Introduction à la programmation en Java

UFR Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

COURS 8

Compléments sur les tableaux

Où en sommes-nous ?

♦Nous savons rédiger le texte d'une classe d'objets, avec ses **champs**, ses **constructeurs**, ses **méthodes**.
♦Nous pouvons exprimer qu'une méthode a ou n'a pas de résultat.
♦Nous savons utiliser les tableaux à une dimension

```
int[] t = new int[5];
t.length ——————————→ 5
```

- `t[0]` est le premier élément, `t[4]` est le dernier élément
- Chaque élément est une variable de type `int` « normale »

♦Étape suivante : approfondir la connaissance des tableaux, notamment sur la recherche et le tri.

Sur l'initialisation d'un tableau

```
int[] tab1 = {4, 6, 3, 7, 9, 2, 3};
tab1.length ——————→ 7
```

```
String[] tab2 = {"chou", "pou", "hibou"};
tab2.length ——————→ 3
tab2[0] ——————→ "chou"
```

```
int[][] tab3 = {{1,2,3}, {4,5,6,7}};
tab3.length ——————→ 2
tab3[0].length ——————→ 3
tab3[1].length ——————→ 4
```

Qu'est-ce qu'un tableau trié ?

♦Un tableau $\{t_0, t_1, \ldots, t_{n-1}\}$ est **trié** (sous-entendu en croissant) lorsque $t_i \leq t_{i+1}$ pour tout $i \in [0,n-2]$.

♦Exemples :

`int[ ] tab1 = {-5, 6, 8, 9, 9, 12}` est trié.

`int[ ] tab2 = {-5, 8, 6, 9, 9, 12}` n'est pas trié !

```
boolean estCroissant()    // classe Tableau TP6
{
   for (int i=0; i< tab.length-1; i+=1)
      if (tab[i] > tab[i+1]) return false;
   return true;
}
```


Recherche dans un tableau trié

```
class Tableau {                              // TP6…
    int[] tab;

    /** on suppose que tab est trié en croissant ! */
    boolean orderedSearch(int x) {
       for(int i=0; i<this.tab.length; i+=1){
          if (this.tab[i] == x) return true;
          else if (this.tab[i] > x) return false;  // STOP !
       }
       return false;
    }

    static void test() {
       int[] tab = {3, 5, 8, 9, 12};
       Tableau t = new Tableau(tab, false);
       System.out.println("7 : " + t.orderedSearch(7));
       System.out.println("8 : " + t.orderedSearch(8));
    }
}
```

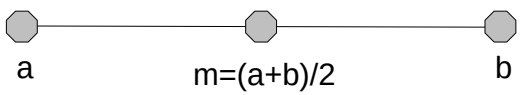
```
7 : false
8 : true
```

Et dans un tableau non trié ?

♦La méthode précédente `orderedSearch(x)` est **séquentielle** (de gauche à droite). Bien que profitant de l'ordre des éléments, elle ne fait guère mieux qu'une recherche non ordonnée. Son coût reste de l'ordre de n (en moyenne n/2) …

♦La **dichotomie** permet de mieux exploiter l'ordre :

```
boolean binarySearch(int x) {
  int a = 0, b = this.tab.length-1;     // indices extrêmes
  int milieu;
  while (a <= b) {
     milieu = (a + b) / 2;
     if (this.tab[milieu] == x) return true;
     else if (this.tab[milieu] < x)  a = milieu + 1;
     else b = milieu - 1;
  }
  return false;
}
```



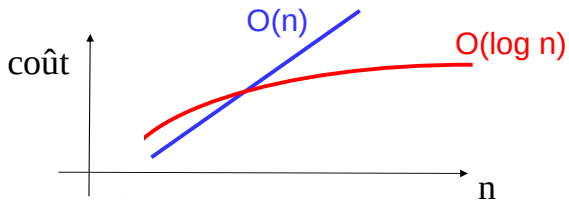
♦Nous avons vu que le coût d'une **recherche séquentielle** est de l'ordre de n. On dit qu'elle est « **linéaire** » (en n) : **O(n)**.

