

# Corrigé du contrôle écrit d'informatique INF 311

Ecole Polytechnique, Promotion 2006

Andreas Enge et Robert Cori

9 juillet 2007, durée 2 heures

Les exercices qui suivent sont indépendants et peuvent être traités dans n'importe quel ordre. Les correcteurs attacheront une grande importance à la clarté et à la concision de la rédaction.

## Exercice 1 (Barème envisagé : 3 pts)

**Question 1.** Expliquez en quelques mots ce que la commande `java Test` affiche une fois le programme suivant compilé.

```
class Test {
    public static void afficher (int[] tab, int i) {
        if (i < tab.length) {
            System.out.print (tab [i] + "_");
            afficher (tab, i+1);
        }
        else System.out.println();
    }
}

public static int somme (int[] tab, int i) {
    if (i < tab.length)
        return somme (tab, i+1);
    else
        return 0;
}

public static void ajouterUn (int[] tab) {
    int[] u = tab;
    for (int i = 0; i < u.length; i++)
        u [i] = u [i] + 1;
}

public static void main (String[] args) {
    int[] t = {1, 2, 3, 4, 5};
    afficher (t, 1);
    ajouterUn (t);
    afficher (t, 1);
    System.out.println (somme (t, 1));
}
}
```

**Solution.** La fonction `afficher` affiche le contenu du tableau d'entiers `tab` à partir de l'indice `i`. Étant appelée avec 1 à la place de `i`, elle omet en effet le premier élément (d'indice 0), et affiche au premier appel les entiers de 2 à 5, chacun sur une nouvelle ligne.

`ajouterUn` modifie le tableau passé en argument en ajoutant 1 à chaque élément du tableau ; par conséquent, le second appel à `afficher` affiche les entiers de 3 à 6.

La fonction `somme` renvoie toujours 0 (c'est le cas terminal de la récursion, et on remarque que malgré son nom, la fonction n'effectue jamais d'addition), de sorte que finalement, une ligne avec un 0 est affichée.

## Exercice 2 (Barème envisagé : 5 pts)

L'installation électrique d'une maison est protégée par un tableau de « fusibles », censés interrompre le courant en cas de court-circuit. Ainsi il faut faire attention à ne pas brancher trop d'appareils électriques sur le même fusible, pour éviter qu'il ne se déclenche quand tous les appareils seront mis en marche. Afin d'empêcher les pannes électriques dans l'usine, on vous demande d'écrire un programme informatique qui gère le tableau de fusibles.

**Question 2.** Écrivez une classe permettant de définir des objets `Fusible` contenant la capacité (un entier représentant la puissance maximum en Watt qui peut être branchée) et la puissance actuellement branchée sur le fusible (un autre entier). Munissez la classe d'un constructeur `Fusible (int c)` qui crée un fusible de capacité `c`, et de puissance branchée 0 (il n'y a pas d'appareil branché). Dans la suite de cet exercice, tout le code sera écrit à l'intérieur de cette classe.

**Solution.**

```
class Fusible {
    int capacite;
    int puissance;

    Fusible (int c) {
        this.capacite = c;
        this.puissance = 0;
    }
}
```

**Question 3.** Nous supposons que le nombre de fusibles est donné par la variable de classe `nbFus`, initialisée à 6. Ajoutez cette variable à la classe. Le tableau de fusibles est naturellement modélisé par un tableau (au sens de la programmation) de `Fusible`, créé à l'intérieur de la fonction `main`. Écrivez cette fonction et initialisez le tableau avec des fusibles de capacité 2200 W.

**Solution.**

```
final static int nbFus = 6;

public static void main (String[] args) {
    Fusible[] tab = new Fusible [nbFus];
    for (int i = 0; i < nbFus; i++)
        tab [i] = new Fusible (2200);
}
```

```
}
```

**Question 4.** On souhaite brancher un nouvel appareil. Écrivez une fonction

```
static Fusible chercherLibre (int puiss, Fusible[] tab)
```

qui renvoie **null** si tous les fusibles sont saturés et sinon le premier fusible dans `tab` pouvant encore supporter le branchement d'un appareil de puissance `puiss`.

**Solution.**

```
static Fusible chercherLibre (int puiss, Fusible[] tab) {
    for (int i = 0; i < nbFus; i++)
        if (tab [i].capacite - tab [i].puissance >= puiss)
            return tab [i];
    return null;
}
```

**Question 5.** Pour mieux équilibrer la charge, on préfère brancher le nouvel appareil sur un des fusibles ayant la plus grande disponibilité (différence entre capacité et puissance branchée). Écrivez une fonction

```
static Fusible chercherMaxLibre (int puiss, Fusible[] tab)
```

qui donne pour résultat le premier fusible correspondant ou **null**.

