Corrigé du Contrôle d'Informatique INF 311

Promotion 2011

Sujet proposé par François Morain

12 juillet 2012

Seuls les documents fournis dans le cadre du cours et les notes personnelles sont autorisés.

Durée : 2 heures. Les étudiants non-francophones ont le droit à 30 minutes supplémentaires.

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.

Un barème pour chaque exercice, pour un total de 20 points, est donné à titre indicatif. Nous donnons le nombre de lignes attendues pour la réponse à chaque question ; si la réponse que vous envisagez est beaucoup plus longue, il est probable qu'il s'agisse d'une solution trop compliquée.

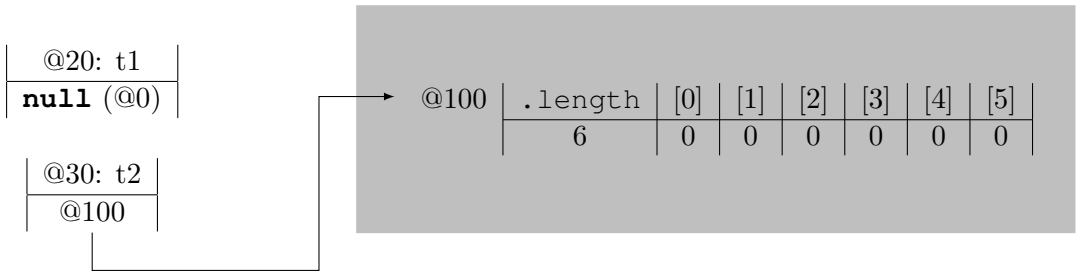
Exercice 1 *barème envisagé : 2 points*

On considère la classe:

```
public class Question1{
  public static void f(long[] t, int n){
    if(n == 0)
      t[n] = 0;
    else{
      f(t, n-1);
      t[n] = n * t[n-1];
    }
  }
  public static long[] f(int n){
    long[] t = null;
    if(n >= 0){
      t = new long[n+1];
      f(t, n);
    }
    return t;
  }
  public static void main(String[] args){
    long[] t1 = f(-1);
    long[] t2 = f(5);
    // (*)
  }
}
```

a) Faire un dessin de la mémoire locale et globale à l'exécution, quand on arrive sur la ligne (*) à la fin de la méthode `main`.

(réponse attendue : un dessin)
Corrigé.



`t1` contient **`null`** et `t2` une référence non nulle à un tableau de taille 6 en mémoire, plus la place pour `t2.length`. Le tableau `t2` ne contient que des 0.

b) Expliquer comment faire pour que `f` remplisse `t[n]` avec la valeur $n!$.

(réponse attendue : 2 lignes)
Corrigé. Il suffit d'écrire

```
if(n == 0)
  t[n] = 1;
```

Exercice 2 *barème envisagé : 10 points*

Le but de cet exercice est de modéliser et programmer une représentation des compétitions de type Euro de football. Aucune connaissance sportive avancée n'est nécessaire pour traiter cet exercice.

La phase finale de la compétition rassemble 16 équipes, et consiste en deux parties, la première avec des matchs de groupes et la seconde procédant par élimination directe.

a) Pour simplifier, chaque équipe représente un pays et a un sélectionneur (*coach*). Écrire une classe `Equipe` qui modélise une équipe. Cette classe devra contenir un constructeur explicite.

(réponse attendue : moins de 10 lignes)

Corrigé.

```
public class Equipe{

  public String pays;
  public String coach;

  public Equipe(String p, String c){
    this.pays = p;
    this.coach = c;
  }
}
```

b) Dans la première phase, les 16 équipes sont réparties en 4 groupes de 4 équipes, chaque groupe étant désigné par une lettre : A, B, C, ou D. Dans chaque groupe, chaque équipe rencontre une seule fois toutes les autres. On note $\mathcal{B}(x, y)$ (respectivement $\mathcal{B}(y, x)$) le nombre de buts marqués

par x contre y (respectivement $\mathcal{B}(y,x)$). Par exemple, si x bat y 3 buts à 2, on aura $\mathcal{B}(x,y)=3$, $\mathcal{B}(y,x)=2$.

i) Écrire une classe `Groupe` qui contiendra toutes les informations pertinentes relatives à ce groupe, notamment un tableau bi-dimensionnel `buts` contenant les quantités $\mathcal{B}(x,y)$. On incluera un constructeur explicite.

