Université de Nice-Sophia Antipolis — POLYTECH
CiP2 — 2011–2012

Examen de Algorithmique et Programmation

Durée : 1h30 minutes
Aucun document autorisé
Mobiles interdits *Note : la qualité des commentaires, avec notamment la présence d'affirmations significatives, ainsi que les noms donnés aux variables, l'emploi à bon escient des majuscules et la bonne indentation rentreront pour une part importante dans l'appréciation du travail.*

▶ 1. Écrivez, de façon *récursive* et en JAVA, la fonction *nbSommants* qui retourne le nombre de décompositions d'un entier naturel p en *au plus* q sommants. Un sommant est un naturel positif qui entre dans une somme quand on décompose un nombre en somme de naturels. Ainsi, les décompositions de 5 en au plus 3 sommants sont $5, 4+1, 3+2, 3+1+1, 2+2+1$, $nbSommants(5,3) = 5$.
Indications :
- cherchez d'abord ce que doit renvoyer la fonction quand $p = 0$, puis quand $q = 0$;
- quand $q > p$, on considère que $nbSommants(p,q) = nbSommants(p,p)$;
- enfin dans le cas général, quand $p \geqslant q$ le nombre de décompositions de p en au plus q sommants est le nombre de décompositions de $p - q$ en au plus q sommants plus le nombre de décompositions de p en au plus $q - 1$ sommants.

Question sur 3 pts

```
static int nbSommants(int p, int q) {
    if (p == 0) return 1;
    else
      if (q == 0) return 0;
      else
         if (q > p) return nbSommants(p,p);
         else return  nbSommants(p-q,q) + nbSommants(p,q-1)
}
```

.....

▶ 2. Écrivez la suite d'appels récursifs pour l'appel `nbSommants(3,2)`. Quel est le résultat de cet appel.

Question sur 2 pts

```
nbSommants(3,2)
   nbSommants(1,2)
     nbSommants(1,1)
        nbSommants(0,1)
        nbSommants(1,0)
```

```
nbSommants(3,1)
   nbSommants(2,1)
      nbSommants(1,1)
         nbSommants(0,1)
         nbSommants(1,0)
      nbSommants(2,0)
   nbSommants(3,0)
```

Le résultat de l'appel est : 2 (3, et 2+1)

.....

▶ 3. À l'aide des fonctions de base de manipulation d'un arbre binaire, écrivez en JAVA la fonction `max`, dans la classe `ArbreBinaireChaîné<T>`, qui renvoie la valeur maximale d'un arbre binaire. Vous appliquerez une approche de parcours en profondeur afin d'explorer tous les noeuds de l'arbre afin de lire la valeur de chaque noeud et ne renvoyer que la valeur la plus grande parmi toutes celles rencontrées. On considérera que le type générique `T` implémente `Comparable` et donc muni de la fonction `compareTo` afin de pouvoir comparer les valeurs de l'arbre. Si vous le souhaitez, mais ce n'est pas une obligation, vous pourrez considérer l'existence d'une fonction `minDeT()` qui retourne la valeur minimale du type générique `T`. Votre fonction émettra une exception si l'arbre courant est vide.

Question sur 4 pts

```
/* Rôle : retourne le max de l'arbre binaire a
 */
private T max(ArbreBinaire<T> a) throws ArbreVideException {
    if (a.estVide()) return null; // le min de T
    else {
        T maxSag =  max(a.sag());
        T maxSad =  max(a.sad());
        if (maxSag == null && maxSad == null)
            // a est une feuille
            return a.valeur();
        // sinon il existe au moins un sous-arbre
        if (maxSag == null) return maxSad;
        if (maxSad == null) return maxSag;
        // a possède 2 sous-arbres
        return (((Comparable) maxSag).compareTo(maxSad) > 0) ?
            maxSag : maxSad;
    }
}
/** Rôle : retourne le max de l'arbre binaire courant
 *
 * @return <code>T</code>
 */
public T max() throws ArbreVideException {
    if (estVide()) throw new ArbreVideException();
    return max(this) ;
}
```

.....

On définit un *arbre quaternaire ordonné*, le type abstrait qui permet de ranger des valeurs appartenant à un espace à deux dimensions où chaque dimension est munie d'une relation d'ordre. Chaque nœud est le père de 4 sous-arbres qui représentent une partition de l'espace à 2 dimensions auquel appartient la valeur du nœud en 4 quadrants. Si nous appelons les deux dimensions respectivement **latitude** et **longitude**, nous pourrons appeler les 4 quadrants, respectivement, sud-ouest (**SO**), nord-ouest (**NO**), sud-est (**SE**) et nord-est (**NE**).