♦*Quid* de la **recherche dichotomique** `binarySearch(x)` ?

♦Dans le pire des cas, son coût est le nombre de fois que l'on peut diviser par 2 la longueur n du tableau. Soit h ce nombre. On a donc approximativement $2^h = n$, soit $h = \log_2(n)$.

♦Tous les logarithmes étant proportionnels, on dit que le coût est « **logarithmique** » (en n) : **O(log n)**.

♦Bien plus rapide que la recherche séquentielle :



Trier un tableau

♦Il existe des dizaines d'algorithmes de tri, plus ou moins rapides et basés sur des idées intéressantes et distinctes.

♦Le principe de base consiste à **trier sur place** (sans utiliser de tableau auxiliaire), et en procédant **par échange d'éléments** dans le tableau.

♦On peut montrer qu'alors les manières les plus rapides de trier un tableau de longueur n auront un coût en O(n log n). Les manières naïves ont un coût « quadratique » en $O(n^2)$ ou plus…

♦Nous commencerons par un exemple simple : le tri par sélection.

Méthode naïve facile à programmer…

Le tri par sélection

♦Il consiste à sélectionner les minima successifs du tableau et à les amener en tête

♦La partie jaune est « en place » et ne bouge plus !

♦Pour 8 éléments, il aura fallu 7 tours de boucle.

♦A chaque tour, il faut parcourir tous les éléments non encore triés (`n-i`).

5	8	4	9	2	1	3	2
1	8	4	9	2	5	3	2
1	2	4	9	8	5	3	2
1	2	2	9	8	5	3	4
1	2	2	3	8	5	9	4
1	2	2	3	4	5	9	8
1	2	2	3	4	5	9	8
1	2	2	3	4	5	8	9

Algorithme du tri par sélection

- Pour chaque position dans le tableau, on cherche tour à tour l'élément qui devrait y être
  - A la position 0, on met le plus petit de tous
  - A la position 1, on met le plus petit de tous (sauf le 1er)
  - A la dernière position, on met le seul qui reste (rien à faire)
  - ➢ `for(int i=0; i<tab.length-1; i++)`
- Le minimum entre l'indice a et l'indice b
  - **`min`** ← **`tab[a]`**
  - si **`tab[a+1]<min`** alors **`min`** ← **`tab[a+1]`**
  - si **`tab[a+2]<min`** alors **`min`** ← **`tab[a+2]`**
  - … pour tous les éléments jusqu'à **`tab[b]`**
- Pour inverser 2 éléments d'un tableau i et j
  - `int tmp = tab[i]; tab[i] = tab[j]; tab[j] = tmp;`

Notes

```
void selectionSort()
{
   for(int i=0; i<this.tab.length-1; i+=1) {
      // recherche du minimum de this.tab[i…n-1]
      int iMin = i, min = this.tab[i];
      for(int j=i+1; j<this.tab.length; j+=1) {
         int x = this.tab[j];
         if (x < min) { iMin = j; min = x; }
      }
      // échange du minimum avec this.tab[i]
      int tmp = this.tab[i];
      this.tab[i] = this.tab[iMin];
      this.tab[iMin] = tmp;
   }
}
```


Quelle est la complexité de selectionSort() ?

- La boucle *for* externe tourne n-1 fois.
- Au i-ème tour de la boucle for externe, on procède :
  - au calcul du minimum de tab[i..n-1] : coût = O(n-i)
  - à l'échange de tab[i] et du minimum : coût = O(1)

  coût constant
- Le nombre total de comparaisons est donc :

$$\sum_{i=1}^{n-1}(n-i) = (n-1)+(n-2)+(n-3)+\ldots+2+1 = O(n^2)$$

- Le nombre total d'échanges est quant à lui $\sum_{i=1}^{n-1} O(1) = O(n)$

Le coût est **quadratique** !

