

Contrôle d'informatique INF 311

Promotion 2005

Sujet proposé par Étienne Duris, Laurent Mauborgne et François Morain

11 juillet 2006

Version 2006/07/12

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

Exercice 1

Question 1. Qu'est-ce que le programme suivant affiche ?

```
public static void absolu(int x) {
    if(x<0) x = -x;
    return;
}

public static void f(int n) {
    int k = n;
    absolu(k);
    if(k>0) f(n-1);
    System.out.println(k);
}

public static void main(String [] args) {
    f(0);
    f(5);
    f(-5);
}
```

Corrigé. La fonction `absolu` n'a aucun effet.

0
0
1
2
3
4
5
-5

Exercice 2 — la chasse aux spams

Comme tout le monde, vous recevez beaucoup de courriers électroniques de publicité (spams). Vous avez décidé d'écrire un programme de gestion des adresses des ordinateurs qui vous envoient ces courriers, de façon à éliminer automatiquement ces spams.

Les deux parties de cet exercice sont indépendantes et peuvent être traitées dans l'ordre de votre choix.

Partie I — La liste noire

Vous allez d'abord mémoriser un ensemble d'adresses d'ordinateurs qui transmettent des spams. Cet ensemble sera appelé la liste noire. L'appartenance à la liste noire n'est pas forcément définitive. En effet, les autorités de régulation d'internet peuvent négocier avec le propriétaire de l'ordinateur fautif et le faire renoncer à l'émission de spam. On associe donc aux adresses d'ordinateur des dates de validité. Quand la date de validité est périmée, cela signifie que l'ordinateur peut ne plus diffuser de spam, et il faut donc le retirer de la liste noire.

Les dates (date courante, dates de validité, ...) seront représentées par des entiers correspondant aux nombres de secondes écoulées depuis une date 0 (le premier janvier 1970 à 0h00). Les ordinateurs seront identifiés par leur adresse IP, représentée par une `String`.

Question 2. Écrivez une classe `IPDate` permettant de stocker une adresse IP (sous la forme d'une `String`) et sa date de validité (sous la forme d'un `int`). Cette classe doit contenir un constructeur `IPDate(String s, int d)`.

Corrigé.

```
public class IPDate {
    String ip;
    int date;

    public IPDate(String s, int n) {
        this.ip = s;
        this.date = n;
    }
}
```

Dans la suite de cette partie, tout le code demandé sera écrit dans cette classe `IPDate`.

On choisit de limiter la taille de notre liste noire et de la stocker sous la forme d'un tableau, chaque élément du tableau contenant l'adresse IP d'un ordinateur et une date limite de validité. Ce tableau sera de taille constante `TAILLELN`, qu'on pourra fixer à 10000. On appellera ce tableau `listeNoire`.

Question 3. Écrivez la déclaration et l'initialisation des variables de classe TAILLELN et listeNoire.

Corrigé.

```
public final static int TAILLELN = 10000;
public static IPDate[] listeNoire = new IPDate[TAILLELN];
```

À tout instant, les cases du tableau listeNoire contiennent soit **null**, soit une IPDate. On dira qu'une case est *vide* si elle contient **null**. On dira qu'une case est *valide à la date d* si elle contient une adresse IP de date de validité supérieure ou égale à *d*.

Question 4.

a) Écrivez une fonction

```
public static boolean estDansListeNoire(String ip)
    qui recherche une adresse IP dans le tableau listeNoire. Elle retourne true si ip est
    dans une IPDate contenue dans une case du tableau et false sinon.
```

b) Donnez la complexité en nombre de comparaisons d'adresses IP de estDansListeNoire en fonction de TAILLELN dans le cas le pire.

Corrigé. Il faut parcourir tout le tableau jusqu'à trouver l'adresse IP. La complexité sera donc de l'ordre de la taille du tableau.

```
public static boolean estDansListeNoire(String ip) {
    for(int i=0; i<TAILLELN; i++)
        if(listeNoire[i]!= null && ip.equals(listeNoire[i].ip))
            return true;
    return false;
}
```

Question 5. Écrivez une fonction

```
public static boolean ajouterEnNull(IPDate ipd)
```

qui cherche la position de plus petit indice de listeNoire qui ne contienne rien (c'est-à-dire **null**) et insère ipd à cette place. La fonction retourne **false** si le tableau listeNoire ne contenait pas de case vide et **true** sinon.

