Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2016-2017**

TD - Séance n°3

Révisions – Récursion, Modélisation

Pour ceux qui ne l'ont pas encore fait : inscrivez-vous sur DidEL !
Une fois inscrit au *COURS*, pensez à aussi vous inscrire dans *VOTRE GROUPE*. Si vous n'avez pas le même groupe de TD et TP, inscrivez-vous dans les deux.
L'adresse de DidEL : *http ://didel.script.univ-paris-diderot.fr/*
Le code du cours : *POOIGTDTP*

Lisez attentivement le sujet, les exercices sont toujours plus faciles à faire quand on a *bien* lu l'énoncé. Parfois il peut être utile de lire les questions suivantes : comme ça on sait où l'exercice veut en venir !

Les exercices de la premiere partie du TD doivent tous être traités avant la semaine prochaine. Finissez ceux que vous n'avez pas eu le temps de faire en TD chez vous. Les exercices de la seconde partie sont à faire si vous vous sentez à l'aise.

1 Exercices obligatoires

Exercice 1 *Questions de cours*

1. Que fait le modificateur `final` appliqué à un attribut ? une fonction ? une classe ?
2. Quel peut-être l'intérêt d'un champ `public static final` ? Avez-vous des exemples ?

Exercice 2 *Évaluation de code*

Lesquels de ces morceaux de codes produisent une erreur à la compilation ? Pourquoi ?

```
1 :
  final int[] tab = {1,2,3,4};
  tab[2] = 5;
2 :
  final int[] tab;
  tab = new int[2];
3 :
  final int[] tab;
  tab = new int[2];
  tab = new int[2];
4 :
  final int[] tab;
  if(Math.random()>0.5)
    tab = new int[] {1,2,3};
  else
    tab = new int[5];
```

```
5 :
  final int[] tab;
  switch((int)(Math.random()*3) {
    case 0 :
    tab = new int[0];
    break;
    case 1 :
    tab = new int[1];
    case 2 :
    tab = new int[2];
    break;
  }
6 :
    final int[][] tab = new
int[2][2];
    tab[1] = new int[3];
```

Exercice 3 *Récursivité*

On veut implémenter un algorithme de tri nommé *tri rapide* (*quicksort* en anglais). Son fonctionnement est le suivant :

1. On reçoit une liste d'entiers.
2. Si la liste est de taille 0 ou 1, on renvoie la liste telle quelle (elle est déjà triée).
3. Sinon, on prend un élément de cet liste (par exemple, le premier, le dernier, ou un au hasard), qu'on nomme *pivot*.
4. On répartit alors la liste en deux listes : celle des éléments inférieurs au pivot, et celle des éléments supérieurs au pivot.
5. On trie les deux listes ainsi créées en appelant notre algorithme dessus.
6. On fusionne les deux listes, maintenant triées, et le pivot.

Son fonctionnement est *récursif*, car il s'appelle lui-même sur de plus petites listes. Voir la Figure 1 pour un exemple d'exécution. On va pour cela créer une classe `Trieur` avec une fonction :
`public void trier(LinkedList<Integer> maListe)`.

1. Donner un code de la fonction `trier`.

   On veut maintenant que le trieur ait une option *bavard*, que l'on peut mettre à `vrai` ou a `faux`. Lorsque cette option est activée, le `Trieur` décrit le déroulement de l'algorithme, en indiquant notamment à chaque appel de `trier` à quel niveau de récursion il se trouve.
2. Comment pourrait-on implémenter cette fonctionnalité ?

On se propose maintenant de réaliser une version légèrement différente, où on peut donner un paramètre au Trieur à la création qui correspond au nombre de pivot qu'on utilise. Pour n pivots, on va alors couper notre liste en $n+1$ sous-listes avant de faire l'appel récursif.

3. Quels attributs pourrait-on donner au champ n, si l'on veut qu'il soit visible de l'extérieur mais fixe après la création de l'objet `Trieur` ?
4. *(très facultatif)* Donner une implémentation de cette version modifiée. On pourra faire l'algorithme sur des exemples avec $n=2$, 3 ... pour saisir la logique.
5. *(encore plus facultatif)* On aimerait donner une dimension orientée objet à ce tri en créant deux classes, `Trieur` et `TacheDeTri`, où `Trieur` gère une pile d'appel de `TacheDeTri`. L'intérêt serait de pouvoir répartir le travail entre plusieurs agents par exemple. Comment pourrait-on procéder ?