Le tri par insertion

- Il consiste à mettre à sa place chaque élément
- La partie jaune est « localement triée » mais peu évoluée !
- Pour 8 éléments, il y aura 7 tours de boucle.
- A chaque tour, il faut pousser les éléments de l'intervalle 0..i-1 pour faire la place au i^{ème} élément.

5	8	4	9	2	1	3	2
5	8	4	9	2	1	3	2
4	5	8	9	2	1	3	2
4	5	8	9	2	1	3	2
2	4	5	8	9	1	3	2
1	2	4	5	8	9	3	2
1	2	3	4	5	8	9	2
1	2	2	3	4	5	8	9

Algorithme du tri par insertion

- Pour chaque élément dans le tableau, on cherche tour à le mettre à sa place relative
  - On suppose que le tableau est trié entre [0,i[
  - Si i=1, c'est vrai (condition initiale)
  - On considère chaque élément non trié et on le fait descendre à sa place
  - `for(int i=1; i<tab.length; i++)`
- On fait descendre tab[i] à sa place
  - Comme tab est trié dans [0,i[, il suffit de trouver le premier élément tab[j] inférieur à tab[i]
  - `for(int j=i-1; j>=0 && tab[j]>tab[j+1]; j-=1){`
- Pour inverser 2 éléments d'un tableau j et j+1
  - `int tmp = tab[j]; tab[j] = tab[j+1]; tab[j+1] = tmp;`

```
public void insertionSort()
{
   for(int i=1; i<this.tab.length; i+=1) {
      // placer this.tab[i] par des échanges
      // tab[0..i-1] est trié, mais PAS en place
      for(int j=i-1; j>=0 && tab[j]>tab[j+1]; j-=1){
          // échange this.tab[j] et this.tab[j+1]
          int tmp = this.tab[j];
          this.tab[j] = this.tab[j+1];
          this.tab[j+1] = tmp;
      }
   }
}
```


Amélioration ?

```
public void insertionSort()
{
   for(int i=1; i<this.tab.length; i+=1) {
      // placer this.tab[i] par des échanges
      // tab[0..i-1] est trié, mais PAS en place
      int j = i;
      int tmp = this.tab[i];
      while(j>0 && this.tab[j-1] > tmp) {
         this.tab[j] = this.tab[j-1];
         j -= 1;
      }
      //   (tab[0..j-1] <= tmp || j==0)
      // && tab[j+1..i] > tmp
      this.tab[j] = tmp;
   }
}
```


Quelle est la complexité de insertionSort() ?

- La boucle *for* externe tourne n-1 fois.
- Au i-ème tour de la boucle for externe, on cherche la place de tab[i]:
  - dans le pire des cas : coût = O(i-1) comparaisons et échanges
  - dans le meilleur des cas : coût = O(1) *coût constant*
- Dans le pire des cas, le nombre total de comparaisons/échanges est :

$$\sum_{i=2}^{n}(i-1) = 1+2+\ldots+n-1 = n\times\frac{n-1}{2} = O(n^2)$$

quadratique !

- Dans le meilleur des cas : $\sum_{i=1}^{n-1} O(1) = O(n)$
- Quand obtient-on le meilleur des cas ? Le pire des cas ?

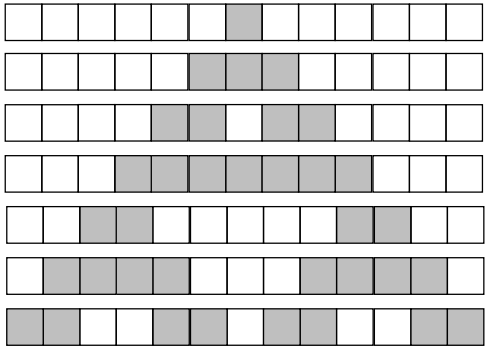
Notes

Les automates cellulaires

« … certains programmes parmi les plus simples peuvent réaliser des comportements aussi compliqués que l'on veut … » **Stephen Wolfram**

