Sorbonne Université

3I003 : Algorithmique

Projet :

Confitures

Ahmed Boukerram
Angelo Ortiz

Licence d'Informatique
Année 2018/2019

Table des matières

Introduction

Sur un forum de cuisine, on pourrait trouver une question portant sur le nombre minimal de pots nécessaires pour la préservation d'une certaine quantité de confiture préparée. Il est important de remplir au complet chaque bocal afin de prolonger au maximum le temps de conservation du produit. Mais la capacité des bocaux n'est pas la même pour tous. C'est pourquoi on veut proposer des algorithmes résolvant cette question : on se fixe pour objectif d'utiliser le moins possible de bocaux.

Pour définir ce problème, on dispose des données suivantes. Il y a S décigrammes de confiture que l'on doit verser dans des bocaux vides. On compte aussi plusieurs bocaux de diverses capacités que l'on range en k classes : chaque classe de bocal correspond à une capacité distincte. On appellera *système de capacités* l'ensemble des capacités dont on dispose. On note ces k capacités par un tableau V de taille k numéroté par ordre croissant :

- $V[1] < V[2] < \cdots < V[k]$; et
- chaque capacité $V[i]$ est exprimée par la quantité en décigrammes que l'on peut mettre dans un bocal.

On se fixe aussi certaines contraintes qui garantissent l'existence d'une solution au problème :

- la quantité S est un nombre entier de décigrammes ;
- la capacité minimal d'un bocal est 1 décigramme, i.e. $V[1] = 1$; et
- on dispose d'une très grande quantité (supposée ainsi illimitée) de bocaux de chacune des capacités.

En effet, en tenant compte des trois hypothèses, on remarque que l'on peut toujours verser la totalité de confiture dans des pots de 1 décigramme.

Notre objectif est de remplir le moins possible de bocaux et qu'ils soient tous remplis exactement à sa capacité maximale. Ainsi, étant donnés :

— un système de k capacités $V[i] \in \mathbb{N}$, $i \in \{1, \ldots, k\}$, avec $V[1] = 1$,

— et une quantité totale $S \in \mathbb{N}$ de confiture,

le but est de déterminer le nombre minimum de bocaux tels que la somme de leurs capacités est égale à S. On cherche donc à retourner un couple (n, A), où n est le nombre de bocaux utilisés et A est un tableau de taille k tel que $A[i]$ représente le nombre de bocaux de capacité $V[i]$ à remplir au complet. On a ainsi que $n = \sum_{i=1}^{k} A[i]$.

Les objectifs de ce projet sont l'analyse théorique et la mise en œuvre de trois algorithmes résolvant le problème décrit ci-dessus. Dans un premier temps, on formalise le problème et propose plusieurs algorithmes dont on fera l'analyse de complexité. Dans un deuxième temps, on implémente ces algorithmes au moyen du langage de programmation Python afin de vérifier expérimentalement les complexités trouvées précédemment.

Première partie
Partie théorique

1 Algorithme I : Recherche exhaustive

Question 1

Soit la propriété suivante $P_1(s)$: `RechercheExhaustive`(k, V, s) se termine et renvoie le nombre minimal de bocaux à remplir pour une quantité s de confiture avec un système de k capacités décrit dans les cases du tableau $V[1..k]$. Montrons-la par récurrence forte sur s.

Preuve

Cas négatif : Pour $s < 0$, la quantité de confiture est négative. Dans ce cas, on ne peut pas remplir les bocaux et on dit que le nombre de bocaux est infini. Par ailleurs, dans le corps de l'algorithme, on rentre dans le bloc `then` du premier branchement conditionnel et renvoie bien $+\infty$. De plus, on n'a effectué que des instructions élémentaires en nombre fini. De ces faits, `RechercheExhaustive` est valide et se termine dans le cas négatif.

Base : Pour $s = 0$, la quantité de confiture est nulle. Aussi, on rentre dans le bloc `else` du premier branchement conditionnel, puis dans le bloc `then` du deuxième branchement conditionnel. Donc, on renvoie 0, ce qui est correct puisque aucun bocal ne doit être rempli.

De la même manière que dans le cas où $s < 0$, on montre ici que l'algorithme se termine car la suite d'instructions effectuées ne comporte que des instructions élémentaires. De ces faits, $P_1(0)$ est vraie.