(réponse attendue : moins de 10 lignes)

Corrigé. Aux éléments factuels, on rajoute un tableau qui contiendra les buts marqués : `buts[i][j]` contient $\mathcal{B}(x,y)$ où x est la i-ème équipe et y la j-ème.

```
public class Groupe{

  public char nom;
  public Equipe[] equipe;
  public int[][] buts;

  public Groupe(char n, Equipe[] te){
    this.equipe = te;
    this.nom = n;
    this.buts = new int[te.length][te.length];
  }
}
```

ii) Créer une classe `Euro2012`, qui contiendra des constantes pour le nombre d'équipes, de groupes. On écrira également une méthode

```
public static Groupe construireGroupeA(){ ... }
```

décrivant les données réelles du groupe A de l'Euro 2012, à savoir les équipes

```
Pologne Grece Russie RepTcheque
```

de sélectionneurs respectifs

```
Smuda Santos Advocaat Bilek
```

Les scores des matchs du groupes A sont les suivants:

```
Pologne Grece 1 1
Russie RepTcheque 4 1
Grece RepTcheque 1 2
Pologne Russie 1 1
Grece Russie 1 0
RepTcheque Pologne 1 0
```

ce que l'on transcrit dans le tableau suivant, où on met sur une ligne les buts marqués par l'équipe de la colonne de gauche contre ses adversaires dans les colonnes:

	Pologne	Grece	Russie	RepTcheque
Pologne	–	1	1	0
Grece	1	–	1	1
Russie	1	0	–	4
RepTcheque	1	2	1	–

(réponse attendue : une dizaine de lignes)

Corrigé.

```
public class Euro2012{

  final public static int NGROUPES = 4;
  final public static int NEQUIPES = 4;

  public static Groupe construireGroupeA(){
    String[] pays = {"Pologne", "Grece", "Russie", "RepTcheque"};
    String[] coachs = {"Smuda", "Santos", "Advocaat", "Bilek"};
    Equipe[] te = new Equipe[NEQUIPES];
    for(int i = 0; i < NEQUIPES; i++)
      te[i] = new Equipe(pays[i], coachs[i]);
    Groupe groupe = new Groupe('A', te);
    groupe.buts = new int[][] {{0, 1, 1, 0},
                               {1, 0, 1, 1},
                               {1, 0, 0, 4},
                               {1, 2, 1, 0}};
    return groupe;
  }
```

iii) Faire un dessin partiel de l'occupation mémoire de ce groupe. On dessinera les champs du groupe et on se restreindra aux données plus précises pour la Pologne.

(réponse attendue : un dessin)

Corrigé. Voir figure 1.

À partir de maintenant, les méthodes demandées seront implicitement écrites dans la classe `Groupe`*.*

c) Il faut désormais déterminer les points obtenus par chaque équipe x à la fin de cette première phase, que l'on note $\mathcal{P}(x)$. Si x rencontre y, alors

$$p(x,y) = \begin{cases} 3 & \text{si } \mathcal{B}(x,y) > \mathcal{B}(y,x), \\ 1 & \text{si } \mathcal{B}(x,y) = \mathcal{B}(y,x), \\ 0 & \text{sinon} \end{cases}$$

et $\mathcal{P}(x)$ est la somme des $p(x,y)$ pour toutes les équipes y rencontrées par x.

Pour le groupe A, on trouve

Pologne	Grece	Russie	RepTcheque
2	4	4	6

i) Écrire dans la classe `Groupe` une méthode

```
public static int pointsDuMatch(int nbuts_x, int nbuts_y){ ... }
```

qui permet de calculer $p(x,y)$ en fonctions des nombres de buts marqués par x et y.

(réponse attendue : 5 lignes)

Corrigé.

