UE LU3IN003 - Algorithmique.
Licence d'informatique.

Partiel du 10 novembre 2023. Durée : 1h30

Documents non autorisés. Seule une feuille A4 portant sur les cours est autorisée.
Tous objets connectés éteints et rangés dans vos sacs.

Exercice 1 (3 points)

Question 1 (1/3) — Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ? Justifier la réponse.

Oui car un algorithme en $O(n)$ est en $O(n^2)$. Exemple d'un tel algorithme : la recherche d'un élément dans un tableau.

Question 2 (1/3) — Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ? Justifier la réponse.

Oui. Par exemple, la complexité du tri par insertion, comptée en nombre de comparaisons, est en $\Theta(n^2)$ dans le pire cas mais elle est en $O(n)$ lorsque les données sont déjà triées (dans le bon ordre).

Question 3 (1/3) — Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'elle soit en $O(n)$ sur *toutes* les données ? Justifier la réponse.

Non. Par l'absurde : si la complexité d'un algorithme est en $O(n)$ sur toutes les données alors la complexité $f(n)$ dans le pire des cas est aussi en $O(n)$, ce qui contredit le fait que $f(n)$ est en $\Theta(n^2)$.

Exercice 2 (5 points)

Une grenouille se situe sur la rive d'un étang, devant une rangée de n nénuphars, indicés de 1 à n (du nénuphar le plus proche au plus éloigné de la rive). La rive correspond à l'indice 0 par convention. La grenouille se déplace systématiquement vers l'avant, en sautant de nénuphar en nénuphar pour aller de la rive au dernier nénuphar de la rangée, autrement dit le nénuphar n. Chaque saut est de longueur 1 (la grenouille atterrit sur le nénuphar $i+1$ depuis le nénuphar i) ou 2 (la grenouille atterrit sur le nénuphar $i+2$ depuis le nénuphar i). Plus précisément, la grenouille est susceptible de réaliser cinq types de sauts depuis un nénuphar i :

A : saut simple vers le nénuphar $i+1$,
B : saut périlleux vers le nénuphar $i+1$,
C : saut simple vers le nénuphar $i+2$,
D : saut périlleux vers le nénuphar $i+2$,
E : double saut périlleux vers le nénuphar $i+2$.

Un saut simple est un saut sans figure, un (double) saut périlleux un saut où la grenouille réalise une (double) roulade en l'air. Une séquence de sauts est donc caractérisée par une suite de caractères sur l'alphabet $\{A, B, C, D, E\}$. Par exemple, pour $n = 4$, si la grenouille réalise un double saut périlleux de la rive au nénuphar 2, puis un saut simple de 2 à 3, et enfin un saut périlleux de 3 à 4, cela correspond à la séquence $[E, A, B]$.

L'objet de cet exercice est de concevoir et d'analyser une fonction récursive qui énumère toutes les séquences de sauts possibles pour aller de la rive au nénuphar n. Cette fonction retourne une liste de listes, chaque liste correspondant à une séquence de sauts possibles.

Question 1 (1/5) — Compléter le pseudo-code ci-dessous de manière à ce qu'il permette d'énumérer toutes les séquence de sauts possibles pour aller de la rive au nénuphar n.

Grenouille(n)

Entrée : $n \in \mathbb{N}^*$
Sortie : liste des séquences de sauts possibles pour aller de la rive au nénuphar n
si $n=1$ **alors**
    retourner $[[A], [B]]$
si $n=2$ **alors**
    retourner $[[A, A], [B, B], [A, B], [B, A], [C], [D], [E]]$
sinon
    $P \leftarrow$ Grenouille($n-1$)
    $Q \leftarrow$ Grenouille($n-2$)
    retourner $[[A]+p : p \in P]+[[B]+p : p \in P]+[[C]+q : q \in Q]+[[D]+q : q \in Q]+[[E]+q : q \in Q]$

Question 2 (1.5/5) — Quel est l'ordre de grandeur en $O(.)$ du nombre de nœuds dans l'arbre des appels récursifs de Grenouille(n) ? Justifier la réponse.

