



**HAL**  
open science

## Algorithme Avancé

Carine Souveyet, Manuele Kirsch Pinheiro

► **To cite this version:**

Carine Souveyet, Manuele Kirsch Pinheiro. Algorithme Avancé. Licence. Algorithme Avancé, Paris (France), France. 2016. hal-03996345

**HAL Id: hal-03996345**

**<https://paris1.hal.science/hal-03996345>**

Submitted on 19 Feb 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



UNIVERSITÉ PARIS 1

PANTHÉON SORBONNE

---

# INF6

# Algorithmique Avancée

Carine Souveyet

Manuele Kirsch Pinheiro

---

# Contenu prévisionnel

---

- Piles et files
- Listes
- **Récurtivité**
  - **Récurtivité dans le calcul**
  - **Récurtivité structurelle**
- Arbres binaires
  - Parcours en profondeur et en largeur
- Généralisation de la notion d'arbre
  - Insertion et suppression de nœuds
- Arbre de recherche
  - Rééquilibrage
- Graphes



UNIVERSITÉ PARIS 1

PANTHÉON SORBONNE

---

# RECURSIVITE DANS LE CALCUL

---

# RECURSIVITE

---

- Une fonction ou une procédure qui s'appelle elle-même est dite *réursive*.

- **Factoriel :  $n!$**

- $n \geq 0$   $\text{fact}(n) = n * \text{fact}(n-1)$  et  $\text{fact}(0) = 1$

- **Suite récurrente (mathématique) :**

- $n > 0$  et  $U(n) = n * U(n-1)$  et  $U(0) = 1$

- **Factoriel :  $n!$**

- $n \geq 0$   $\text{fact}(n) = n * \text{fact}(n-1)$  et  $\text{fact}(0) = 1$

- **Suite récurrente :**

- $n > 0$  et  $U(n) = n * U(n-1)$  et  $U(0) = 1$

- $n=3 \Rightarrow U(3) = 3 * U(2)$

- $\Rightarrow U(2) = 2 * U(1)$

- $\Rightarrow U(1) = 1 * U(0)$

- $\Rightarrow U(0) = 1$

- $\Rightarrow U(3) = 3 * 2 * 1 * 1$

- $\Rightarrow U(3) = 6$

- **Factoriel : n!**

- $n \geq 0$  **fact**(n)=n\***fact**(n-1) et **fact** (0)=1

- **Fonction** double **fact** (int n)

- si**  $n < 0$  => **ArithmeticException**

- sinon**

- si** ( $n == 0$ ) alors => retourne 1 // **arrêt de la récursion**

- sinon** => retourne  $n * \mathbf{fact}(n-1)$

Appel à fact(**3**)

. 3\*fact(**2**) = ?

. Appel à fact(**2**)

. . 2\*fact(**1**) = ?

. . Appel à fact(**1**)

. . . 1\*fact(**0**) = ?

. . . Appel à fact(**0**)

. . . Retour de la valeur **1**

. . . 1\*1

. . . Retour de la valeur **1**

. . 2\*1

. Retour de la valeur **2**

. 3\*2

Retour de la valeur **6**

# RECURSIVITE

---

- **Pile *Exécution*** pour un programme en cours d'exécution : emplacement mémoire destiné à mémoriser les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.
- **Gestion LIFO** des appels de méthodes imbriquées
  - **Attention :**
  - Ne pas oublier le ***point terminal*** (test d'arrêt de la récursion)
  - la pile d'exécution a une taille ***limite***.

- **Exercices pour s'entraîner :**
  - **Suite de Fibonacci :**
    - $n \geq 0$   $U(n) = U(n-1) + U(n-2)$  avec  $U(0) = U(1) = 1$
    - Ecrire la fonction de cette suite en Java,
    - Ecrire les test unitaires pour tester la fonction,
    - Simuler l'exemple pour  $n=5$  et simuler le fonctionnement avec une pile d'exécution,
    - Est ce que l'implémentation récursive est performante par rapport à l'implémentation itérative ?

