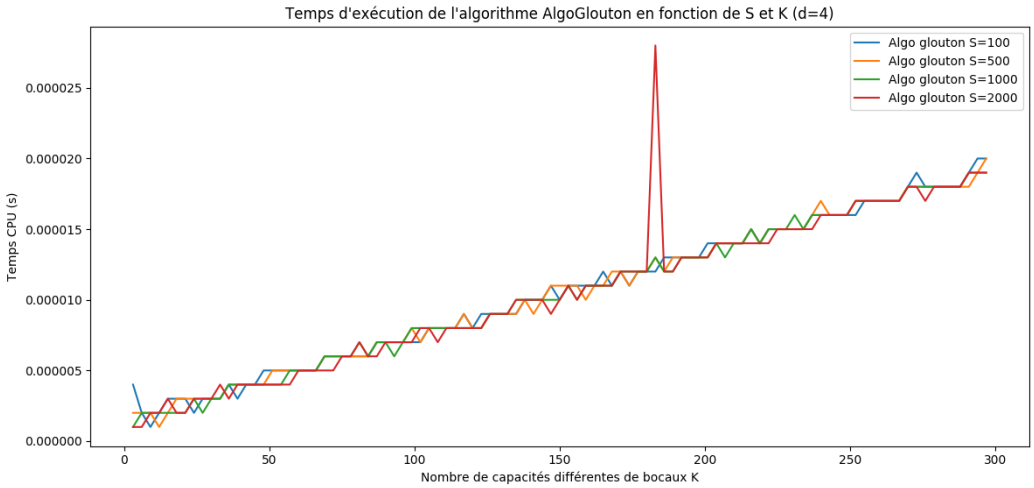


Les trois graphes ci-dessus représentent l'évolution du temps cpu en secondes de l'exécution de AlgoProgDyn en fonction de la quantité de confiture S et ce pour des valeurs différentes de k (le nombre de capacités différentes de bocaux), on remarque que pour k fixé le temps cpu croit linéairement en fonction de s, on remarque aussi que les valeurs du temps cpu sont plus grandes pour les grandes valeurs de k, ce qui rejoint le résultat théorique, AlgoProgDyn est polynomial.

3-Algorithme glouton :







Les trois graphes ci-dessus représentent l'évolution du temps cpu en seconds de l'exécution de AlgoGlouton en fonction du nombre de capacités différentes k et ce pour des valeurs différentes de s (la quantité de confiture) .Il est clair que dans les trois cas l'évolution du temps cpu croit linéairement en fonction de k, et cette croissance linéaire ne dépend pas de la valeur de s ,puisque on remarque que dans les trois figures les courbes pour toutes les valeurs différentes de s sont superposées et donc l'évolution du temps cpu est indépendante de s , ce qui confirme le résultat de la question 7 (la complexité de AlgoGlouton est en $O(k)$).

c) <u>Comparons les performances des algorithmes :</u>

Il est clair d'après les graphes ci-dessus que pour un système de capacité Expo : l'algorithme AlgoGlouton est le plus rapide suivi par l'algorithme AlgoProgDyn suivi par l'algorithme RechercheExhaustive qui prend le plus de temps, par exemple :

AlgoGlouton : pour s=100dg et k=50 et d=4 on obtient d'après les graphes un temps cpu t=0,000005s

AlgoProgDyn : pour s=100dg et k=50 et d=4 on obtient d'après les graphes un temps cpu t=0,001s

RechercheExhaustive : pour s=39dg et k=24 et d=4 on obtient d'après les graphes un temps cpu t=2,5s

Ces résultats observés à partir des courbes sont en cohérence avec les résultats obtenues théoriquement, puisque nous avons vu que :

- AlgoGlouton est en $O(k)$.
- AlgoProgDyn est polynomial
- RechercheExhaustive est exponentielle

Remarque : Dans le cas où le système de capacités n'est pas glouton-compatible, même si AlgoGlouton est le plus rapide il ne renvoie pas toujours une solution qui est optimale.

Réponse à la question 13 :

Principe du programme :

- On fait varier une variable i de 1 à un nombre suffisamment grand g :
  - On génère aléatoirement un system de capacité de taille i, les valeurs V[2],…,V[i] sont aléatoirement prises dans l'intervalle [2,p_{max}].
  - Si ce système est glouton compatible, on incrémente le compteur true.
  - Sinon on incrémente le compteur false.
- Enfin la proportion de systèmes glouton-compatibles parmi l'ensemble des systèmes possibles est donnée par la formule $p := \frac{true}{true+false}$

Résultat expérimentaux obtenu :

Pour un $p_{max} = 8000$ et $g = 2000$ on obtient :

- $true = 2$
- $false = 1997$
- $p = 0{,}001$

Après plusieurs tests, on remarque que presque toujours, $true = 2$, les systèmes glouton-compatibles détectés correspondent aux systèmes de capacité de taille 1 et 2

- Systèmes de capacité de taille 1 sont glouton compatibles (solution unique et triviale s bocal de capacité V[1])

- Systèmes de taille 2, d'après la question 10 sont glouton-compatible.

A part ces deux cas il est très rare de trouver un système qui soit glouton-compatible.

Voir le fichier de code « Proportion_glouton-compatible.py »

Réponse à la question 14 :

Principe du programme :

- On fait varier une variable i de 1 à un nombre suffisamment grand j :
  - On génère aléatoirement un system de capacité de taille i, les valeurs V[2],…,V[i] sont aléatoirement prises dans l'intervalle [2,p_{max}].
  - Si ce système est non glouton-compatible
    - On exécute AlgoProgDyn et AlgoGlouton pour chaque quantité s entre p_{max} (la capacité du plus grand bocal) et $f.p_{max}$ (f entier suffisamment grand), on cumul dans une variable l'écart entre les deux solutions et on stocke aussi le pire écart entre les deux solutions.
  - On pourra alors pour chaque système déterminer le pire écart et l'écart moyen.
- On pourra aussi déterminer l'écart moyen moyen (la moyenne des écarts moyens) et générer des statistiques sur l'ensemble des systèmes générés (glouton-compatible ou non) (nombre des systèmes non glouton compatibles parmi ceux générés et l'écart moyen moyen en considérant les systèmes gloutons compatibles)

Résultat expérimentaux obtenu :

Pour $j = 20, p_{max} = 800, f = 500$

- L'écart moyen moyen pour les systèmes non glouton-compatibles est de : 39,18
- Le pire écart est de : 502
- L'écart moyen moyen en prenant en considération tous les systèmes : 33,30
- Le nombre de systèmes non glouton-compatibles parmi ceux générés : 17 (parmi 20)

Voir le fichier de code « Ecarts_Statistiques.py »

4 Conclusion

Au terme de ce mini-projet nous avons pu comparer l'efficacité de trois algorithmes différents résolvants un même problème. Grâce aux outils mathématiques à notre portée et aux informations acquises tout au long du semestre, nous avons été capables de réaliser une étude théorique de ces algorithmes et en particulier de leurs complexités, et de vérifier par la suite ces résultats théoriques en implémentant ces algorithmes.