Induction : Supposons $P_1(s)$ vraie pour tout $s < s_0$, pour un $s_0 > 0$ fixé. Montrons $P_1(s_0)$ vraie.

Comme la quantité de confiture s_0 est positive, on a besoin d'*au moins* un bocal où la verser. On remarque qu'il n'y a pas de contrainte sur ce premier bocal. Pour s'assurer de bien calculer le nombre minimum de bocaux à remplir, on essaie donc toutes les capacités possibles pour le premier bocal, puis on résout le problème récursif induit par chacune d'entre elles. De cette manière il suffit de choisir la configuration optimale parmi les k possibles.

Par ailleurs, en ce qui concerne l'algorithme `RechercheExhaustive`, on rentre dans le bloc `else` du premier branchement conditionnel, puis dans le bloc `else` du deuxième branchement conditionnel, puisque $s_0 > 0$. On sait que la capacité minimum est 1 dg. On en déduit que le nombre *maximum* de bocaux à utiliser est s_0.

On cherche à améliorer la solution de départ. Pour ce faire, on teste chaque capacité $V[i], i \in \{1, \ldots, k\}$, pour le premier bocal à remplir. Il reste ainsi $s_0 - V[i]$ décigrammes de confiture à verser. D'après les conditions du problème, $V[i] \geq 1$, d'où $s_0 - V[i] < s_0$. Par hyporhèse de récurrence, `RechercheExhaustive`(k, V, s),

où $s = s_0 - V[i]$, se termine et est valide. On rajoute 1 à cette solution en raison du premier choix. On teste ensuite si cette solution améliore la solution courante et, le cas échéant, on la met à jour. On répète ce procédé pour chacune des capacités du tableau $V[1..k]$.

Comme mentionné précédemment, à la fin de l'examen des k capacités pour le premier bocal, on se retrouve avec la solution optimale. Étant donné que l'appel `RechercheExhaustive`(k, V, s_0) renvoie cette solution optimale, il est valide. Par ailleurs, par hypothèses de récurrence, les appels récursifs se terminent. On effectue k tours de boucle du bloc `for` qui contient ces appels récursifs et quelques instructions élémentaires de plus. On en déduit que `RechercheExhaustive`(k, V, s_0) se termine.

Conclusion :

$$\left.\begin{array}{l} P_1(0) \text{ vraie} \\ \forall s_0 > 0, [(\forall s < s_0, P_1(s)) \Rightarrow P_1(s_0)] \end{array}\right\} \begin{array}{l} \forall s \in \mathbb{N}, \texttt{RechercheExhaustive}(k, V, s) \\ \text{se termine et est valide.} \end{array}$$

Cas particulier

D'après le résultat précédant, pour $s = S \in \mathbb{N}$ (la quantité de confiture dans le cadre de ce projet), on a que l'algorithme est valide et se termine lorsqu'il est appelé par l'appel initial `RechercheExhaustive`(k, V, S).

Question 2

Soient $b(s)$ et $c(s)$ les suites définies par

$$b(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ 2 \cdot b(s-2) + 2 & \text{si } s \geq 2 \end{cases} \text{ et } c(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 \cdot c(s-1) + 2 & \text{si } s \geq 1 \end{cases}$$

Supposons que les seules capacités de bocal disponibles soient 1 dg et 2 dg. Soit $a(s)$ le nombre d'appels récursifs effectués par `RechercheExhaustive`$(2, [1, 2], s)$.

a) La suite $a(s)$ est définie par

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

b) Soit la propriété suivante $P_2(s), s \geq 0 : b(s) \leq a(s) \leq c(s)$. Montrons-la par récurrence d'ordre 2 sur s.

Preuve

Base : Pour $s = 0$, on a $b(0) = a(0) = c(0) = 0$. Donc, $P_2(0)$ est vraie.
Pour $s = 1$, on a $b(1) = a(1) = c(1) = 2$. Donc, $P_2(1)$ est vraie.