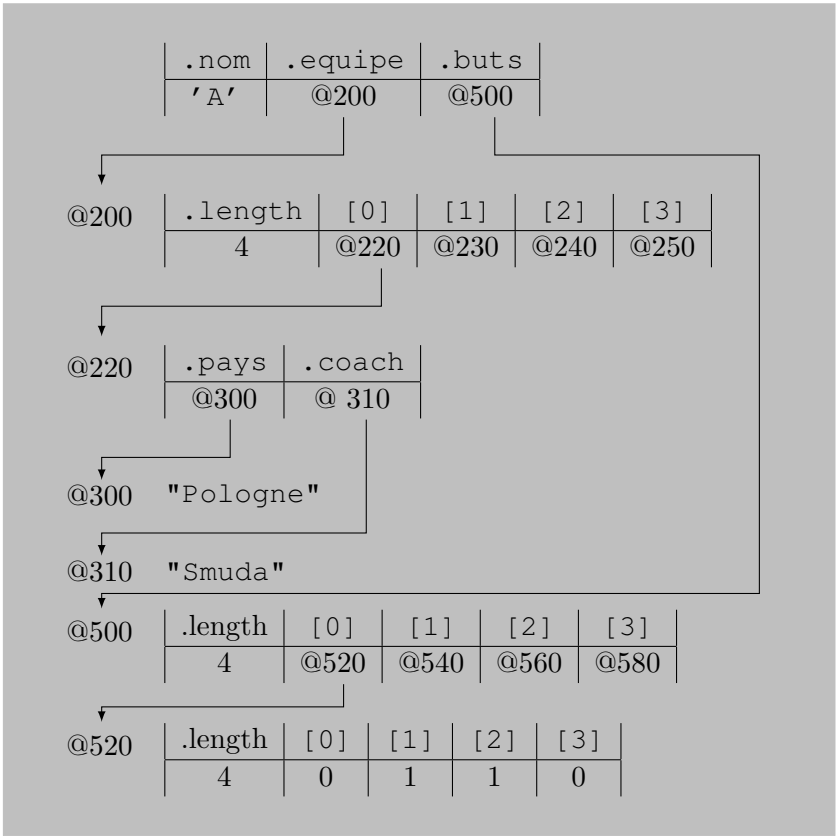


Figure 1: Stockage de groupe en mémoire.

```
public static int pointsDuMatch(int nbuts_x, int nbuts_y){
  if(nbuts_x > nbuts_y)
    return 3;
  else if(nbuts_x < nbuts_y)
    return 0;
  else
    return 1;
}
```

ii) En déduire la méthode

```
public int[] calculerPoints(){ ... }
```

qui retourne un tableau contenant les points de chaque équipe.

(réponse attendue : une dizaine de lignes)

Corrigé.

```
public int[] calculerPoints(){
  int ne = this.equipe.length;
  int[] points = new int[ne];
```

```
    for(int i = 0; i < ne; i++){
      for(int j = i+1; j < ne; j++){
        points[i] += pointsDuMatch(this.buts[i][j],
                                   this.buts[j][i]);
        points[j] += pointsDuMatch(this.buts[j][i],
                                   this.buts[i][j]);
      }
    }
    return points;
  }
```

d) Il faut maintenant classer les équipes de chaque groupe.

i) Écrire une méthode

```
public static int[] trierPoints(int[] points){ ... }
```

qui trie le tableau d'entiers `points` en utilisant une copie du tableau. Par exemple, dans le cas où

```
points = {2, 4, 4, 6},
```

la méthode créera le nouveau tableau trié `temp = {6, 4, 4, 2}`. Les cas d'égalité seront traités plus loin, donc l'ordre des 4 importe peu ici.

On veut également que la méthode retourne un tableau `ind` qui permette de relier les valeurs triées aux valeurs de départ, pour permettre de lister les équipes associées aux nombres de points triés. On initialisera `ind[i] = i` et on fera évoluer ce tableau au cours du tri. Ainsi, à la fin de l'algorithme, `groupe.equipe[ind[i]]` contiendra l'équipe ayant le i-ème rang. Pour l'exemple de `points = {2, 4, 4, 6}`, l'algorithme trouvera

```
ind = {3, 1, 2, 0}.
```

On pourra s'inspirer du tri sélection vu en cours.

(réponse attendue : moins d'une vingtaine de lignes)

Corrigé. On fait subir à `ind` les mêmes permutations qu'au tableau `tmp`, qui contient une copie de `points`:

```
public static int[] trierPoints(int[] points){
  // copie
  int[] tmp = new int[points.length];
  for(int i = 0; i < points.length; i++)
    tmp[i] = points[i];

  // initialisation
  int[] ind = new int[points.length];
  for(int i = 0; i < points.length; i++)
    ind[i] = i;

  // on imite le tri selection
  for(int i = 0; i < tmp.length; i++){
    int indmax = i;
```

```
    for(int j = i+1; j < tmp.length; j++)
      if(tmp[j] > tmp[indmax])
        indmax = j;
    int itmp = tmp[i];
    tmp[i] = tmp[indmax];
    tmp[indmax] = itmp;
    itmp = ind[i];
    ind[i] = ind[indmax];
    ind[indmax] = itmp;
  }
  return ind;
}
```

ii) En déduire la méthode d'objet:

```
public Equipe[] classer(){ ... }
```

qui trie les équipes par ordre décroissant à l'aide de la méthode précédente et retourne un tableau formé des équipes dans l'ordre de qualification.