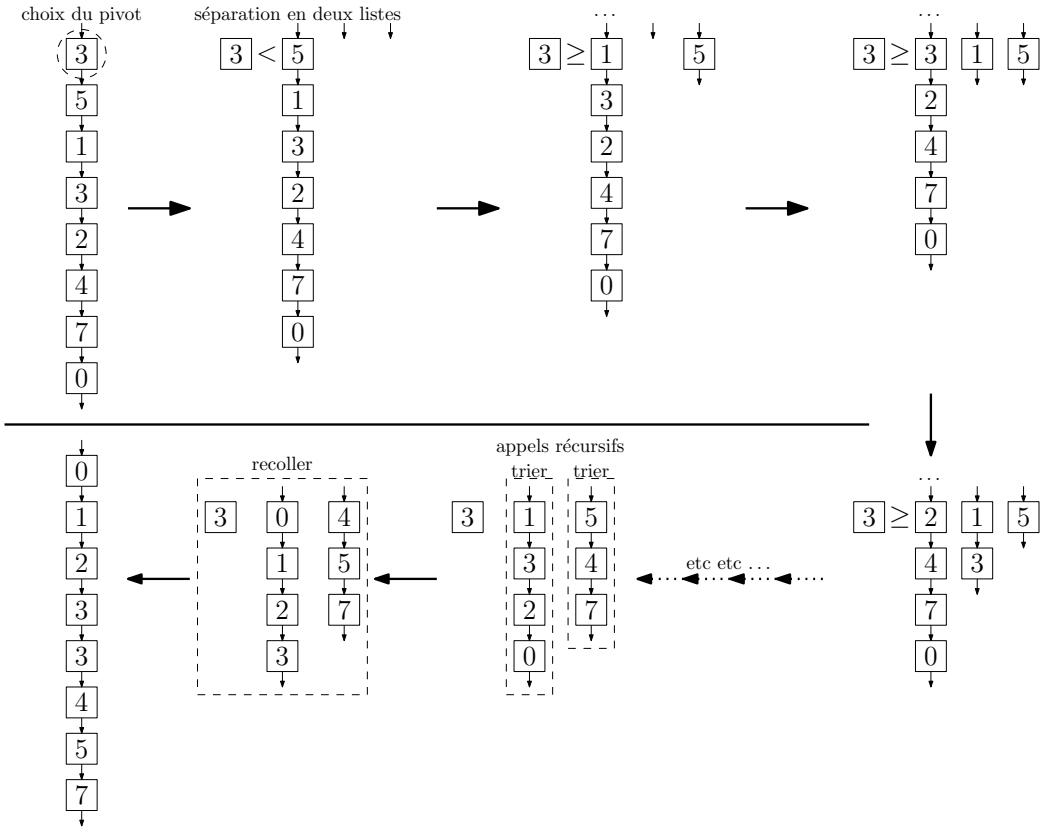

FIGURE 1 – Exemple d'exécution du tri rapide

2 Si vous avez du temps . . .

. . .vous pouvez déjà vous familiariser avec le jeu que vous implémenterez durant la prochaine séance de TP. Comment pourrait-on le modéliser en Java ? Réfléchissez aux classes que vous comptez définir, et commencez à spécifier leurs méthodes, sans encore les implémenter. (Donnez seulement les signatures des méthodes et décrivez ce qu'elles feront.)

Évidemment, il y a plusieurs manières de faire, mais certaines solutions sont plus judicieuses que d'autres. Si vous avez un doute, parlez-en à votre enseignant.

Exercice 4 *Démineur*

Le démineur est un jeu où le joueur doit trouver les mines d'un terrain miné sans les déclencher.

Au départ, le terrain est complêtement invisible. Á chaque tour, le joueur peut décider de découvrir une case. Si cette case est dépourvue de mine, la case est révélée, s'il y avait une mine, le jeu s'arrête, et on a perdu.

Une case révélée indique le nombre de mines dans les 8 cases adjacentes à la case révélée. S'il n'y a aucune mine dans les 8 cases adjacentes, alors on révèle aussi les 8 cases autour de la première case révélée, car il n'y a aucun risque à le faire. Si dans ces 8 cases il y a encore une case sans mines autour d'elle, alors on révèle aussi ses voisins etc etc ...

Le joueur a aussi la possibilité de poser un drapeau sur certaines cases pour se souvenir qu'il soupçonne qu'il y ait une mine sur cette case. Ce drapeau sert comme une sécurité : si le joueur demande à révéler une case avec un drapeau dessus, le jeu refuse de révéler la case. Le joueur doit d'abord enlever le drapeau.

Le jeu s'achève par une victoire lorsque toutes les cases dépourvues de mines sont révélées.

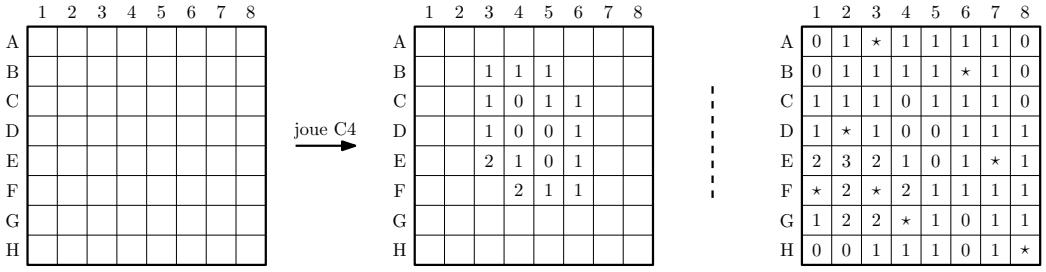

FIGURE 2 – À gauche, le joueur joue son premier coup en C4 et découvre toute une zone. À droite, le terrain complètement découvert