Induction : Supposons $P_2(s-2)$ et $P_2(s-1)$ vraies pour un $s \geq 2$ fixé. Montrons $P_2(s)$ vraie.
D'après les hypothèses de récurrence, on a :

$$b(s-2) \leq a(s-2) \leq c(s-2) \tag{1}$$
$$b(s-1) \leq a(s-1) \leq c(s-1) \tag{2}$$

Par définition de ces suites récursives, on a :

$$b(s) = 2 \cdot b(s-2) + 2 \tag{3}$$
$$a(s) = a(s-2) + a(s-1) + 2 \tag{4}$$
$$c(s) = 2 \cdot c(s-1) + 2 \tag{5}$$

Étant donné que les suites $b(s)$ et $c(s)$ sont croissantes (c.f. annexe A), les inégalités (1) et (2) deviennent respectivement

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

Il en résulte que

$$2 \cdot b(s-2) + 2 \leq a(s-2) + a(s-1) + 2 \leq 2 \cdot c(s-1) + 2 \tag{6}$$

En remplaçant les expressions (3), (4) et (5) en (6), on obtient les inégalités cherchées : $b(s) \leq a(s) \leq c(s)$.

Conclusion :

$$\left.\begin{array}{l} P_2(0) \text{ et } P_2(1) \text{ vraies} \\ \forall s \geq 2, [(P_2(s-2) \text{ et } P_2(s-1)) \Rightarrow P_2(s)] \end{array}\right\} \forall s \in \mathbb{N}, b(s) \leq a(s) \leq c(s)$$

c) Soit la propriéte suivante $P_3(s)$, $s \geq 0$: $c(s) = 2^{s+1} - 2$. Montrons-la par récurrence faible sur s.

Preuve

Base : Pour $s = 0$, on a $c(0) = 0$. De plus, $2^{0+1} - 2 = 2^1 - 2 = 0$. Donc, $P_3(0)$ est vraie.

Induction : Supposons $P_3(s)$ vraie pour un $s \geq 0$ fixé. Montrons $P_3(s+1)$ vraie.
On a $s + 1 \geq 1$. Par définition, $c(s+1) = 2 \cdot c(s) + 2$. Or, par hypothèse de récurrence, $c(s) = 2^{s+1} - 2$.
Donc, $c(s+1) = 2 \cdot (2^{s+1} - 2) + 2 = 2^{s+2} - 2$. D'où, $P_3(s+1)$ est vraie.

Conclusion :

$$\left.\begin{array}{l} P_3(0) \text{ vraie} \\ \forall s \geq 0, [P_3(s) \Rightarrow P_3(s+1)] \end{array}\right\} \forall s \in \mathbb{N}, c(s) = 2^{s+1} - 2$$

d) Faire

e) On déduit des deux questions précédentes que $b(s) = 2^{\lceil \frac{s}{2} \rceil + 1} - 2$.
D'après la question b), on a $b(s) \leq a(s) \leq c(s)$.
De ces faits, $2^{\lceil \frac{s}{2} \rceil + 1} - 2 \leq a(s) \leq 2^{s+1} - 2$.
Pour le calcul de la complexité temporelle de l'algorithme, considérons seulement les appels récursifs effectués par `RechercheExhaustive`$(2, [1, 2], s)$. La complexité pour ce système de deux capacités est donnée par $a(S)$. La complexité est ainsi en $\Omega\left(\sqrt{2}^S\right)$ et $\mathcal{O}(2^S)$.
N.B. Pour la complexité de cet algorithme dans le cas général, regardez l'annexe B.

2 Algorithme II : Programmation dynamique

Question 3

On note $m(S)$ le nombre minimum de bocaux pour S décigrammes de confiture et un tableau de capacités V. On définit une famille de problèmes intermédiaires de la façon suivante : étant donnés un entier s et un entier $i \in \{1, \ldots, k\}$, on note $m(s, i)$ le nombre minimum de bocaux nécessaires pour une quantité totale s en ne choisissant des bocaux que dans le système de capacités $V[1], V[2], \ldots, V[i]$.
On pose :

$$\begin{aligned} m(0, i) &= 0 && \forall i \in \{1, \ldots, k\} \\ m(s, 0) &= +\infty && \forall s \geq 1 \\ m(s, i) &= +\infty && \forall i \in \{1, \ldots, k\}\ \forall s < 0 \end{aligned}$$

a) On a $m(S) = m(S, k)$.

b) Soit la relation de récurrence suivante pour tout $i \in \{1, \ldots, k\}$

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

Montrons-la par récurrence.