(réponse attendue : moins de 10 lignes)

Corrigé.

```
public Equipe[] classer(){
  int[] points = this.calculerPoints();
  int[] ind = trierPoints(points);
  Equipe[] tmp = new Equipe[this.equipe.length];

  for(int e = 0; e < this.equipe.length; e++)
    tmp[e] = this.equipe[ind[e]];
  return tmp;
}
```

e) Nous allons maintenant traiter les cas d'égalité de points. On s'intéresse à classer entre elles les r équipes x_i, x_{i+1}, ..., x_{j-1}, avec $j - i = r \geqslant 2$. C'est le cas par exemple de $\{6, 4, 4, 2\}$ où $i = 1$, $j = 3$, $r = 2$.

On applique alors la règle suivante : si une des équipes x_k a battu toutes les autres équipes de x_i, ..., x_{j-1}, alors x_k est classée en tête, et il ne reste plus qu'à classer les autres équipes.

Commentaire: cette règle suffit à départager les équipes de l'Euro 2012. On trouvera facilement les règles complètes de la compétition sur le web.

i) Écrire une méthode

```
public static boolean regle(int[][] buts, int[] ind,
                            int i, int j){ ... }
```

qui implante la règle et retourne `true` si la règle a pu trouver un gagnant `ind[k]` dans les équipes d'indices `ind[i]`, ..., `ind[j-1]`, auquel cas elle retourne `true` et `false` sinon. Si la règle s'applique, le tableau `ind` est modifié de sorte que les indices `ind[i]` et `ind[k]` ont été échangés. Il ne reste plus alors qu'à appliquer la règle pour `ind[i+1]`, ..., `ind[j-1]`.

Dans l'exemple numérique, on part de `ind[1]` et `ind[2]` et on vérifie que l'un des deux est plus grand, et on a terminé.

(réponse attendue : une quinzaine de lignes)

Corrigé.

```
// OUTPUT: true si la regle s'applique
//         false sinon
// EFFET DE BORD : ind[] est modifie pour tenir compte du resultat
public static boolean regle(int[][] buts, int[] ind, int i, int j){
  boolean oui = true;

  for(int k = i; k < j; k++){
    // est-ce que x_k a battu tous les autres?
    // il ne peut y avoir qu'un seul k dans ce cas...!
    oui = true;
    for(int r = i; r < j; r++)
      if((r != k) && (buts[ind[k]][ind[r]] <= buts[ind[r]][ind[k]])){
        oui = false;
        break;
      }
    if(oui){
      // on permute ind[k] et ind[i]
      int tmp = ind[i];
      ind[i] = ind[k];
      ind[k] = tmp;
      break;
    }
  }
  return oui;
}
```

ii) En déduire la méthode récursive

```
public static boolean traiterEgalites(int[][] buts, int[] ind,
                                       int i, int j){ ... }
```

qui applique la règle tant que c'est possible. Elle retournera `true` si tous les cas d'égalité ont été tranchés et `false` sinon.

(réponse attendue : moins de 10 lignes)

Corrigé.

```
// [i..j[ est une sous-suite constante
public static boolean traiterEgalites(int[][] buts, int[] ind,
                                       int i, int j){
  if((j-i) == 1)
    return true;
  if(regle(buts, ind, i, j))
    return traiterEgalites(buts, ind, i+1, j);
  TC.println("Je n'ai pas pu re'soudre ["+i+" "+j+"[");
  return false;
}
```

iii) Écrire la méthode finale

```
public static void traiterEgalites(int[][] buts, int[] points,
                                   int[] ind){ ... }
```

qui recherche les sous-suites constantes et traite les cas d'égalité.

(réponse attendue : moins de 10 lignes)

Corrigé.

```
public static void traiterEgalites(int[][] buts,
                                   int[] points,
                                   int[] ind){
  // trouvons les sous-suites constantes
  int i = 0;
  while(i < points.length){
    int j;
    for(j = i+1; j < points.length; j++)
      if(points[ind[j]] != points[ind[i]])
        break;
    // [i..j[ est une sous-suite constante
    traiterEgalites(buts, ind, i, j);
    i = j;
  }
}
```

iv) Compléter la méthode `classer()` de la question d-ii en rajoutant les cas d'égalité.