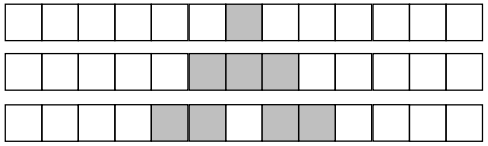
- Issus des travaux de Von Neumann
- Ont comme support une matrice de cellules de dimension 1, 2, 3 ou plus
- À chaque **génération**, toutes les cellules sont redéfinies à partir de la génération d'avant en appliquant une **fonction de transition simple**
- Suscite beaucoup d'intérêts :
  - Bien que les fonctions de transitions peuvent être simplistes, on obtient des comportements complexes qui rappellent d'autres modèles physiques ou biologiques ;
  - Le plus célèbre est le « Jeu de la vie » de Conway.

Un exemple



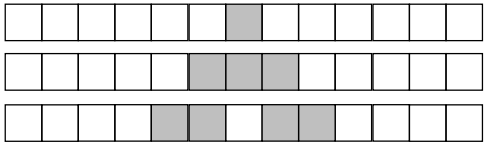
- Automate cellulaire à 1 dimension
- 2 états possibles pour chaque cellule : *active* ou *passive*
- Fonctions de transitions :
  - Une cellule reste active si elle a au plus 1 voisin actif
  - Une cellule devient active si elle a 1 ou 2 voisin actifs
  - Une cellule devient passive si elle a 2 voisins actifs

Concepts clés



- **Voisinage** :
  - Le nouvel état est déterminé à partir de la position spatiale en examinant l'état des cellules voisines (avec un **rayon** donné)
- **Parallélisme** :
  - Toutes les cellules sont mises à jour simultanément
- **Déterminisme** (≠ stochastique) :
  - La génération suivante est calculée à partir de l'état courant
- **Homogénéité** :
  - Toutes les cellules utilisent la même fonction de transition

Caractéristiques d'un automate cellulaire

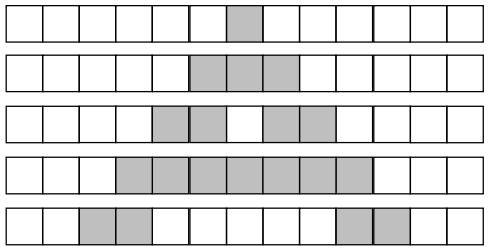


- **Dimension** :
  - 1 (automate élémentaire de Wolfram), 2 (jeu de la vie de Conway)
- **Voisinage** (r):
  - Les cellules utilisées pour calculer l'état suivant
  - Souvent limité à la cellule cible et à ses voisines directes (rayon≥1)
- **Espace d'état** (k):
  - Ici les cellules peuvent avoir seulement 2 états (active ou passive)
  - La couleur représente l'état
- **Fonctions de transitions** :
  - k^r fonctions différentes

Règles de transition

- **Conventions** de nom : automate 1D k=2 r=1 (2 états, rayon=1)

8 configurations possibles du voisinage	111	110	101	100	011	010	001	000
Le nouvel état de la cellule centrale	.0.	.1.	.1.	.1.	.1.	.1.	.1.	.0.
#règle	$01111110_2=126_{10}$							



Génération 0
Génération 1
Génération 2
Génération 3
Génération 4

Implémentation en Java

```
class AutomateCellulaire {
/** courante représente la génération courante */
   private boolean[] courante;
/**
 * @param taille de l'automate
 * @param init vaut true ssi on veut initialiser l'automate aléatoirement
 */
   AutomateCellulaire(int taille, boolean init) {
     this.courante = new boolean[taille];
     if (init) { // initialisation pseudo-aléatoire
       java.util.Random gen = new java.util.Random();
       for(int i = 0; i < taille; i += 1)
         this.courante[i] = gen.nextBoolean();
     }
   }
```



```
/**
 * calcule la génération suivante (#règle 126) pour simuler le parallélisme il faut
 * nécessairement utiliser un autre tableau
 */
  void generationSuivante() {
    boolean[] suivante = new boolean[courante.length];
    for (int i=0; i<this.courante.length; i+=1) {
      int v = this.getVoisinage(i);
      suivante[i] = (v!=0 && v!=7);
    }
    this.courante = suivante;
  }
```