# RECURSIVITE

---

- Puissance  $x^n$ : **Puissance**  $(x,n)=x*$ **Puissance**  $(x,n-1)$ 
  - Ecrire la fonction récursive avec l'optimisation ci-dessous et les tests unitaires Junit
  - Optimisation : si n est pair  $X^n=(X)^{2*(n/2)}$   
si n est pair  $X^n=((X)^2)^{(n/2)}$
  - Montrer en quoi l'implémentation récursive est plus performante que l'implémentation itérative

# RECURSIVITE

– Puissance  $x^n$ : **Puissance**  $(x,n)=x*$ **Puissance**  $(x,n-1)$

- Optimisation : si n est impair  $X^n=X*(X^{(n-1)})$

Fonction **Puissance** (int x, int n)

si  $n < 0 \Rightarrow$  ArithmeticException

sinon si  $x == 0 \Rightarrow$  retourne 0 //  $0^n = 0$

sinon si  $x == 1 \Rightarrow$  retourne 1 //  $1^n = 1$  évite la récursion

sinon si  $n == 0 \Rightarrow$  retourne 1 //  $x^0 = 1$

sinon si n est pair  $\Rightarrow$  retourne **Puissance** $(x^2, n/2)$

si n est impair  $\Rightarrow$  retourne  $x*$ **Puissance** $(x, n-1)$



UNIVERSITÉ PARIS 1

PANTHÉON SORBONNE

---

# RECURSIVITE STRUCTURELLE

---

# RECURSIVITE – PILE (1)

---

- Une structure de données est dite *réursive* lorsque l'un des éléments composants est du type de cette structure.
  - **PileR** <élément sommet, PileR>
  - **Interface de Pile** : LIFO
    - boolean : **estVide()**
    - void **empiler** (element)
    - void **depiler** () // est défini pour une pile non vide
    - element **getSommet()** // est défini pour une pile non vide

# RECURSIVITE – PILE (2)

---

- **PileR** <sommet : élément, p : PileR>
- **estVide()** : **pas réursive**
  - si (sommet==null) => retourne true
  - sinon => retourne false
- element **getSommet()** : **pas réursive et Pile non vide**
  - si (pile vide) alors => **PileVideException**
  - sinon => retourne sommet.

# RECURSIVITE – PILE (3)

---

– PileR <sommet : élément, p : PileR>

– empiler(element e) : **récursive**

3 cas :

**pile vide :**

sommet=e

**pile avec au moins un élément :**

si (p==null) alors //un 1 élément

=> p = nouvelle pile

**p.empiler(sommet)**

sommet=e

# RECURSIVITE – PILE (4)

---

– **PileR <sommet : élément, p : PileR>**

– **depiler () : réursive et Pile non vide**

**4 cas :**

**Pile vide => PileVideException**

**Pile avec un élément**

**sommet=null**

**Pile avec avec 2 éléments au minimum**

**sommet=p.getSommet()**

**p.depiler()**

**si p.estVide() => p=null**

---

# RECURSIVITE – FILE (1)

---

– **FileR** <premier : élément , f : FileR>

– **Interface de File: FIFO**

- boolean : **estVide()**
- void **enfiler** (element)
- void **defiler** () // Pile non vide
- element **getPremier()** // Pile non vide

# RECURSIVITE – FILE (2)

---

- **FileR** <premier : élément , f : FileR>
- boolean : **estVide()** : **pas récursive**
  - si (premier==null) => retourne true
  - sinon => retourne false
- element **getPremier()** : **pas récursive et Pile non vide**
  - si (file vide) alors => FileVideException
  - sinon => retourne premier.

# RECURSIVITE – FILE (3)

---

- **FileR** <premier : élément , f : FileR>
- void **enfiler** (element) : **récursive**

**3 cas :**

**file vide :**

**premier=e**

**file avec au moins un élément :**

**si (f==null) alors // un seul élément**

**=> f= nouvelle file**

**f.enfiler (element)**

# RECURSIVITE – FILE (2)

---

- **FileR** <premier : élément , f : FileR>
- void **defiler()** : **récursive et File Non Vide**

**4 cas :**

**file vide : => FileVideException**

**file avec un élément :**

**premier=null**

**file avec au moins deux éléments :**

**premier=f.getPremier()**

**f.defiler()**

**si f.estVide() => f=null**

– **ListeR** <élément premier , ListeR>

– **Interface de Liste :**

- boolean : **estVide()**
- Int **longueur()**
- void **ajouter** (element, int i) // indice valide
- void **supprimer**(int i) // Pile non vide et indice valide
- element **getPremier()** // Pile non vide
- element **getDernier()** // Pile non vide
- element **getElement**(int i) // Pile non vide et indice valide

- **Exercice à rendre:**
  - Implémenter à l'aide d'une classe paramétrée le type abstrait Liste et de manière *récursive*
  - Développer les *tests unitaires* pour valider l'implémentation de votre classe