Corrigé.

```
public static boolean ajouterEnNull(IPDate ipd) {
    for(int i=0; i<TAILLELN; i++) {
        if(listeNoire[i] == null) {
            listeNoire[i] = ipd;
            return true;
        }
    }
    return false;
}
```

0	1	2	3	...
"189.67.230.1",12	"125.34.32.189",7	null	"134.4.56.21",4	...

FIG. 1 – Un exemple d'une partie de `listeNoire` qui montre les cases 0 à 3

Question 6. Pour faire de la place, on peut régulièrement rechercher les informations qui ne sont plus valides, c'est-à-dire dont la date de validité est strictement inférieure à la date courante, et les remplacer par **null**.

Écrivez une fonction

```
public static void actualiser(int d)
```

qui remplace toutes les cases de date de validité strictement inférieure à `d` par **null**.

Corrigé.

```
public static void actualiser(int d) {
    for(int i=0; i<TAILLELN; i++) {
        if(listeNoire[i] != null && listeNoire[i].date < d)
            listeNoire[i] = null;
    }
}
```

Une autre façon de procéder est de considérer au moment de l'ajout que les adresses IP dont la date de validité est dépassée sont équivalentes à des **null** dans `listeNoire`. Sur l'exemple de la figure 1, à la date 5, la première case libre serait la case 2, mais à la date 10 la première case libre serait la case 1.

Question 7. Écrivez une fonction

```
public static boolean ajouterEnLibre(int d, IPDate ipd)
```

qui cherche la première position dans `listeNoire` qui ne contienne pas une adresse de date de validité supérieure ou égale à `d` (c'est le cas si cette case contient **null**) et insère `ipd` à cette place. La fonction retourne **false** si le tableau `listeNoire` était entièrement rempli d'adresses IP de date de validité supérieure ou égale à `d`. Elle retourne **true** sinon.

Corrigé.

```
public static boolean ajouterEnLibre(int d, IPDate ipd) {
    for(int i=0; i<TAILLELN; i++) {
        if(listeNoire[i] == null || listeNoire[i].date < d) {
            listeNoire[i] = ipd;
            return true;
        }
    }
    return false;
}
```

Question 8. Dans le cas où le tableau est entièrement rempli de cases valides à la date où on cherche à insérer une nouvelle adresse IP, il faudrait nécessairement accepter de perdre une information valide. On va alors choisir de perdre l'information qui deviendra invalide la première.

Écrivez une fonction

```
public static void remplacer(IPDate ipd)
```

qui suppose que `listeNoire` est entièrement rempli de cases valides. Si la date de validité de `ipd` est inférieure ou égale à toutes les dates de validité des adresses IP de `listeNoire`, cette fonction ne fait rien. Sinon, elle remplace la case de `listeNoire` contenant la plus petite date de validité par `ipd`. Dans le cas où plusieurs cases du tableau contiennent la plus petite date de validité, elle remplace par `ipd` celle de plus petit indice.

Corrigé.

```
public static void remplacer(IPDate ipd) {  
    int dateMin = ipd.date;  
    int indiceDateMin = -1;  
    for(int i = 0; i<TAILLELN; i++) {  
        if(listeNoire[i].date < dateMin) {  
            indiceDateMin = i;  
            dateMin = listeNoire[i].date;  
        }  
    }  
    if(indiceDateMin >= 0)  
        listeNoire[indiceDateMin] = ipd;  
}
```

Question 9. Écrivez une fonction

```
public static void ajouter(int d, IPDate ipd)
```

qui prend en argument la date courante `d` et une information `ipd` à stocker dans la liste noire. Si `listeNoire` contient une case vide ou une adresse IP de date de validité dépassée, elle insère `ipd` à cette place. Sinon, si la date de validité de `ipd` est strictement supérieure à au moins une date de validité d'une case de `listeNoire`, `ajouter` remplace la case contenant la date de validité la plus petite par `ipd`. Dans chacun des cas, si plusieurs places vérifient les critères d'insertion, `ipd` est ajouté dans celle de plus petit indice.