Preuve

Soit s la capacité de confiture à traiter. Deux cas sont possibles :

i) $\underline{s = 0 :}$ Dans ce cas, il n'y a pas de confiture à préserver. On n'utilise donc aucun bocal, i.e. on a $m(0, i) = 0$. ■

ii) $\underline{s \neq 0 :}$ Puisque $m(s, i) = +\infty$ pour toute quantité $s < 0$, la relation de récurrence ne concerne dans ce cas que les quantités $s > 0$. De même, on a $m(s, 0) = +\infty$ pour toute quantité $s \geq 1$. Donc, on ne considère que les $i > 0$.

Étant donné le système de capacités $V[1], V[2], \ldots, V[i]$, deux choix sont possibles : soit on utilise des bocaux de capacité $V[i]$, soit on ne s'en sert plus.

D'un côté, le premier choix implique forcément l'utilisation d'*au moins* un bocal de ladite capacité. Ceci se traduit mathématiquement par $m(s - V[i], i) + 1$.

D'un autre côté, le deuxième choix rejete les bocaux de ladite capacité. Donc, le résultat est donné par $m(s, i - 1)$.

Comme on cherche à minimiser le nombre de bocaux à utiliser, $m(s, i)$ doit valoir le minimum entre ces deux résultats. ■

Question 4

Soit M un tableau doublement indicé tel que la case $M[s, i]$, où $s \in \{0, \ldots, S\}$ et $i \in \{0, \ldots, k\}$, contient la valeur $m(s, i)$.

a) Soit le couple $(s, i) \in \{1, \ldots, S\} \times \{1, \ldots, k\}$. D'après la relation de récurrence prouvée ci-dessus, pour calculer $m(s, i)$, on nécessite des valeurs stockées dans les cases $(s, i-1)$ et $(s-V[i], i)$, si $s-V[i] \geq 0$. Autrement dit, la case à gauche et toutes les cases au-dessus de (s, i) doivent être traitées préalablement au calcul de la valeur de cette dernière.

   Cela implique que le remplissage du tableau doit commencer à la case $(0, 0)$ et aller jusqu'à la case (S, k) en suivant un parcours matriciel. Ce parcours peut donc se faire soit par lignes, soit par colonnes.

   N.B. Dans le cadre du projet, on a sélectionné le parcours séquentiel des lignes.

b) On déduit des deux questions précédentes un algorithme `AlgoProgDyn` qui pour un système de k capacités $V[1], V[2], \ldots, V[k]$ et une quantité S de confiture retourne $m(S)$. En voici le pseudocode :

c) Pour l'analyse de la complexité temporelle on utilise l'accès à une case du tableau M comme instruction élémentaire. Ainsi, on remarque que pour chacune des $(S + 1) \cdot (k + 1)$ itérations de la boucle interne on accède au plus 5 fois à des éléments du tableau : l'une pour l'écriture de la case et les quatre restantes pour les lectures des cases dont on cherche le minimum des valeurs. Par ailleurs, la création et l'initialisation de la matrice M se font en $\Theta(S \cdot k)$. Donc, la complexité temporelle de l'algorithme est en $\Theta(S \cdot k)$.

   Quant à la complexité spatiale, on se sert de deux structures de données gourmandes, à savoir le tableau V de k cases d'entiers et la matrice M de $(S + 1) \cdot (k + 1)$ cases d'entiers. En prenant comme taille de base celle d'un entier, on réalise que la complexité spatiale de l'algorithme est aussi en $\Theta(S \cdot k)$.

Question 5

On désire à présent permettre à l'algorithme de retourner un tableau A indiquant le nombre de bocaux pris pour chaque type de capacités.

Algorithm 1 AlgoProgDyn

Require: $k \geq 0$ **and** $S \geq 0$

```
1: function ConfitureForwards(k : integer, V : k-integer array, S : integer)
2:     M : (S + 1) × (k + 1) integer matrix filled with +∞
3:     for s = 0 to S do
4:         for i = 0 to k do
5:             M[s][i] ← GetMsi(V, M, s, i)
6:     return M[S][k]                                  ▷ the value of m(S)

7: function GetMsi(V : k-integer array, M : (S + 1) × (k + 1) integer matrix,
   s : integer, i : integer)
8:     if s = 0 then
9:         return 0
10:    else if s < 0 then
11:        return +∞
12:    else if i = 0 then
13:        return +∞
14:    else if M[s][i] < +∞ then                       ▷ M[s][i] already calculated
15:        return M[s][i]
16:    else
17:        return min(GetMsi(V, M, s, i − 1), GetMsi(V, M, s − V[i], i) + 1)
```

a) Dans un premier temps, on décide de placer dans chaque case du tableau $M[s, i]$, un tableau A indiquant les bocaux pris pour la capacité totale s et des capacités de bocaux prises parmi $V[1], \ldots, V[i]$.