Le nombre de nœuds est caractérisé par l'équation de récurrence :

$$A(1) = 1,\ A(2) = 1,\ A(n) = 1 + A(n-1) + A(n-2) \text{ pour } n \geq 3$$

Le polynôme caractéristique de $A'(n) = A'(n-1) + A'(n-2)$ est $r^2 - r - 1$, dont la racine positive est $(1 + \sqrt{5})/2 \simeq 1.618$. On en déduit que le nombre de nœuds est en $O(1.618^n)$.

Question 3 (1.5/5) — Quel est l'ordre de grandeur en $O(.)$ de la longueur des listes P et Q ? Justifier la réponse.

Le nombre u_n de séquences pour n nénuphars est caractérisé par l'équation de récurrence :

$$u_1 = 2,\ u_2 = 7,\ u_n = 2u_{n-1} + 3u_{n-2} \text{ pour } n \geq 3$$

Le polynôme caractéristique est $r^2 - 2r - 3$, dont la racine positive est $(2 + \sqrt{16})/2 = 3$. On en déduit que la longueur des listes P et Q est en $O(3^n)$.

Question 4 (1/5) — En déduire la complexité de Grenouille(n) en fonction de n.

En réalisant le produit du nombre de nœuds par la complexité de chaque appel, on obtient $O((3 \times 1.618)^n) \equiv O(4.854^n)$.

Exercice 3 (6 points)

Dans cet exercice, on conçoit et on analyse un algorithme pour convertir un nombre entier naturel x représenté en base décimale (base 10) en sa représentation en base binaire (base 2). Par exemple, le nombre entier $x = 42$ (en base 10) est converti en 101010_2, où l'indice 2 sert à signifier que la représentation est en base binaire. Dans tout l'exercice, on suppose que l'on dispose d'un tableau `bin[0...9]`, où `bin[`c`]` contient la représentation binaire du digit (chiffre) c.

c	0	1	2	3	4	5	6	7	8	9
`bin[`c`]`	0_2	1_2	10_2	11_2	100_2	101_2	110_2	111_2	1000_2	1001_2

On suppose également disposer d'une fonction `mult2`(x, y) réalisant efficacement le produit de deux nombres entiers binaires x et y (algorithme de Karatsuba). En s'appuyant sur `mult2`, on va concevoir une méthode diviser pour régner pour la conversion en base 2. Dans la suite, on fait l'hypothèse simplificatrice que n est une puissance de 2 (l'approche peut toutefois se généraliser à n'importe quel n sans détériorer la complexité).

Question 1 (1/6) — Compléter la fonction `pui2bin` ci-dessous, qui permet de convertir un nombre entier décimal 10^n (un 1 suivi de n zéros) en binaire, où n est une puissance de 2.

```
fonction pui2bin(n)
    si n = 1 alors retourner 1010_2
    sinon :
        z = | pui2bin(n/2) |
        retourner mult2(z, z)
```

Question 2 (1.5/6) — Soit $T(n)$ la complexité de `pui2bin`(n). Donner l'équation de récurrence satisfaite par $T(n)$. En déduire la complexité de `pui2bin`(n).

Indication : La représentation en base 2 d'un entier à n digits (chiffres 0 à 9) comporte $\Theta(n)$ bits, et `mult2`(x, y) est de complexité $\Theta(n^d)$ pour multiplier deux entiers binaires x et y à n bits, où $d = \log_2 3 (\simeq 1.58)$.

La fonction `pui2bin`(n) réalise un appel récursif `pui2bin`$(n/2)$ puis un appel à la fonction `mult2` sur deux entiers binaires comportant $\Theta(n/2) \equiv \Theta(n)$ bits. On a donc :

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n^{\log_2 3})$$

On applique le théorème maître avec $a = 1$, $b = 2$ et $d = \log_2 3$. Comme $a = 1 < 3 = b^d$, on en déduit une complexité $\Theta(n^{\log_2 3})$.

Question 3 (1.5/6) — L'algorithme `dec2bin2`(x) ci-dessous vise à convertir n'importe quel entier décimal x à n digits (où n est une puissance de 2) en binaire. Compléter l'algorithme.

```
fonction dec2bin2(x)
   si n = 1 alors retourner bin[x]
   sinon :
       partitionner x en x_G = x_1 ... x_{n/2} et x_D = x_{n/2+1} ... x_n
       retourner  mult2(dec2bin2(x_G),pui2bin(n/2))+dec2bin2(x_D)
```

Question 4 (2/6) — Soit $T(n)$ la complexité de `dec2bin2`(x) pour un entier décimal x à n bits. Donner l'équation de récurrence satisfaite par $T(n)$. En déduire la complexité de `dec2bin2`(x).