Corrigé. On peut utiliser un appel à `ajouterEnLibre` et, si le résultat est **false**, appeler `remplacer`. On peut aussi ne faire qu'un parcours de tableau :

```
public static void ajouter(int d, IPDate ipd) {  
    int dateMin = ipd.date;  
    int indiceDateMin = -1;  
    for(int i = 0; i<TAILLELN; i++) {  
        if(listeNoire[i] == null || listeNoire[i].date < d) {  
            listeNoire[i] = ipd;  
            return;  
        }  
    }  
}
```

```

        if(listeNoire[i].date < dateMin) {
            indiceDateMin = i;
            dateMin = listeNoire[i].date;
        }
    }
    if(indiceDateMin >= 0)
        listeNoire[indiceDateMin] = ipd;
}

```

Pour être plus efficace dans nos recherches, nous choisissons désormais de maintenir le tableau `listeNoire` trié par ordre croissant d'adresses IP. Si le tableau n'est pas complètement rempli, toutes les cases contenant des informations seront rassemblées au début du tableau (toutes les cases vides viennent après, à la fin du tableau). Formellement, cela revient à dire que pour tous les indices i et j tels que $0 \leq i < j < \text{TAILLELN}$:

- si `listeNoire[i]` est **null**, alors `listeNoire[j]` aussi;
- si `listeNoire[j]` est une `IPDate` d'adresse IP a_j , alors `listeNoire[i]` est aussi une `IPDate`, dont l'adresse IP a_i est telle que a_i arrive avant a_j dans l'ordre alphabétique.

Lorsqu'il vérifie ces propriétés, on dira que `listeNoire` est *compacté*.

Question 10. On suppose qu'on a accès à une fonction

```
public static void trierNonNulles(int imax)
```

qui suppose que les cases de `listeNoire` du début du tableau jusqu'à `imax` (inclus) ne sont pas vides, et qui les trie par ordre d'adresse IP croissante. Écrivez une fonction

```
public static void compacterEtTrier()
```

qui compacte et trie le tableau `listeNoire`.

Corrigé.

```

public static void compacterEtTrier(){
    // on cherche le premier null
    int i, j;
    for(i = 0; i < TAILLELN; i++)
        if(listeNoire[i] == null)
            break;
    j = i; // si i < TAILLELN, c'est la premiere case contenant null
    for(++i; i < TAILLELN; i++)
        // listeNoire[j] est la prochaine case a occuper
        // on a toujours i > j
        if(listeNoire[i] != null){
            listeNoire[j] = listeNoire[i];
            listeNoire[i] = null;
            j++;
        }
    // on trie
    trier(j-1);
}

```

Question 11. Écrivez une nouvelle fonction de recherche d'une adresse IP dans le tableau `listeNoire` utilisant le fait que le tableau est trié

```
public static boolean estDansListeNoireTriee(String ip)
```

Cette fonction devra avoir une complexité dans le cas le pire de l'ordre du logarithme de la taille de `listeNoire` (c'est-à-dire `TAILLELN`). *Pour alléger l'écriture, vous pourrez utiliser les symboles de comparaison usuels (<, >, ≤ et ≥) au lieu de la méthode `compareTo` qui permet de comparer deux `String` selon l'ordre alphabétique en Java.*