   Pour ce faire, on effectue quelques modifications à l'algorithme `AlgoProgDyn`. Tout d'abord, la matrice M devient une matrice de tableaux d'entiers d'ordre $(S+1)\times(k+1)$. De plus, la fonction GetMsi est modifiée de sorte à utiliser le tableau de bocaux utilisés pour chaque case de la matrice lorsque nécessaire. Finalement, on remplace l'appel à GetMsi par un appel à GetA à la ligne 5 de l'algorithme initial. Cette nouvelle fonction reprend la logique de la fonction GetMsi initiale, sauf que l'on renvoie le tableau A correspondant au lieu du nombre total de bocaux utilisés. Voici le pseudocode des modifications effectuées :

   En termes de mémoire, chaque case $M[s][i]$, où $i > 0$ contient maintenant un tableau à i cases d'entier. Ainsi, on a $\frac{k\cdot(k+1)}{2}$ cases d'entier dans chaque ligne, i.e. il y a $(S+1)\cdot\frac{k\cdot(k+1)}{2}$ cases d'entier dans toute la matrice M. De ce fait, la complexité spatiale de cet algorithme est en $\Theta(S\cdot k^2)$.

b) Dans un deuxième temps, on souhaite éviter de copier dans chaque case du tableau M, les tableaux A des bocaux utilisés. Pour ce faire, on conserve l'algorithme initial et on reconstitue *a posteriori* le tableau A de la solution optimale. Voici le pseudocode du deuxième algorithme :

   Force est de constater que l'algorithme suit *dans le pire cas* un chemin de déplacements unitaires sans detours de la case (S, k) à la case $(0, 0)$. C'est

```
1: function GetMsi(V : k-integer array, M : (S + 1) × (k + 1) integer array matrix, s : integer, i : integer)
2:     A : integer array
3:     if s = 0 then
4:         return 0
5:     else if s < 0 then
6:         return +∞
7:     else if i = 0 then
8:         return +∞
9:     else
10:        A ← M[s][i]
11:        return ∑_{j=1}^{i} A[j]

12: function GetA(V : k-integer array, M : (S+1)×(k+1) integer array matrix, s : integer, i : integer)
13:    A : integer array filled with i zeros
14:    if s > 0 and i > 0 then
15:        left ← GetMsi(V, M, s, i − 1)
16:        up ← GetMsi(V, M, s − V[i], i) + 1
17:        if left < up then
18:            A[1 . . . i − 1] ← M[s][i − 1]
19:        else
20:            A[1 . . . i] ← M[s − V[i]][i]
21:            A[i] ← A[i] + 1
22:    return A
```

Algorithm 2 AlgoProgDynRet

Require: $k \geq 0$ **and** $S \geq 0$

```
1: function ConfitureBackwards(k : integer, V : k-integer array, S : integer, M : (S + 1) × (k + 1) integer matrix)
2:     A : integer array filled with k zeros
3:     s, i : integers
4:     s ← S
5:     i ← k
6:     while s ≠ 0 do
7:         if i > 0 and M[s][i] = M[s][i − 1] then    ▷ no more V[i]-capacity jars
8:             i ← i − 1
9:         else                                        ▷ M[s][i] = M[s − V[i]][i] + 1
10:            A[i] ← A[i] + 1
11:            s ← s − V[i]
12:    return A
```

pourquoi on effectue *au plus* $S + k$ tours de boucle. On remarque aussi que le corps de la boucle comporte uniquement des instructions élémentaires. La complexité temporelle de la boucle de l'algorithme-retour est ainsi en $\mathcal{O}(S + k)$.

Par ailleurs, l'initialisation du tableau A est en $\Theta(k)$. Ce tableau A est la seule addition en termes de mémoire par rapport à l'algorithme de base et il occupe autant de cases mémoire que le tableau V. De ces faits, les compléxités temporelle et spatiale de l'algorithme-retour sont respectivement en $\mathcal{O}(S+k)$ et $\Theta(S \cdot k)$.

