Examen d'algorithmique

Mercredi 22 juin 2016 12h–14h30 / Aucun document autorisé

Mode d'emploi : Le barème est donné à titre indicatif. **La qualité de la rédaction des algorithmes et des preuves sera très fortement prise en compte pour la note.** On peut toujours supposer une question résolue et passer à la suite.

Exercice 1 : Dérouler et analyser (2 points)

On considère l'algorithme ci-dessous :

```
Def P(T,g,d) :
  Si g>=d  Alors Retourner 0
  Si g==d-1 Alors Retourner T[g]
  Si g==d-2 Alors Retourner T[g]+T[g+1]
  Sinon
    i1 = g+(d-g)/3
    i2 = g+(2*(d-g))/3
    x1 = P(T,g,i2)
    x2 = P(T,i1,d)
    Retourner x1+x2+T[i1]
```

0 1 2 3 4 5 6 7

1. Décrire ce que fait l'algorithme `P` appelé sur le tableau `[4,2,5,7,6,4,2,9]` avec `g` $= 0$ et `d` $= 8$. On détaillera tous les appels récursifs effectués et on donnera la valeur retournée. 146
2. Quelle est sa complexité pour un tableau de taille n (d'indices $0, \ldots, n-1$) avec $g = 0$ et $d = n$?

Exercice 2 : Plus longue sous-séquence stable (6 points)

Étant donné un tableau d'entiers `T` d'indices $0, \ldots, |\texttt{T}| - 1$, on cherche à calculer la taille du sous-tableau stable le plus long de `T`. Un sous-tableau stable est une suite continue d'indices du tableau contenant la même valeur. Par exemple, avec le tableau `T = [1,6,8,8,8,4,2,1,1,8,4,4]`, la valeur recherchée est 3 et elle correspond au sous-tableau d'indices $\{2, 3, 4\}$ de taille 3.

1. Proposer un algorithme Diviser-pour-régner pour résoudre ce problème. L'algorithme sera de la forme `algo-rec(T,`g, d`)` et renverra la valeur recherchée (la taille du sous-tableau stable le plus long) pour la partie de `T` entre les indices g et d.
   Appliquer votre algorithme sur `T = [1,6,6,8,4,2,2,2,1,8,4,4]`
   Donner sa complexité.

2. Proposer un algorithme de programmation dynamique pour résoudre ce problème. On pourra construire des tableaux `Nb[-]` et `Nbf[-]` tels que :
   — `Nb[i]` est la taille d'un sous-tableau stable le plus long dans la partie de `T` restreinte aux indices $0, \ldots, i$, et
   — `Nbf[i]` est la taille d'un sous-tableau stable le plus long dans la partie de `T` restreinte aux indices $0, \ldots, i$ et se terminant à l'indice i.

   Dans l'exemple `T = [1,6,8,8,8,4,2,1,1,8,4,4]`, la valeur de `Nb[8]` est 3, et `Nbf[8]` vaut 2.

   Appliquer votre algorithme sur `T = [1,6,6,8,4,2,2,2,1,8,4,4]`
   Donner sa complexité.
3. Reprendre l'algorithme de la question précédente en évitant l'utilisation des tableaux `Nb[-]` et `Nbf[-]`. Quelle est la complexité de cet algorithme ?

Exercice 3 : Calcul de notes - 3 points

On cherche à calculer les notes de semestre d'étudiants qui ont passé plusieurs unités d'enseignement (chacune d'elle reçoit un coefficient représentant un nombre de crédits). Pour valider un semestre, un étudiant doit valider 30 crédits. Il a pu suivre (et valider) un ensemble d'UEs qui dépassent les 30 crédits et pour calculer sa note finale, on cherche à trouver les UE validées qui lui permettent d'obtenir son semestre (30 crédits) avec la meilleure moyenne pondérée[1] possible. Pour cela, il est possible de diminuer le coefficient d'une UE si c'est nécessaire pour arriver à exactement 30 crédits (mais bien sûr, il n'est pas possible d'augmenter le crédit d'une note).

Par exemple, supposons que les notes sont les suivantes (avec le nombre de crédits entre parenthèses) : 12 (6), 17 (8), 11 (3), 15 (3), 14 (4), 8 (5), 13 (6), 18 (5). Il est alors préférable (pour l'étudiant) de prendre les notes 12 (6 dégradé en 4), 17 (8), 15 (3), 14 (4), 13 (6), 18 (5). Cela lui donne une moyenne de 15,1 (*ie* 453/30).