Les deux appels récursifs `dec2bin2`(x_G) et `dec2bin2`(x_D) retournent en $T(n/2)$ un nombre binaire comportant $\Theta(n/2) \equiv \Theta(n)$ bits.
L'appel `pui2bin`$(n/2)$ retourne en $\Theta(n^{\log_2 3})$ (d'après la question 6) un nombre binaire avec $\Theta(n)$ bits.
Les deux arguments de `mult2` comportent donc $\Theta(n)$ bits, et l'appel a une complexité $\Theta(n^{\log_2 3})$.
Enfin, l'addition de deux nombres binaires comportant $\Theta(n)$ bits est en $\Theta(n)$.
L'équation de récurrence est donc :

$$T(n) = 2T\left(\tfrac{n}{2}\right) + \Theta(n^{\log_2 3})$$

Théorème maître avec $a=2$, $b=2$ et $d=\log_2 3$. Comme $a=2<3=b^d$, la complexité est $\Theta(n^{\log_2 3})$.

Exercice 4 (7 points)

Un graphe orienté $G(S, A)$ est dit **sans détour** s'il comporte une racine et si pour tout couple (u, v) de sommets dans S, il existe au plus un chemin élémentaire de u à v. Dans cet exercice, on cherche à concevoir un algorithme pour vérifier si un graphe est sans détour.

Question 1 (2/7) — On cherche tout d'abord à déterminer l'existence ou non d'une racine dans G.
a) Proposer une méthode faisant appel à un ou plusieurs parcours en profondeur pour déterminer si un graphe G comporte une racine. On ne demande pas de justification.
b) Quelle est la complexité de cette méthode ? Justifier brièvement la réponse.

Précision : Réponse d'autant mieux notée que la complexité de la méthode proposée sera faible.

a) Deux méthodes possibles :

1. On réalise un parcours en profondeur de G en partant d'un sommet choisi arbitrairement. Si le seul point de régénération est le sommet de départ du parcours, alors ce sommet est racine de G. Dans le cas contraire, on réalise un parcours en profondeur à partir du dernier point de régénération, noté s. Si ce parcours permet de visiter tous les sommets de G, alors s est racine de G, sinon pas de racine.
2. On réalise un parcours en profondeur à partir de chacun des sommets de G. Si l'on trouve un sommet à partir duquel on parvient à visiter tous les autres sommets sans recourir à un (ou plusieurs) point(s) de régénération, alors c'est une racine. Si l'on n'en trouve pas, alors G ne comporte pas de racine.

b) Avec la méthode 1, on réalise au plus deux parcours en profondeur, ce qui peut être fait en $O(n+m)$ si G est représenté par des listes de successeurs. Avec la méthode 2, on réalise au plus n parcours en profondeur, pour une complexité $O(n(n+m))$ si G est représenté par des listes de successeurs.

On suppose désormais que G comporte une racine et on note r celle-ci. On considère un parcours en profondeur L de G depuis r. On rappelle que les arcs du graphe n'appartenant pas à la forêt $F(L)$ associée au parcours (constituée ici d'une unique arborescence) peuvent être partitionnés en : arcs avant, arrière ou transverse [1].

Question 2 (1/7) — Le graphe G est-il sans détour si l'on détecte au terme du parcours L :
a) un arc (u, v) avant ? Justifier la réponse.
b) un arc (u, v) transverse ? Justifier la réponse.

Soit C le chemin de r à u dans $F(L)$, et C' le chemin de r à v.
a) La concaténation de C et (u, v) constitue un chemin élémentaire de r à v. Le chemin C' en est un autre. Donc G n'est pas sans détour.
b) Idem.

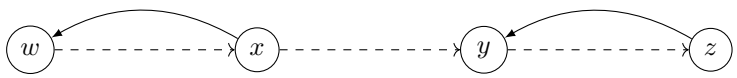
Question 3 (1.5/7) — On suppose dans cette question que G ne comporte ni arc avant ni arc transverse au terme du parcours L.

Remarquons tout d'abord que si l'on détecte un *unique* arc arrière au terme du parcours, alors il est clair que G est sans détour (G est une arborescence avec un arc supplémentaire qui crée un circuit unique).

