université de BORDEAUX

Année universitaire 2017/2018
Session 1 d'automne CPBX AN2

Parcours / Étape : 4TCBX2 Code UE : 4TBX311
Épreuve : 4TBX311EX Informatique
Date : 23/01/2018 Heure : 11h30 Durée : 1h30
Aucun document autorisé
Épreuve de M. Dorbec Paul

Collège Sciences et Technologie

Le sujet est découpé en trois parties indépendantes. La première partie consiste en l'application des cas du cours de calcul de complexité. La deuxième partie, la plus longue, est un problème classique d'algorithmique, étudié sous différentes coutures. Il est possible de faire la partie 2.4 sans avoir fait la partie 2.3. Enfin, la troisième partie est un petit problème d'invariant.

Le barème, donné à titre indicatif, est sur 25 pts notamment parce que le sujet est long. Ne perdez pas de temps ; n'hésitez pas à avancer si une question vous fait perdre du temps pour y revenir plus tard.
Les notes de cours ne sont pas autorisées, l'énoncé du master théorème est rappelé à la fin du sujet.

---

1 Complexité et formules de récurrence (7pts)

[1] Imaginez que vous ayez à choisir entre les trois algorithmes suivants pour résoudre un problème de taille n :

- L'algorithme A résout le problème en le divisant en 5 sous problèmes de taille $\frac{n}{2}$, en les résolvant récursivement, puis en recombinant les solutions en temps linéaire.
- L'algorithme B résout le problème en résolvant récursivement deux problèmes de taille $n-1$ puis en combinant les solutions en temps constant.
- L'algorithme C résout le problème en le découpant en 9 sous-problèmes de taille $\frac{n}{3}$ qui sont résolus récursivement, puis en combinant les solutions avec un algorithme quadratique (en $O(n^2)$).

1. Pour chaque algorithme, vous donnerez explicitement la formule de récurrence satisfaite par la complexité de l'algorithme, et l'ordre de grandeur du temps d'exécution.
2. Quel algorithme a la meilleure complexité asymptotique ?

R :

On applique deux fois le master Theorem (pour A et C), et on fait explicitement la récurrence pour B.

- Pour l'algorithme A, on obtient les constantes $a = 5$, $b = 2$, et $d = 1$. On est dans le premier cas du théorème, on obtient une complexité en $O(n^{\log_2(5)}) \approx n^2.32$.
- Pour l'algorithme B, le théorème ne s'applique pas. On a $T(n) = 2T(n-1) + c$. En définissant $T'(n) = T(n) - c$, on a $T'(n) = 2T'(n-1)$. Donc le résultat est en $O(2^n)$, cet algo est pourri!
- Pour l'algorithme C, le master theorem s'applique avec $a = 9$, $b = 3$, $d = 2$. On est dans le cas $d = 2 = \log_3(9)$ et donc la complexité est en $O(n^2 \log(n))$.

Le meilleur choix est ici l'algo C.

---

1. Exercice inspiré de "Algorithms", de S. Dasgupta, C. Papadimitriou, U. Vazirani

2 Le problème du rendu de monnaie (15pts)

Le problème du rendu de monnaie est le suivant :

> Avec des pièces de valeurs $p_1 < p_2 < \ldots < p_k$, si je dois rendre un montant x en monnaie, quel est le plus petit nombre de pièces qu'il est nécessaire d'utiliser ?

Typiquement, en Europe, le jeu de pièces (et billets) contient les valeurs $1, 2, 5, 10, 20, \ldots$

3. Supposons que l'on veuille rendre 34 de monnaie avec un jeu de pièces $(1, 2, 5, 10, 20)$, quelles pièces choisiriez-vous ?

R : Je choisirais une pièce de 20, une pièce de 10, et deux pièces de 2.

2.1 Algorithme Glouton (4pts)

On considère que le jeu de pièces est décrit par les valeurs $p_1 < p_2 < \ldots < p_k$. On suppose aussi que tous les montants x sont réalisables avec les pièces de monnaie. Ce n'est pas toujours le cas [2], par exemple avec un jeu de pièces $[3, 6, 10]$, il n'est pas possible de réaliser le montant 17.