Proposer un algorithme glouton pour calculer la meilleure moyenne possible. Donner sa complexité. Justifier votre algorithme.

Exercice 4 : Partition équitable d'un ensemble de valeurs - 6 points

Étant donnée une liste de n valeurs entières, on veut les répartir dans deux sacs de manière la plus équilibrée possible (telle que les sommes des éléments de chaque sac soient les plus proches possibles). On va chercher un algorithme qui calcule la **valeur** (= la somme des éléments contenus dans un sac) des deux sacs. Par exemple, si $L = \{6, 1, 3, 1\}$, un sac contiendra les objets $\{6\}$ et l'autre $\{1, 3, 1\}$ et les valeurs des sacs seront donc 6 et 5.

1. On suppose que l'on dispose d'une fonction $\mathsf{Sac}(L, v)$ qui renvoie VRAI si et seulement si il est possible de faire, avec les objets de L, un sac dont la valeur vaut v. Par exemple, pour $L = \{2, 8, 3, 7\}$ et $v = 12$, c'est possible (avec le sac $\{2, 3, 7\}$), mais si on prend $v = 14$, ce n'est pas possible : on aura donc $\mathsf{Sac}(\{2, 8, 3, 7\}, 12) = \mathsf{Vrai}$ et $\mathsf{Sac}(\{2, 8, 3, 7\}, 14) = \mathsf{Faux}$.

1. par le poids en crédits

Proposer un algorithme qui, étant donnée L, retourne des **valeurs** pour deux sacs les plus équilibrés possibles. Par exemple, pour $L = \{6, 1, 3, 1\}$, l'algorithme retournera le couple $(6, 5)$ ou $(5, 6)$.
Donner sa complexité en supposant que la fonction `Sac(-,-)` prend un temps constant.

2. Appliquer votre algorithme sur $L = \{2, 5, 1, 3\}$, puis sur $L = \{4, 1, 3, 6\}$.
3. On cherche à présent à trouver un algorithme pour la procédure `Sac`(L, v). Pour cela, on va calculer (et stocker dans un tableau) les valeurs de `Sac`(L, v) pour $0 \leqslant v \leqslant V_{max}$ où V_{max} est un entier fixé.
   (a) Proposer un algorithme de programmation dynamique pour construire un tableau Booléen à deux dimensions $T[i, v]$ où $0 \leqslant i \leqslant n$ et $0 \leqslant v \leqslant V_{max}$ tel que $T[i, v]$ vaut VRAI si et seulement si il est possible de faire un sac de valeur v avec les i premiers objets L. Justifier l'algorithme et donner sa complexité.
   (b) Appliquer l'algorithme sur l'exemple $L = \{2, 6, 9, 3, 7, 4\}$ et $V_{max} = 20$.
4. En déduire, l'algorithme complet pour le calcul des valeurs des deux sacs équilibrés (celui de la question 1 avec la procédure pour `Sac`(L, v) de la question 3). Donner sa complexité.

Exercice 5 : (3 points)

1. Vous avez 12 pièces, toutes de même poids exceptée une qui est légèrement plus lourde que les autres. Trouver la pièce "lourde" en 3 pesées seulement. Même problème avec 27 pièces.
2. Généraliser votre algorithme pour n pièces (on pourra supposer que n est une puissance de 3) et exprimer le nombre de pesées nécessaires (on pourra utiliser le Master Theorem).
3. Deux extensions :
   (a) Que se passe-t-il si la pièce recherchée est plus lourde *ou* plus légère ?
   (b) Que se passe-t-il si il y a deux pièces plus lourdes ?

Annexe : Master theorem

Soient $a \geqslant 1$, $b > 1$, $f(n)$ une fonction positive et $t(n) = \begin{cases} a \cdot t(\frac{n}{b}) + f(n) & \text{si } n > 1 \\ \Theta(1) & \text{si } n = 1 \end{cases}$;

- Si $f(n) = O(n^{\log_b a - \epsilon})$ avec $\epsilon > 0$, alors $t(n) = \Theta(n^{\log_b a})$
- Si $f(n) = \Theta(n^{\log_b a})$, alors $t(n) = \Theta(n^{\log_b a} \cdot \log n)$
- Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour $\epsilon > 0$ et si $\exists c < 1$ tel que $a \cdot f(\frac{n}{b}) \leqslant c \cdot f(n)$ pour n assez grand, alors $t(n) = \Theta(f(n))$