- Ici, on fait une utilisation énorme de la mémoire
- On pourrait par soucis d'économie, remplacer la variable locale suivante par un attribut
- A chaque nouvelle génération, les tableaux sont échangés !
  - Comment fait-on pour échanger deux variables ??

Notes

```
  private boolean[] suivante;  // à initialiser dans le constructeur

/** (Deuxième version qui optimise l'utilisation de la mémoire)
 * calcule la génération suivante (#règle 126) pour simuler le parallélisme il faut
 * nécessairement utiliser un autre tableau
 */
  void generationSuivante() {
    for (int i=0; i<this.courante.length; i+=1) {
      int v = this.getVoisinage(i);
      this.suivante[i] = (v!=0 && v!=3);
    }
    boolean[] tmp = this.courante;
    this.courante = this.suivante;
    this.suivante = tmp;
  }
```



```
/** permet à une autre classe de récupérer l'automate pour l'afficher
 * @return tableau booléen qui représente la génération courante, true=active
 */
  boolean[] getCourante() {
    return this.courante;
  }
/** @param pos numéro de la cellule dont on veut calculer les voisins
 * @return code de Wolfram pour le voisinage de la cellule pos
 */
  int getVoisinage(int pos) {
    assert(pos>=0 && pos<this.courante.length);
    int nb = 0;
    if (pos>0 && this.courante[pos-1]) nb += 4;  //gauche (poids 4)
    if (pos<this.courante.length-1
        && this.courante[pos+1]) nb += 1; //droite (poids 1)
    if (this.courante[pos]) nb += 2; // centre (poids 2)
    return nb;
  }
```


Le Jeu de la Vie

♦Créé en 1970 par John Horton Conway, mathématicien à Cambridge, à la suite de travaux de John Von Neumann sur les fondements théoriques de la programmation : modéliser une « machine » universelle, modèle mathématique d'un ordinateur abstrait (cf. cours de 3ème et 4ème année…).

♦Il a ouvert la voie à la théorie des « automates cellulaires » développée notamment par le physicien Steven Wolfram.

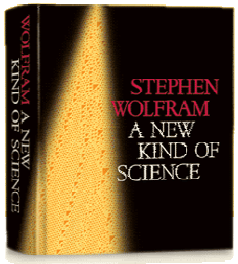
♦Références :

- *http://fr.wikipedia.org/wiki/Automate_cellulaire*
- *http://www.vieartificielle.com/index.php?action=article&id=82*

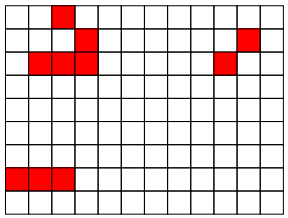
"A new kind of Science", http://www.wolframscience.com/thebook.html

- *http://www.bitstorm.org/gameoflife*
- *http://www.univ-lemans.fr/enseignements/physique/02/divers/conway.html*

« The Recursive Universe », W. Poundstone, Oxford Univ. Press, 1985



♦Le Jeu de la Vie est un **automate cellulaire** constitué d'une population de cellules dans un plan grillagé.



Les 8 voisins possibles d'une cellule

♦Cette population va évoluer, pour passer d'une génération à la génération suivante, d'après deux règles :

- si une cellule est libre, elle devient occupée **(naissance)** si elle a exactement 3 voisins.
- si une cellule est occupée, elle devient libre **(mort)** si son nombre de voisins est ≥ 4 **(étouffement)** ou ≤ 1 **(solitude)**.