4. Justifiez qu'il faut et qu'il suffit que $p_1 = 1$ pour que toutes les valeurs soient accessibles.

R : Si 1 est parmi les valeurs, alors toute valeur x est réalisable par x pièces de 1. Si 1 n'est pas une valeur, un n'est pas réalisable.

Une méthode pour essayer de retourner de la monnaie avec un petit nombre de pièces est de choisir toujours les pièces de valeur la plus grande possible.

5. Décrivez par un algorithme cette méthode naturelle pour rendre une somme de monnaie x définie en paramètre.

R : Tant que $x > 0$, (soit p_i la piece la plus grande $\leq x$, ajouter la pièce p_i, $x = x - c_i$).

6. Quelle est l'ordre de grandeur de la complexité de l'algorithme en fonction de la valeur x en entrée dans le cas général ? Justifiez brièvement.

R : La complexité est en $O(x)$ car x décroit à chaque passage, ou plutôt $O(x + k)$ pour être précis!

Ce type d'algorithme est appelé algorithme glouton. Il n'est pas toujours optimal, comme nous allons le voir maintenant.

7. Si les valeurs des pièces sont $[1, 4, 6, 21, 30, 37]$, comment l'algorithme précédent rend-il 51 en monnaie ? Quel serait le choix optimal ?

R : L'algorithme précédent donne $37 + 6 + 6 + 1 + 1$. Il serait déjà possible d'optimiser avec $37 + 4 + 4$, mais la meilleure solution reste $30 + 21$.

2.2 Algorithme brute-force (2pts)

Voici une méthode pour calculer le nombre optimal de pièces. On remarque que la solution optimale utilise soit au moins une pièce de plus grande valeur, soit n'en utilise aucune. On obtient donc la propriété récursive suivante qui calcule le nombre optimal de pièces pour atteindre une somme x avec un jeu de pièces $(p_1 < p_2 < \ldots < p_k)$:

$$opt(x, [p_1, \ldots, p_k]) = \min \begin{pmatrix} 1 + opt(x - p_k, [p_1, \ldots, p_k]), \\ opt(x, [p_1, \ldots, p_{k-1}]) \end{pmatrix}$$

On en déduit l'algorithme suivant (décrit en Python) :

2. Le plus grand nombre qu'il est impossible d'atteindre avec un ensemble de pièces donnée est appelé nombre de Frobenius, ce n'est pas le problème ici ;-)

```
def opt(x, L):
    """ Version brute-force du calcul de l'optimum. """
    k = len(L)
    if x == 0 or k == 1:  # On admet que L[0] == 1
        return x
    else:
        sol2 = opt(x, L[0 : k - 1])
        if x - L[k - 1] >= 0:
            sol1 = 1 + opt(x - L[k - 1], L)
            if sol1 < sol2:
                return sol1
        return sol2
```

8. Qu'est ce qui garantit que cet algorithme s'arrête ? Répondez en donnant une propriété monotone précise, et son évolution à chaque appel de la fonction.

R : La valeur de $x + k$ décroît strictement à chaque appel.

2.3 Complexité de la méthode Brute-force (5pts)

On s'intéresse dans cette partie à la complexité de l'algorithme brute-force. Si vous perdez pied, passez à la section suivante.
On va approximer le pire cas de la complexité de l'approche brute-force par le cas où les pièces ont toutes la valeur 1 (c'est à dire $p_1 = p_2 = \ldots = p_k = 1$). On note c pour toute partie constante de la complexité, et donc $f(x, k)$ la complexité de l'appel `opt(x,L)` où L est une liste composée de k fois la valeur 1.

9. Quelle formule de récurrence est satisfaite par $f(x, k)$?

R : $$f(x, k) = f(x - 1, k) + f(x, k - 1) + c$$

10. Quels sont les cas de base de la récurrence ?

R : Si $x = 0$ ou $k = 1$, la complexité de l'algorithme est en $O(1)$

11. En remplaçant partout la valeur de c par 1, calculez les valeurs de $f(x, k)$ pour tous les x de 0 à 4 et k de 1 à 5 (présentés dans un petit tableau).