Supposons maintenant que l'on détecte (au moins) deux arcs arrière le long d'un même chemin de l'arborescence $F(L)$, arcs arrière que l'on suppose *sans extrémité commune* dans cette question. Indiquer les trois agencements possibles des deux arcs arrière en les dessinant sur les graphes ci-dessous, où les pointillés entre deux sommets figurent une portion de chemin dans $F(L)$ (pas nécessairement un unique arc). Dans chaque cas, on indiquera si cela permet de conclure que G est *avec* détour, et si oui on indiquera le couple u, v de sommets qui en témoigne et les deux chemins élémentaires distincts de u à v.

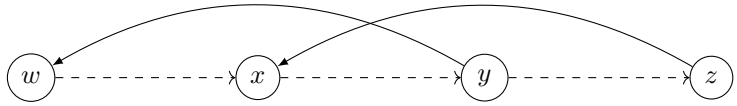
Précision : Pour la lisibilité du dessin, on tracera des arcs arrière incurvés.

a)



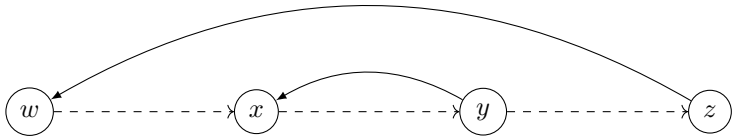
Cela ne permet pas de conclure que G est avec détour.

b)



Il existe deux chemins élémentaires de y à x : $(y, w, \ldots, x)$ et $(y, \ldots, z, x)$. Donc G est avec détour.

c)



Il existe deux chemins élémentaires de y à x : (y, x) et $(y, \ldots, z, w, \ldots, x)$. Donc G est avec détour.

Question 4 (1/7) — En s'appuyant sur les numéros de prévisite et de postvisite des sommets dans le parcours en profondeur, il est possible de caractériser les paires a, b d'arcs arrière qui conduisent à

1. Un arc est ici dit *transverse* s'il n'est ni dans $F(L)$, ni avant, ni arrière.

conclure que le graphe est *avec* détour. Notons $a = (a^-, a^+)$ et $b = (b^-, b^+)$ une paire d'arcs arrière au terme du parcours. En vous aidant de la réponse à la question précédente, compléter la formule ci-dessous qui donne cette caractérisation (pas de justification demandée) :

$$\text{pre}\left(\boxed{}\right) < \text{pre}\left(\boxed{}\right) < \text{post}\left(\boxed{}\right) < \text{post}\left(\boxed{}\right)$$

$$\text{et pre}\left(\boxed{}\right) < \text{pre}\left(\boxed{}\right) < \text{post}\left(\boxed{}\right) < \text{post}\left(\boxed{}\right)$$

Question 5 (1.5/7) — Conclure en décrivant la méthode complète pour déterminer si un graphe orienté G est sans détour, et en donnant sa complexité (justifier la complexité).

Précision importante : On admettra dans la réponse que détecter l'existence de deux arcs arrières vérifiant la condition de la question 4 peut se faire sans "surcoût" de complexité.

On suppose que G est représenté par des listes de successeurs. D'après la question 1, on peut déterminer si G comporte une racine en $O(n+m)$ ou en $O(n(n+m))$ (selon la méthode proposée). Si G ne comporte pas de racine alors il n'est pas sans détour. Sinon on fait un parcours en profondeur depuis la racine trouvée. Si l'on détecte un arc avant ou transverse, alors G n'est pas sans détour. Détecter un tel arc peut se faire en cours de parcours, sans surcoût de complexité : si un successeur v du sommet examiné courant u est déjà visité, et l'appel récursif sur v terminé. S'il n'y a pas d'arc avant ou transverse, alors il est admis dans l'énoncé de la question que détecter deux arcs arrière vérifiant la condition de la question 4 peut se faire également sans surcoût de complexité. S'il n'y en a pas, alors G est sans détour.
Déterminer si un graphe G orienté est sans détour se fait donc soit en trois parcours en profondeur au plus si l'on utilise la méthode 1 pour déterminer une racine, pour une complexité $O(n+m)$, soit en $n+1$ parcours en profondeur au plus si l'on utilise la méthode 2, pour une complexité en $O(n(n+m))$.