Ainsi, étant donnée une valeur particulière qui occupe un nœud de l'arbre quaternaire, son sous-arbre sud-est ne comprend que des valeurs dont la latitude est inférieure ou égale à la sienne et la longitude strictement supérieure à la sienne. Remarquez la dissymétrie des relations, nécessaire pour que l'on puisse placer sans hésitation des points distincts mais qui ont une coordonnée en commun.

$$Arbre4 = \emptyset \mid < Valeur, Arbre4, Arbre4, Arbre4, Arbre4 >$$

où Valeur est le couple de coordonnées *(longitude, latitude)* de type T.

▶ 4. Définissez l'axiomatique de la fonction *ajouter* :

$$\text{ajouter} : arbre4 \times T \times T \ \rightarrow \ arbre4$$

qui ajoute dans l'arbre une valeur représentée par ses deux coordonnées, longitude et latitude, de type T.

Question sur 1 pt

Définition axiomatique de la fonction *ajouter* :

$$\text{ajouter} : arbre4 \times T \times T \ \rightarrow \ arbre4$$

$\text{ajouter}(<>, l, L) =< l, L, \emptyset, \emptyset, \emptyset, \emptyset >$
$\text{ajouter}(< x, y, SE, SO, NE, NO >, l, L) \qquad (1)$
$l \leqslant x, L > y, \qquad (1) =< x, y, \text{ajouter}(SE, l, L), SO, NE, NO >$
$l \leqslant x, L \leqslant y, \qquad (1) =< x, y, SE, \text{ajouter}(SO, l, L), NE, NO >$
$l > x, L > y, \qquad (1) =< x, y, SE, SO, \text{ajouter}(NE, l, L), NO >$
$l < x, L \leqslant y, \qquad (1) =< x, y, SE, SO, NE, \text{ajouter}(NO, l, L) >$

.....

▶ 5. Écrivez les déclarations JAVA de l'interface générique `Arbre4<T>` et la classe `Arbre4Chaîné<T>` pour représenter un arbre quaternaire (constantes, variables et constructeurs). Puis, programmez la fonction `ajouter`.

Question sur 2 pts

```
public interface Arbre4<T> {
    public Arbre4<T> ajouter(T latitude, T longitude);
    public boolean estVide();
}
public class Arbre4Chaîné<T> implements Arbre4<T> {
    public static final Arbre4 arbre4Vide = new Arbre4Chaîné();
    protected T x, y;
    protected Arbre4<T> se, so, ne, no;
```

```
    // uniquement pour construire la constante arbre4Vide
    private Arbre4Chaîné() {
        se = so = ne = no = null;
    }
    /**
     * Rôle : créer une feuille de l'arbre
     *         étiquetée par x et y
     */
    public Arbre4Chaîné(T x, T y) {
        this.x = x;
        this.y = y;
        this.se = this.so =
        this.ne = this.no = (Arbre4Chaîné<T>) arbre4Vide;
    }
    /**
     * Rôle : retourne true si l'arbre courant est vide
     *         et false sinon
     */
    public boolean estVide() {
        return this == arbre4Vide;
    }
   /*
    * Rôle : ajouter la latitude et la longitude dans l'Arbre4 courant
    */
    public Arbre4<T> ajouter(T latitude, T longitude) {
        Arbre4<T> a;
        if (estVide())
            a = new Arbre4Chaîné<T>(latitude, longitude);
        else
            if (((Comparable) latitude).compareTo(x)<=0)
                // sud
                if (((Comparable) longitude).compareTo(y)>0)
                    // se
                    a = se.ajouter(latitude,longitude);
                else
                    // so
                    a = so.ajouter(latitude,longitude);
            else
                // nord
                if (((Comparable) longitude).compareTo(y)>0)
                    // ne
                    a = ne.ajouter(latitude,longitude);
                else
                    // no
                    a = no.ajouter(latitude,longitude);
        //
        return a;
    }
}
```

.....