Corrigé. L'algorithme optimal procède bien sûr par dichotomie. La seule originalité est la présence de cases `null` en fin de tableau, mais il suffit de les traiter comme toujours supérieures à la valeur recherchée. La complexité est donc de l'ordre du logarithme de la taille du tableau.

```
public static boolean dichotomie(int g, int d, String ip) {
    if(g==d) {
        IPDate ipd = listeNoire[g];
        return (ipd != null) && ip.equals(ipd.ip);
    }
    int m = (g+d)/2;
    IPDate ipdm = listeNoire[m];
    if((ipdm == null) || (ip.compareTo(ipdm.ip)<=0))
        return dichotomie(g,m,ip);
    return dichotomie(m+1,d,ip);
}
public static boolean estDansListeNoireTriee(String ip) {
    return dichotomie(0, TAILLELN-1, ip);
}
```

Partie II — La liste blanche

Le principal danger du traitement automatique est de perdre des courriels valides. Pour éviter cela, on va faire une liste d'adresses électroniques (de type `String`) connues dont on pense qu'elles envoient toujours des courriels valides.

Question 12. Écrivez une classe `ListeBlanche` représentant une liste chaînée de `String`. Cette classe devra contenir un constructeur explicite.

Corrigé.

```
public class ListeBlanche {
    String adr;
    ListeBlanche suite;

    public ListeBlanche(String mel, ListeBlanche l) {
        this.adr = mel;
        this.suite = l;
    }
}
```

Dans la suite de cette partie, tout le code demandé sera écrit dans cette classe `ListeBlanche`.

Question 13. Écrivez une fonction

```
public static boolean estDansListeBlanche(String mel, ListeBlanche l)
```

qui teste si une adresse électronique est dans une liste blanche.

Corrigé.

```
public static boolean estDansListeBlanche(String mel, ListeBlanche l) {  
    for(ListeBlanche m = l; m != null; m = m.suite)  
        if(mel.equals(m.adr))  
            return true;  
    return false;  
}
```

Question 14. Pour donner la possibilité d'ajouter une adresse électronique à la liste blanche, écrivez une fonction

```
public static ListeBlanche ajouter(String mel, ListeBlanche l)
```

qui renvoie la liste l à laquelle on a ajouté l'adresse électronique mel. Attention, la liste ne doit pas contenir deux fois la même adresse électronique.

Corrigé.

```
public static ListeBlanche ajouter(String mel, ListeBlanche l) {  
    if(estDansListeBlanche(mel, l))  
        return l;  
    return new ListeBlanche(mel, l);  
}
```

Question 15. Si on s'est trompé, il faut pouvoir retirer une adresse d'une liste blanche. Écrivez une fonction

```
public static ListeBlanche retirer(String mel, ListeBlanche l)
```

qui retourne la liste l privée de l'adresse électronique mel (on ne suppose pas *a priori* que cette adresse est présente dans la liste de départ, mais on suppose qu'elle ne peut y être qu'une fois au plus). On demande que cette fonction ne crée pas de nouvelle cellule de liste (pas d'appel direct ou indirect au constructeur de ListeBlanche).

Corrigé.

```
public static ListeBlanche retirer(String mel, ListeBlanche l) {  
    if(l==null)  
        return l;  
    if(mel.equals(l.adr))  
        return l.suite;  
    l.suite = retirer(mel, l.suite);  
    return l;  
}
```

Question 16. Il peut être aussi utile de pouvoir importer des listes blanches dans sa liste blanche plutôt que de rentrer toutes les adresses électroniques une par une.

Écrivez une fonction

```
public static ListeBlanche fusionner(ListeBlanche l1, ListeBlanche l2)
```

qui retourne une liste contenant les adresses électroniques de l1 et l2. On demande que cette fonction ne modifie pas ses arguments : l1 et l2 doivent être les mêmes avant et après l'appel à fusionner(l1,l2) (les appels au constructeur de ListeBlanche sont ici autorisés). Le résultat de fusionner ne doit pas contenir d'adresse électronique en double. On pourra supposer que ni l1 ni l2 ne contient d'adresse en double.

Corrigé.

```
public static ListeBlanche fusionner(ListeBlanche l1, ListeBlanche l2) {  
    if(l1 == null)  
        return l2;  
    if(estDansListeBlanche(l1.adr, l2))  
        return fusionner(l1.suite, l2);  
    return new ListeBlanche(l1.adr, fusionner(l1.suite, l2));  
}
```