Question 6

On constate que, pour l'algorithme global, ce sont les complexités de l'algorithme initial qui l'emportent, d'où celles du premier algorithme sont retenues, à savoir des complexités en $\Theta(S \cdot k)$. De ce fait, on peut dire que cet algorithme est polynomial.

3 Algorithme III : Cas particulier et algorithme glouton

Question 7

Voici le pseudo-code de l'algorithme `AlgoGlouton` :

Algorithm 3 AlgoGlouton

Require: $k \geq 1$ **and** $S \geq 0$

```
1:  function MAIN(k : integer, V : k-integer array, S : integer)
2:      A : integer array filled with k zeros
3:      s, i, n : integers
4:      s ← S
5:      i ← k
6:      n ← 0
7:      while s ≠ 0 do
8:          A[i] ← s ÷ V[i]
9:          s ← s mod V[i]
10:         n ← n + A[i]
11:         i ← i − 1
12:     return (n, A)
```

L'algorithme `AlgoGlouton` consiste en une boucle contenant seulement quatre instructions élémentaires. De plus, on effectue au plus k tours de boucle car, le cas échéant, pour la k-ième itération, on utilise des bocaux de capacité 1 dg et on peut ainsi verser toute la confiture restante dans des bocaux de cette capacité. De ces faits, la complexité temporelle de l'algorithme `AlgoGlouton` dans le pire des cas est en $\mathcal{O}(k)$.

Question 8

On dit qu'un système de capacités est **glouton-compatible** si, pour ce système de capacités, l'algorithme `AlgoGlouton` produit la solution optimale quelle que soit soit la quantité totale S. Montrons qu'il existe des systèmes de capacités qui ne sont pas glouton-compatibles.

Preuve

Il suffit de donner un exemple de système de capacités V tel qu'il y a au moins une quantité totale S pour laquelle l'algorithme `AlgoGlouton` ne renvoie pas la solution optimale.

Soit le système de capacités $V = [1, 4, 5]$. Soit $S = 28$ la quantité totale de confiture. L'algorithme `AlgoGlouton` produit la solution $(8, [3, 0, 5])$, alors que la solution optimale est $(6, [0, 2, 4])$, puisque 5 bocaux ne suffisent pas. Ceci est donc un contre-exemple cherché.

Question 9

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

Preuve

D'après les conditions du problème, on sait que les solutions g et o sont différentes. Donc, il existe *au moins* un indice $j \in \{1, \ldots, k\}$ tel que $o_j < g_j$, car sinon la solution g n'utiliserait toute la quantité de confiture. Par ailleurs, on sait que k est fini, d'où il existe un plus grand indice $j \in \{1, \ldots, k\}$ remplissant ladite condition.

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

Preuve

D'aprés la question précédente, on a $o_l \geq g_l$ pour tout l dans $\{j+1, \ldots, k\}$. De plus, puisque au cours de l'algorithme glouton on choisit le nombre maximum possible de bocaux de capacité $V[l]$, on a $o_l \leq g_l$. On en déduit que $o_l = g_l$. Donc, on a :

$$\sum_{i=1}^{k} V[i] \cdot g_i = \sum_{i=1}^{k} V[i] \cdot o_i \tag{7}$$

$$\sum_{i=j+1}^{k} V[i] \cdot g_i = \sum_{i=j+1}^{k} V[i] \cdot o_i \tag{8}$$

En soustrayant (8) de (7), on obtient $\sum_{i=1}^{j} V[i] \cdot g_i = \sum_{i=1}^{j} V[i] \cdot o_i$.

c) Montrons que $\sum_{i=1}^{j-1} V[i] \cdot o_i \geq V[j]$.