R :

$x \backslash k$	1	2	3	4	5
0	1	1	1	1	1
1	1	3	5	7	9
2	1	5	11	19	29
3	1	7	19	39	69
4	1	9	29	69	139

12. On note $f'(x, k) = f(x, k) - c$. Quelle est la récurrence satisfaite par cette fonction ? Cela vous rappelle-t-il quelque chose ?

R : $$f'(x, k) = f'(x - 1, k) + f'(x, k - 1)$$
ce qui n'est pas sans faire penser à la récurrence de la fonction binomiale!

13. On veut montrer que $f'(x, k)$ est au moins exponentielle en $x + k$. On définit

$$g(t) = \min_{x+k=t} (f'(x, k)) .$$

Peut-on donner une formule de récurrence pour minorer $g(t)$? Qu'en déduisez-vous ?

R : Si on pose $g(t)$ ainsi définie, alors on observe que $g(t) \geq 2g(t - 1)$. Par conséquent, $g(t)$ est minorée par 2^t.

2.4 Un algorithme malin (3pts)

Dans la méthode précédente, on se rend compte qu'on risque de faire plusieurs fois indépendamment le même appel. Par exemple, si les pièces sont $L = [\ldots, 13, 24, 27, 51]$, pour l'appel `opt(74,L)`, on fera deux fois indépendamment l'appel `opt(23,[...,13])` : une fois après avoir choisi une pièce de 51, et une fois après avoir choisi une pièce de 24 et une pièce de 27. C'est cela qui conduit à une complexité exponentielle de l'algorithme.
Voici une autre approche. Pour calculer la solution optimale pour x, on va calculer les solutions optimales pour toutes les valeurs inférieures, et les utiliser par la suite.

```
def algo2(x,pieces):
    """ Calcul du nombre de pièces optimal pour toutes les valeurs <= x """
    L = [0]
    for y in range(1, x + 1):
        if y in pieces:
            L.append(1)
        else:
            L.append(None)
            for p in pieces:
                if y - p >= 0 and ( L[y] is None or L[y] > L[y - p] + 1):
                    L[y] = L[y - p] + 1
    return L
```

14. Quelle est la complexité de cet algorithme ?

R : On lit assez vite que pour chaque valeur entre 1 et x, on fait au plus k tests, donc la complexité est $O(x + k)$

15. Que calcule la boucle `for p in pieces` ?

R : Elle calcule le minimum des $1 + x$ pour x l'optimum du nombre de pièces atteignant $y - p$.

16. Justifiez pourquoi, si `L` contient le nombre de pièces optimal pour les valeurs de 1 à $y - 1$ à l'entrée de la boucle `for y`, $L[y]$ contient la valeur optimale à la sortie de la boucle.

R : Pour réaliser une valeur x, il est nécessaire d'utiliser au moins une pièce parmi celles dans la liste. Le nombre de pièces optimal est donc nécessairement de la forme $1 + x'$ où x' est une valeur parmi la liste des montant $x - p$. Or au cours de l'algorithme, on a identifié la valeur optimale pour chacun de ces $x - p$, on calcule bien le minimum parmi ces valeurs.

3 Invariants (3pts)

Un sac contient initialement n boules noires et b boules blanches. Tant que le sac contient au moins 2 boules, on tire 2 boules du sac et

— on les jette si elles sont de la même couleur,

— on jette la noire et on remet la blanche dans le sac sinon.

Si le sac est vide, on remet une boule noire.

17. Justifiez que le processus s'arrête.
18. Quelle est, en fonction de n et b, la couleur de la boule dans le sac à la fin ?

---

4 Rappel de cours

Théorème 1 (Master Theorem) *Soit T qui satisfait la relation de récurrence suivante :*

$$T(n) = a\, T\left(\frac{n}{b}\right) + O(n^d)\,.$$

Alors :

— *Si $d < \log_b(a)$ alors $T(n) = O(n^{\log_b(a)})$.*

— *Si $d = \log_b(a)$ alors $T(n) = O(n^d \log(n))$.*

— *Si $d > \log_b(a)$ alors $T(n) = O(n^d)$.*