**Solution.**

```
static Fusible chercherMaxLibre (int puiss, Fusible[] tab) {
    // chercher le fusible avec la plus grande capacite restante
    Fusible maxFus = null;
    int maxCap = -1;
    for (int i = 0; i < nbFus; i++) {
        int cap = tab [i].capacite - tab [i].puissance;
        if (cap > maxCap) {
            maxFus = tab [i];
            maxCap = cap;
        }
    }
    // verifier si la capacite restante suffit
    if (maxCap >= puiss)
        return maxFus;
    else
        return null;
}
```

### Exercice 3 (Barème envisagé : 12 pts)

Pour mieux maîtriser la consommation de vos appareils électriques, vous avez décidé de les gérer par un programme informatique. Un appareil est caractérisé par son nom (une chaîne de caractères telle que "four", "télé", "ordinateur" etc.), sa consommation en Watt (un entier) et un booléen qui indique si l'appareil est allumé ou éteint.

**Question 6.** Écrivez la classe `Appareil` regroupant cette information et un constructeur `Appareil (String s, int p)` permettant de créer un appareil éteint de nom `s` et de consommation `p`.

**Solution.**

```
class Appareil {  
  
    String nom;  
    int consommation;  
    boolean allume;  
  
    Appareil (String s, int p) {  
        this.nom = s;  
        this.consommmation = p;  
        this.allume = false;  
    }  
}
```

**Question 7.** Un ensemble d'appareils est convenablement modélisé par une liste chaînée. En utilisant les listes vues en cours (ou dans le polycopié) pour lesquelles la liste vide est représentée par `null`, écrivez la classe `ListeA` correspondante avec son constructeur. Ajoutez une fonction `main` qui crée une nouvelle liste avec deux appareils dont vous pourrez choisir librement les caractéristiques.

**Solution.**

```
class ListeA {  
  
    Appareil contenu;  
    ListeA suivant;  
  
    ListeA (Appareil c, ListeA s) {  
        this.contenu = c;  
        this.suivant = s;  
    }  
  
    public static void main (String[] args) {  
        ListeA u = new ListeA(new Appareil ("radiateur", 2000), null);  
        u = new ListeA (new Appareil ("lave-vaisselle", 600), u);  
    }  
}
```

**Question 8.** Écrivez une fonction *itérative* `static void allumer (String s, ListeA u)` qui cherche l'appareil de nom `s` dans la liste `u` et l'allume s'il est présent.

**Solution.**

```
static void allumer (String s, ListeA u) {
    while (u!= null && !s.equals(u.contenu.nom))
        u = u.suivant;
    if (u != null)
        u.contenu.allume = true;
}
```

**Question 9.** C'est le jour d'enlèvement des déchets encombrants, et vous décidez de vous débarrasser de vos vieilles machines. Écrivez une fonction *réursive*

```
static ListeA enlever (String s, ListeA u)
```

qui enlève l'appareil de nom *s* de la liste s'il y est présent et qui renvoie la liste modifiée (vous pouvez supposer que les appareils ont tous des noms différents, et modifier la liste *u*).

**Solution.**

```
static ListeA enlever (String s, ListeA u) {
    if (u == null)
        return null;
    else if (u.contenu.nom.equals (s))
        return u.suivant;
    else {
        u.suivant = enlever (s, u.suivant);
        return u;
    }
}
```

**Question 10.** Vous voulez connaître les gros consommateurs parmi vos appareils. Écrivez une fonction

```
static ListeA grosConsommateurs (int puiss, ListeA u)
```

qui renvoie une nouvelle liste contenant les éléments de *u* (allumés ou éteints) consommant au moins *puiss*, la liste initiale devra rester intacte.

**Solution.**

```
static ListeA grosConsommateurs (int puiss, ListeA u) {
    if (u == null)
        return u;
    else if (u.contenu.consommation >= puiss)
        return new ListeA(u.contenu, grosConsommateurs(puiss, u.suivant));
    else
        return grosConsommateurs(puiss, u.suivant);
}
```

**Question 11.** En vue de brancher vos appareils, vous voulez les regrouper de sorte que la consommation cumulée dans chaque groupe n'exède pas la capacité d'un fusible. Évidemment, vous voulez regrouper les appareils pour tenter de minimiser le nombre de fusibles utilisés. Pour

commencer, écrivez une fonction

```
static ListeA[] extraireUnGroupe (int capacite, ListeA u)
```

qui partage la liste u en deux listes contenues dans un tableau. La première liste de ce tableau devra contenir des appareils qui seront branchés sur un premier fusible de la capacité donnée, la deuxième liste contiendra tous les autres appareils de sorte qu'aucun appareil de cette deuxième liste ne puisse être branché en plus sur le premier fusible (tout en satisfaisant la contrainte de capacité).

L'algorithme suivant, dit « glouton », permet de résoudre ce problème : parcourir la liste élément par élément ; si la consommation de l'élément est suffisamment basse pour tenir sur le fusible, l'ajouter à la première liste ; sinon l'ajouter à la seconde. Vous êtes libre pour cette question de détruire ou non la liste initiale et de créer ou non de nouvelles cellules.