(réponse attendue : 5 lignes)

Corrigé.

```
public Equipe[] classer(){
  int[] points = this.calculerPoints();
  int[] ind = trierPoints(points);

  traiterEgalites(this.buts, points, ind);

  Equipe[] tmp = new Equipe[this.equipe.length];
  for(int e = 0; e < this.equipe.length; e++)
    tmp[e] = this.equipe[ind[e]];
  return tmp;
}
```

Exercice 3 *barème envisagé : 8 points*

Soit A un arbre binaire de recherche avec n nœuds, chaque nœud contenant un entier. On écrira un arbre binaire $A = (r, G, D)$ où r est la racine de A, G et D les sous-arbres gauche et droit respectivement.
a) On considère les deux arbres de la figure 2. Quelle est leur hauteur ? Combien ont-ils de nœuds internes, de feuilles ? Pourquoi est-ce que ce sont des arbres binaires de recherche ?
(réponse attendue : 3 lignes)

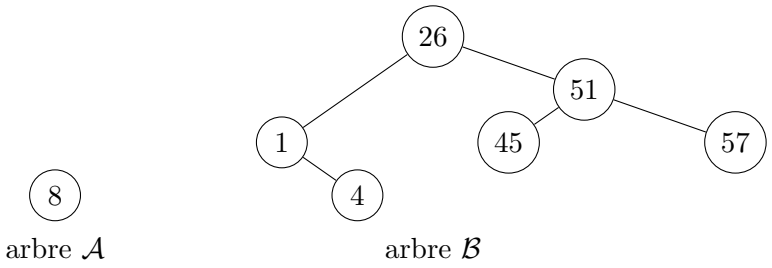


Figure 2: Deux arbres.

Corrigé. L'arbre $\mathcal{A}$ a hauteur 1, possède 0 nœuds internes et une feuille. C'est un arbre binaire car toute racine est certainement plus grande qu'un ensemble vide: toute propritété *quel que soit* $x \in \emptyset$ *on a* $P(x)$ est en effet vraie.

L'arbre $\mathcal{B}$ est de hauteur 3, possède 3 nœuds internes et 3 feuilles. Chaque nœud est plus grand que tous ses enfants gauches et plus petit que tous ses enfants droits.

b) Écrire la classe `ABR`. Celle-ci devra contenir une définition récursive de la structure, un constructeur explicite, ainsi qu'une méthode d'insertion

```
public static ABR inserer(ABR A, int x){ ... }
```

et une méthode transformant un tableau d'entier en ABR:

```
public static ABR deTableau(int[] t){ ... }
```

(réponse attendue : 20 lignes)
Corrigé. On copie-colle la classe du cours et on rajoute la méthode demandée:

```
public class ABR{
    private int racine;
    private ABR gauche, droit;

    public ABR(int r, ABR g, ABR d){
        this.racine = r;
        this.gauche = g;
        this.droit = d;
    }

    public static ABR inserer(ABR a, int x){
        if(a == null)
            // on cree un nouvel ABR
            return new ABR(x, null, null);
        else if(x <= a.racine){
            // on insere dans le sous-arbre gauche
            a.gauche = inserer(a.gauche, x);
            return a;
        }
        else // x > a.racine;
            // on insere dans le sous-arbre droit
            a.droit = inserer(a.droit, x);
```

```
        return a;
    }

    public static ABR deTableau(int[] t){
        if(t.length == 0)
            return null;
        ABR a = new ABR(t[0], null, null);
        for(int i = 1; i < t.length; i++)
            a = inserer(a, t[i]);
        return a;
    }
}
```

On considère l'algorithme suivant, qui *aplatit* l'arbre: si $A = \emptyset$, on ne fait rien; si $A = (r, G, D)$:

- on aplatit G;
- on affiche r;
- on aplatit D.

c) *i)* Que doit afficher l'algorithme sur l'arbre $\mathcal{B}$ de la figure 2. Que constate-t-on ?

(réponse attendue : 2 lignes)

Corrigé. On trouve : `1 4 26 45 51 57`. La suite est triée par ordre croissant.

ii) Démontrer ce résultat en toute généralité.

(réponse attendue : un dessin, moins de 5 lignes)

Corrigé. On trouve à la figure 2 le résultat de l'aplatissement de l'arbre $\mathcal{B}$.