Preuve

On sait que

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

En utilisant le résultat de la question précédente, il en résulte que

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

D'où,

$$V[j] \cdot (g_j - o_j) \leq \sum_{i=1}^{j-1} V[i] \cdot o_i.$$

Or $g_j - o_j \geq 1$ d'après la question a) et le fait que ce sont des entiers naturels. Donc, $V[j] \leq \sum_{i=1}^{j-1} V[i] \cdot o_i$.

d) Supposons que $o_i \leq d-1$ pour tout $i \in \{1, \ldots, j-1\}$. D'après la question précédente, on a

$$\begin{aligned} V[j] &\leq \sum_{i=1}^{j-1} V[i] \cdot o_i \\ &\leq \sum_{i=1}^{j-1} V[i] \cdot (d-1) \\ &\leq (d-1) \cdot \sum_{i=1}^{j-1} 2^{i-1} \\ &\leq (d-1) \cdot \frac{d^{j-1}-1}{d-1} \\ d^{j-1} &\leq d^{j-1} - 1 \qquad \text{car } d-1 \geq 1 \end{aligned}$$

On arrive donc à une contradiction, i.e. il existe un indice $l \in \{1, \ldots, j-1\}$ tel que $o_l \geq d$.

e) Posons la suite o' telle que $o'_l = o_l - d, o'_{l+1} = o_{l+1} + 1$ et $o'_i = o_i$ pour tout $i \neq l$ et $i \neq l+1$. Montrons que o' est une solution du problème.

Preuve

D'après les conditions du problème, on a

$$V[l] \cdot d = V[l+1].$$

D'où,

$$0 = -V[l] \cdot d + V[l+1].$$

Donc,

$$V[l] \cdot o_l + V[l+1] \cdot o_{l+1} = V[l] \cdot (o_l - d) + V[l+1] \cdot (o_{l+1} + 1),$$

i.e.

$$V[l] \cdot o_l + V[l+1] \cdot o_{l+1} = V[l] \cdot o'_l + V[l+1] \cdot o'_{l+1}.$$

Par ailleurs, on a $o_i = o'_i$ pour tout $i \in \{1, \ldots, k\} \setminus \{l, l+1\}$.

De ces faits, $\sum_{i=1}^{k} V[i] \cdot o_i = \sum_{i=1}^{k} V[i] \cdot o'_i$, i.e. o' est aussi une solution du problème.

déduire que Expo est glouton-compatible

Question 10

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

Preuve

On a montré dans la question précédente que le système **Expo** est glouton-compatible.

Soit $V = [1, p]$, où $p > 1$, un système de capacités quelconque. On remarque que ce système est un système **Expo** avec $k = 2$ et $d = p \geq 2$.

De ces faits, tout système de capacité V avec $k = 2$ est bien glouton-compatible.

Deuxième partie

Mise en œuvre

4 Implémentation

Le but de ce second exercice est de mesurer de manière expérimentale la complexité de la fonction `tailleMaxRec` et de l'algorithme décrit dans la question 5 de l'exercice précédent. Le choix du langage de programmation est libre. La complexité expérimentale de l'exécution d'une fonction correspond ici au temps qu'il faut pour que la fonction s'exécute.

Dans cet exercice, n désigne la taille de la séquence initiale.

Question 3

La plus grande valeur de n que l'on peut traiter sans problème de mémoire ou de temps d'exécution de quelques minutes pour la fonction `tailleMaxRec` est 18.

Quant à `tailleMaxIter`, la valeur de n correspondante est 1600.

Question 4

On a mis dans le tableau ci-dessous le temps d'exécution de plusieurs appels à la fonction `tailleMaxRec` selon les valeurs de n.

n	$CRec(n)$ (en s.)	$\log CRec(n)$
11	0.0915572	-1.0383075
12	0.3087260	-0.5104268
13	0.9356748	-0.0288751
14	3.0208589	0.4801304
15	9.7992974	0.9911949
16	30.8693360	1.4895273
17	100.5011405	2.0021710
18	322.5870859	2.5086470

On en déduit la droite de régression de $\log CRec(n)$ en fonction de n :
$\Delta : y = 0.505452x - 6.592292$. On remarque que la pente de la droite est 0.505452, ce qui était attendu car la valeur est comprise entre $\log 2 = 0.301030$ et $\log 4 = 0.602060$.