**Solution.**

```
static ListeA[] extraireUnGroupe (int capacite, ListeA u) {
    ListeA[] res = {null, null};
    while (u != null) {
        if (u.contenu.consommation <= capacite) {
            res [0] = new ListeA (u.contenu, res [0]);
            capacite = capacite - u.contenu.consommation;
        }
        else
            res [1] = new ListeA (u.contenu, res [1]);
        u = u.suivant;
    }
    return res;
}
```

**Question 12.** La fonction de la question précédente crée un seul groupe d'appareils à brancher sur un même fusible. En vous servant de cette fonction, et en supposant que tous les fusibles ont la même capacité, écrivez une fonction

```
static void regrouper (int capacite, ListeA u)
```

qui affiche successivement les listes correspondant à la partition des appareils de u en groupes à brancher sur un même fusible. Vous utiliserez la méthode

```
static void afficher(ListeA u)
```

qu'il ne sera pas nécessaire d'écrire.

**Solution.**

```
static void regrouper (int capacite, ListeA u) {
    while (u != null) {
        ListeA[] groupes = extraireUnGroupe (capacite, u);
        afficher (groupes [0]);
        u = groupes [1];
    }
}
```

**Question 13.** Expliquer succinctement comment modifier cette fonction de façon qu'elle renvoie l'ensemble de groupes d'appareils plutôt que d'afficher ces groupes.

**Solution.** Il y a plusieurs solutions, l'une étant de renvoyer une liste de listes, il faudrait écrire cette classe. Une autre est de faire une première passe qui compte le nombre de fusibles nécessaires; puis une deuxième qui commence par construire un tableau dont les éléments de type `ListeA` de taille égale au résultat donné par la première, et qui le remplit ensuite.

**Question 14.** On considère 4 appareils de consommations respectives 2000, 2000, 3000, 3000. Proposez une valeur de capacité de fusible qui montre que l'algorithme glouton ne donne pas toujours le nombre minimum de fusibles permettant de brancher tous les appareils.

**Solution.** Il suffit de prendre la valeur 5000, l'algorithme glouton branche les deux premiers appareils sur un seul fusible et chacun des deux derniers sur un autre, alors qu'on peut n'utiliser que 2 fusibles.

**Question 15.** Pour améliorer le résultat on propose un autre algorithme que celui de la Question 11, il procède aussi par étapes en construisant des groupes d'appareils à brancher sur un même fusible. Toutefois à chaque étape il calcule le groupe dont la somme des consommations s'approche le plus de la capacité commune à tous les fusibles. On vous demande dans cette question de programmer une partie du calcul d'une étape, en utilisant un algorithme de recherche exhaustive. Plus précisément on vous demande d'écrire une fonction :

**static** `ListeA plusProche(Appareil[] tab, int capacite)` dont les données sont un tableau des  $n$  appareils à brancher et la capacité des fusibles, le résultat doit être une liste d'appareils dont la somme des consommations est inférieure à la capacité et la plus proche possible de celle-ci.

*Indications :* La méthode consiste à parcourir tous les sous-ensembles dont la somme des consommations est inférieure à la capacité; il vous faudra maintenir, tout au long de ce parcours, le meilleur sous-ensemble déjà calculé dans une variable de classe **static** `ListeA meilleure`. Pour cela vous écrirez une fonction récursive :

```
static void sousEnsAmeliore(Appareil[] tab, int i,
                           ListeA enCours, int capacite)
```

qui tente de compléter la liste `enCours`, contenant des appareils d'indices inférieurs à  $i$ , en ajoutant certains appareils parmi `tab[i] ... tab[n-1]`. La liste `meilleure` sera ainsi remplacée par `enCours` chaque fois qu'elle est plus intéressante.

**Solution.** On écrit deux fonctions qui seront utiles par la suite, l'une calcule la consommation d'une liste d'appareils et l'autre recopie une liste d'appareils :

```
static ListeA meilleure;

static int consommationL(ListeA u){
    if (u == null) return 0;
    else return u.contenu.consommation + consommationL(u.suivant);
}
static ListeA recopie(ListeA u){
    if (u == null) return null;
    else return new ListeA(u.contenu, recopie(u.suivant));
}
static ListeA plusProche(Appareil[] tab, int capacite){
    meilleure = null;
```

```

    ListeA enCours = null;
    sousEnsAmeliore(tab, 0, enCours, capacite);
    return meilleure;
}

static void sousEnsAmeliore(Appareil[] tab, int i,
    ListeA enCours, int capacite){
    int c = consommationL(enCours);
    if (i == tab.length) {
        if (c > consommationL(meilleure))
            meilleure = recopie(enCours);
    }
    else {
        if (c + tab[i].consommation <= capacite){
            enCours = new ListeA(tab[i], enCours);
            sousEnsAmeliore(tab, i+1, enCours, capacite);
            enCours = enCours.suivant;
        }
        sousEnsAmeliore(tab, i+1, enCours, capacite);
    }
}

```

**Question 16.** Donner un exemple qui montre que l'algorithme décrit à la question précédente ne donne pas non plus le nombre minimum de fusibles.

**Solution.** On considère par exemple six appareils de consommations respectives 3000, 3000, 3000, 5000, 5000, 5000 et on choisit 9000 pour la capacité des fusibles. Quel que soit l'ordre dans lequel sont considérés les appareils les trois premiers seront connectés sur un seul fusible et chacun des trois derniers sur un autre. On a ainsi au total 4 fusibles, alors que l'on pourrait faire avec 3 fusibles seulement.