Contrôle Continu 1

Yann Rotella, yann.rotella@uvsq.fr

13 mars 2021

Tous les documents sont autorisés, la durée de l'examen est de deux heures. La qualité de la rédaction et la précision des raisonnements seront pris en compte dans la note de chaque réponse.

1 Analyse d'algorithmes

Exponentiation Rapide (11 points)

Dans cet exercice, on souhaite calculer x^a, étant donné $x \in \mathbb{N}^*$ et $a \in \mathbb{N}$. Pour ce faire on utilise l'algorithme suivant. On rappelle que l'on représente nos entiers dans potentiellement plusieurs registres dans l'ordinateur.

Algorithm 1 exp(x,a)

```
r ← 1
e ← a
y ← x
while e > 0 do
    if e mod 2 = 1 then
        r ← r * y
    end if
    y ← y * y
    e ← e/2
end while
return r
```

1. (1 point) Quelle est la mémoire nécessaire pour stocker l'entrée et la sortie de l'algorithme ?
2. (2 points) Prouver la terminaison de l'algorithme.
3. (6 points) Prouver l'exactitude de l'algorithme [indice : utiliser une variable muette i].
4. (2 points) Quelle est la complexité de cet algorithme en nombre de multiplications (O, Ω et Θ) ?

Complexité (16 points)

1. (2 points) Soit $f : n \mapsto n^n$ et $g : n \mapsto n!$. Est-ce que $f \in O(g)$? Est-ce que $g \in O(f)$? Montrer le résultat.
2. (1 point) Quel est le coût de l'addition (ou de la soustraction) de deux entiers de taille n ?
3. (2 points) Le coût d'une multiplication de deux entiers est plus important qu'une addition. Cela se remarque en particulier lorsque l'on manipule des grands entiers. Reprenez comment vous posez (à la main) une multiplication de deux grands entiers x et y. Quelle est la complexité de cette multiplication naïve en fonction de x et de y ?

4. (2 points) Au lieu de faire une multiplication naïve, on peut "diviser pour mieux régner". Pour n'importe quels entiers x, y et m, on peut écrire $x = x_1 2^m + x_0$ et $y = y_1 2^m + y_0$. Ainsi

$$xy = z_2 2^{2m} + z_1 2^m + z_0$$

avec $z_2 = x_1 y_1$, $z_1 = x_1 y_0 + x_0 y_1$, $z_0 = x_0 y_0$. Écrire la relation de récurrence sur la fonction de coût d'un algorithme qui utiliserait cette astuce pour multiplier deux grands entiers de même taille (n).

5. **(Karatsuba)** (2 points) Montrer qu'il est possible de calculer z_2, z_1 et z_0 en seulement 3 multiplications. Montrer alors que

$$T(n) \leq 3T(\lceil n/2 \rceil) + cn + d$$

où c et d sont deux constantes et n désigne la taille des entiers à multiplier.

6. (5 points) Montrer que la multiplication qui utilise cette astuce se fait en $\Theta(n^{\log(3)})$.
7. (2 points) En supposant que la multiplication se fait en $\Theta(n^{\log(3)})$. On fixe l'exposant a. Donner le représentant de la classe de complexité de l'algorithme 1.

2 Programmation

Le tri par tas (13 points)

Nous avons vus en cours que nous pouvons "imaginer" un arbre binaire dans une liste si, à l'élément en position i, on considère que les éléments aux positions $2i+1$ et $2i+2$ sont respectivement les fils gauches et droits du parent à la position i. Nous faisons exactement cela dans cet exercice.

1. (1 point) Dessiner l'arbre binaire induit par la liste suivante en Python.

$$[12, 5, 9, 2, 0, 4, 21, 13, 16]$$

2. (1,5 points) Écrire les fonctions `filsDroit`, `filsGauche` et `Parent` qui prennent en entrée la liste, un indice et renvoient l'indice du fils droit, du fils gauche ou du parent s'ils existent, et -1 sinon.
3. (1 point) Écrire une fonction `fils` qui prend en entrée la liste et un indice et renvoie la liste des fils du noeud à la position i.
4. (1,5 points) On appelle **tas** tout arbre binaire dans lequel toute étiquette d'un noeud est inférieur ou égal aux étiquettes de ses fils. Écrire une fonction `estUnTas` qui prend en entrée une liste et renvoie `True` si la liste représente un tas et `False` sinon.
5. (2 points) Le principe du tri par tas est de transformer la liste en tas afin de faire remonter le plus petit élément du tableau. Pour cela, on fait une procédure récursive et en particulier nous utilisons la fonction `tasRec` qui prend en entrée la liste et un indice i, et permute s'il faut l'élément à la position i, avec le plus petit de ses fils, et appelle récursivement au noeud qui vient d'être modifié s'il y a eu modification.
6. (3 points) Écrire une procédure `transformerEnTas` qui, en partant du fond de l'arbre, fait appel à la fonction précédente plusieurs fois et permet de transformer la liste donnée en entrée en un tas et renvoie cette liste, afin de faire remonter les plus petits éléments en "haut" de l'arbre.
7. (3 points) Écrire une fonction `triTas` qui fait appel plusieurs fois à `transformerEnTas` et permet de trier n'importe quelle liste donnée en entrée.