Question 5

On a mis dans le tableau ci-dessous le temps d'exécution de plusieurs appels à la fonction `tailleMaxIter` selon les valeurs de n.

n	$CIter(n)$ (en s.)	$\frac{CIter(n)}{n^3}$
200	0.5192115	$6.490\,143\,4 \times 10^{-8}$
400	4.1114257	$6.424\,102\,6 \times 10^{-8}$
600	14.2562266	$6.600\,104\,9 \times 10^{-8}$
800	34.8566699	$6.807\,943\,3 \times 10^{-8}$
1000	71.8084174	$7.180\,841\,7 \times 10^{-8}$
1200	124.3441816	$7.195\,843\,8 \times 10^{-8}$
1400	200.7089246	$7.314\,465\,2 \times 10^{-8}$
1600	314.0727653	$7.667\,792\,1 \times 10^{-8}$

On en déduit la droite de régression de $\frac{CIter(n)}{n^3}$ en fonction de n : $\Delta : y = 8.842\,541 \times 10^{-12}x - 6.164\,326 \times 10^{-8}$. On remarque l'ordre de grandeur de la pente de la droite : $8.842\,541 \times 10^{-12} \ll 1$. Ceci implique que la fonction est bien constante de valeur $6.960\,154\,6 \times 10^{-8}$.

A Suites croissantes

On rappelle la définition des suites $b(s)$ et $c(s)$ comme suit :

$$b(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 & \text{si } s = 1 \\ 2 \cdot b(s-2) + 2 & \text{si } s \geq 2 \end{cases} \text{ et } c(s) = \begin{cases} 0 & \text{si } s = 0 \\ 2 \cdot c(s-1) + 2 & \text{si } s \geq 1 \end{cases}$$

Montrons que les deux suites sont croissantes.

Suite $b(s)$

Soit la propriété suivante $Q_1(s), s \geq 0 : b(s) \leq b(s+1)$. Montrons-la par récurrence d'ordre 2 sur s

Preuve

Base : On a $b(0) = 0 \leq 2 = b(1) \leq 2 \cdot 0 + 2 = b(2)$. Donc, la propriété est vérifiée aux rangs 0 et 1.

Induction : Supposons $Q_1(s-2)$ et $Q_1(s-1)$ vraies pour un $s \geq 2$ fixé. Montrons $Q_1(s)$ vraie.

Par hypothèse de récurrence, $b(s-2) \leq b(s-1)$. D'où, $2 \cdot b(s-2) + 2 \leq 2 \cdot b(s-1) + 2$. Or, par définition, $b(s+1) = 2 \cdot b(s-1) + 2$. Donc, $b(s) \leq b(s+1)$, i.e. la propriété est vérifiée au rang s.

Conclusion :

$$\left.\begin{array}{l} Q_1(0) \text{ et } Q_1(1) \text{ vraies} \\ \forall s \geq 2, [(Q_1(s-2) \text{ et } Q_1(s-1)) \Rightarrow Q_1(s)] \end{array}\right\} \forall s \in \mathbb{N}, b(s) \leq b(s+1)$$

On remarque que ceci correspond à la définition d'une suite croissante. ■

Suite $c(s)$

Soit la propriété suivante $Q_2(s), s \geq 0 : c(s) \geq 0$. Montrons-la par récurrence faible sur s.

Preuve

Base : Pour $s = 0$, on a $c(0) = 0 \geq 0$. Donc, la propriété est vérifié aux rang 0.

Induction : Supposons $Q_2(s)$ vraie pour un $s \geq 0$ fixé. Montrons $Q_2(s+1)$ vraie.

Par définition, $c(s+1) = 2 \cdot c(s) + 2$. Or, par hypothèse de récurrence, $c(s) \geq 0$. D'où, $2 \cdot c(s) + 2 \geq 2$. Donc, $c(s+1) \geq 2 \geq 0$. On en déduit par transitivité que la propriété est vérifiée au rang $s+1$.

Conclusion :

$$\left.\begin{array}{l} Q_2(0) \text{ vraie} \\ \forall s \geq 0, [Q_2(s) \Rightarrow Q_2(s+1)] \end{array}\right\} \ \forall s \in \mathbb{N}, c(s) \geq 0$$

Par ailleurs, on a $c(s) = 2 \cdot c(s-1) + 2, s \geq 1$. D'où, $c(s) \geq 2 \cdot c(s-1)$. Comme cette suite est positive, on sait que $2 \cdot c(s-1) \geq c(s)$. Donc, par transitivité, $c(s) \geq c(s-1)$. Autrement dit, pour tout $s \geq 0$, $c(s+1) \geq c(s)$. ■

B Complexité de `RechercheExhaustive` dans le cas général