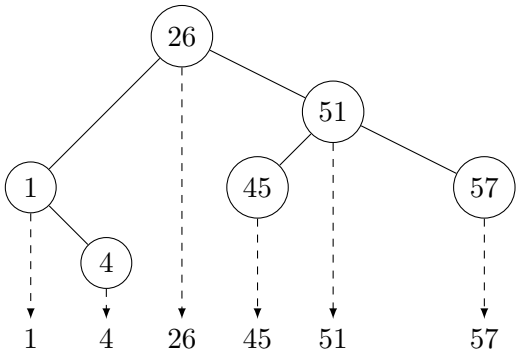


Figure 3: Aplatissement de l'arbre $\mathcal{B}$.

Si $A = (r, \emptyset, \emptyset)$, l'aplatissement donne r qui est bien trié. Par récurrence sur la hauteur des arbres, $apl((r, G, D)) = apl(G), r, apl(D)$, on applique la propriété démontrée pour les arbres plus petits G et D et on se rappelle que tous les nœuds de G sont plus petits que r, lui-même plus petit que tous les nœuds de D.

d) *i)* Écrire la méthode

```
public static void aplatir(ABR A){ ... }
```

(réponse attendue : 5 lignes)
Corrigé.

```
public static void aplatir(ABR A){
  if(A != null){
    aplatir(A.gauche);
    TC.print(" "+A.racine);
    aplatir(A.droit);
  }
}
```

ii) Quel est le nombre d'appels de cette méthode en fonction du nombre de nœuds n?

(réponse attendue : 1 ligne)

Corrigé. On a $A(1) = 2$; pour $n > 1$, $A(n) = 2A(n/2)$, d'où $A(n) = 2n$ et au total $2n + 1$ en comptant le premier appel.

e) Modifier la méthode précédente pour qu'elle remplisse un tableau avec le résultat de l'aplatissement :

```
public static int aplatirDansTableau(int[] t, ABR A, int i0){ ... }
```

où la méthode prend l'indice `i0` à partir duquel on doit remplir `t` avec le contenu de `A` et retourne le prochain indice à utiliser. On supposera que `t` a une taille suffisante. On écrira également la méthode d'appel:

```
public static int[] aplatirDansTableau(ABR A, int n){ ... }
```

qui aplatit un arbre avec n nœuds dans un tableau de n éléments qui est retourné par la méthode.

(réponse attendue : une dizaine de lignes)

Corrigé.

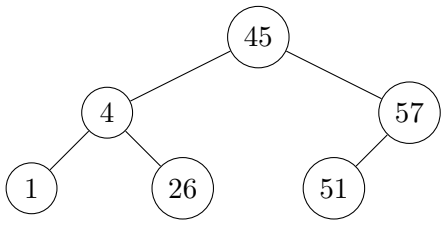
```
public static int aplatirDansTableau(int[] t, ABR A, int i0){
  if(A == null)
    return i0;
  else{
    int i = aplatirDansTableau(t, A.gauche, i0);
    t[i] = A.racine;
    return aplatirDansTableau(t, A.droit, i+1);
  }
}

public static int[] aplatirDansTableau(ABR A, int n){
  int[] t = new int[n];
  aplatirDansTableau(t, A, 0);
  return t;
}
```

f) Écrire une méthode

```
public static ABR reconstruire(int[] t){ ... }
```

qui prend en entrée un tableau trié et construit un ABR le plus équilibré possible. Sur l'exemple `t = {1, 4, 26, 45, 51, 57}`, on devra obtenir:



Indication: considérer `t[n/2]`.

(réponse attendue : moins de 10 lignes)

Corrigé. On procède par dichotomie: on construit un arbre de racine `t[n/2]` dont les enfants gauche et droit sont constitués récursivement à partir de `t[0..n/2[` et `t[1+(n/2)..n[`. Chaque sous-arbre aura ainsi à peu près la même taille.

```
// t est deja trie, on doit reconstruire un arbre le plus
// equilibre possible, a partir de la mediane
public static ABR reconstruire(int[] t, int i, int j){
  if(i >= j)
    return null;
  int m = (i+j)/2;
  ABR g = reconstruire(t, i, m);
  ABR d = reconstruire(t, m+1, j);
  return new ABR(t[m], g, d);
}

public static ABR reconstruire(int[] t){
  return reconstruire(t, 0, t.length);
}
```