▶ 6. Une liste linéaire est représentée par une structure simplement chaînée implémentée par la classe générique JAVA `ListeChaînée<T>`. Dans cette classe, écrivez la méthode `retirer` qui, étant donnée une valeur de type `T` en paramètre, retire

de la liste linéaire courante, les éléments dont la valeur est égale à cette valeur passée en paramètre. Vous utiliserez de la méthode `equals` pour tester si la valeur d'un élément de la liste est identique à la valeur passée en paramètre de la méthode `retirer`. Exemple : soit l une `ListeChainée` d'`Integer`,

```
l.insérer(1,10);
l.insérer(2,13);
l.insérer(3,10);
// écrire la liste sur la sortie standard
System.out.println(l);
l.retirer(10):
// écrire la liste l, qui ne contient plus que 13
System.out.println(l);
```

Question sur 4 pts

```
public void retirer(T val) {
    // traiter les éléments de tête
    Noeud<T> p=tête;
    while (p!=null && (p.valeur()).equals(val)) {
        p = p.noeudSuivant();
        lg--;
    }
    tête = p;
    // p == null ou l'élément de tête est différent de val
    if (p==null) return;
    // sinon parcourir le reste de la liste
    while (p.noeudSuivant()!=null)
        if (p.noeudSuivant().valeur().equals(val)) {
            // supression du noeud courant de valeur val
            p.noeudSuivant(p.noeudSuivant().noeudSuivant());
            lg--;
        }
        else
            // pas de suppression, on avance dans la liste
            p = p.noeudSuivant();
}
```

.....

▶ 7. Expliquez le principe l'algorithme du tri rapide (quicksort), et précisez l'influence du choix du pivot sur la complexité du tri. Vous donnerez les complexités possibles de ce tri.

Question sur 3 pts

Le tri rapide est un tri par échangees et *partitions*. Il consiste à choisir une clé particulière dans la liste à trier, appelée *pivot*, qui divise la liste en deux sous-listes. Tous les éléments de la première sous-liste de clé supérieure au pivot sont transférés dans la seconde. De même, tous les éléments de la seconde sous-liste de clé inférieure au pivot sont transférés dans la première. La liste est alors formée de deux partitions dont les éléments de la première possèdent des clés inférieures ou égales au pivot, et ceux de la seconde possèdent des clés supérieures ou égales

au pivot. Le tri se poursuit selon le même algorithme sur les deux partitions si celles-ci possèdent une longueur supérieure à un.

Le choix du pivot conditionne fortement les performances du tri rapide. En effet, la complexité moyenne du nombre de comparaisons dans la phase de partitionnement d'une liste de longueur n est $\mathcal{O}(n)$, puisqu'on compare le pivot aux $n-1$ autres valeurs de la liste. Il y a n ou $n+1$ comparaisons. Si le choix du pivot est tel qu'il divise systématiquement la liste en deux partitions de même taille, c'est-à-dire que le pivot correspond à la médiane, le nombre de comparaisons sera égal à $n \log_2 n$. En revanche, si à chaque étape, le choix du pivot divise la liste en deux partitions de longueur 1 et $n-1$, les performances du tri chutent de façon catastrophique et le nombre de comparaisons est $\mathcal{O}(n^2)$. Dans le cas moyen, on a démontré, sous l'hypothèse de clés différentes et équiprobables, que le nombre de comparaisons est $2n\ln(n) \approx 1,38n\log_2 n$; sa complexité reste égale à $\mathcal{O}(n \log_2 n)$.

.....

▶ 8. Écrivez en JAVA la fonction booléenne `sommeDeux` de complexité $\mathcal{O}(n \log n)$ qui prend en paramètre un tableau `t` de `n` entiers et un entier `s`, et renvoie *vrai* s'il existe deux entiers e_1 et e_2 de t tels que $e_1 + e_2 = s$, et *faux* sinon. Vous considérerez que vous vous avez à votre disposition la méthode `sort` du paquetage `java.util.Arrays` qui trie un tableau d'entiers en $\mathcal{O}(n \log n)$.

Question sur 2 pts

```
public static boolean sommeDeux(int [] t, int s){
    java.util.Arrays.sort(t);
    for (int i = 0; i < t.length - 1; i++)
           if (rechercheDichoDeuxième(t, s, t[i], i+1, t.length-1))
                return true;
    // pas d'éléments dont la somme est égale à s
    return false;
}

public static boolean rechercheDichoDeuxième(int t[], int s,
                              int premier, int posInit, int posFin)
{
  if (posInit <= posFin) {
    if ((premier + t[(posFin+posInit)/2]) == s) return true;
    if (t[posInit] + (t[(posFin+posInit)/2]) < s)
      return rechercheDichoDeuxième(t,s,premier,(posFin+posInit)/2+1,posFin);
    else
      return rechercheDichoDeuxième(t,s,premier,posInit,(posFin+posInit)/2-1);
 //
 return false;
}
```

.....