Notes du cours d'algorithmique (ALG2)

Nathan Furnal

29 mai 2021

Table des matières

1 Introduction

La majorité de l'information reprise dans ces notes se trouve dans le syllabus d'algorithmique.

2 Tableaux à deux dimensions

2.1 Rappels

Là où une variable peut contenir une valeur en mémoire, un tableau (*array*) peut en contenir plusieurs. Chaque cellule du tableau contient une valeur et est caractérisée par un **indice**. L'indice commence à 0, jusqu'à $n-1$ où n est la longueur du tableau.

Toutes les valeurs du tableau ont le même type et une fois crée, on ne peut plus changer sa taille.

Voici un exemple de création de tableau d'entier de taille 5.

```java
int[] tabEntier = new int[5]; // Tableau vide de 5 entiers
int[] autreTab = {1, 2, 3, 4, 5}; // Tableau rempli de 5 entiers
```

2.2 Tableaux à 2D

Dans le cas du tableau à deux dimensions, on précise deux indices, un pour les lignes et un pour les colonnes. Grâce à ces indices, on peut représenter des coordonnées en deux dimensions dans le tableau.

Voici une représentation conceptuelle de tableau en deux dimensions. Il a 2 dimensions, 3 lignes et 4 colonnes et donc 12 entrées dans le tableau.

	0	**1**	**2**	**3**
0				
1			7	
2				

En java, on l'écrira :

```java
int[][] tab = new int[3][4];
tab[1][2] = 7;
```

2.2.1 Exercices

1. Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et qui retourne un booléen indiquant si la case désignée contient ou pas la valeur nulle.

```java
public class CasNul {

    public static boolean arrayHasZero(int[][] tab, int row, int col){
        return tab[row][col] == 0;
    }

    public static void main(String[] args) {
        int[][] tab = {{1, 2, 3, 0}, {0, 1, 1, 0}};
        System.out.println(arrayHasZero(tab, 0, 0)); // false
        System.out.println(arrayHasZero(tab, 0, 3)); // true
    }

}
```

2. Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que des coordonnées et qui retourne un booléen indiquant si ces coordonnées désignent bien une case du tableau, c'est-à-dire qu'elles sont dans le bon intervalle (0 à n-1 et 0 à m-1).

```java
public class CasExiste {

    public static boolean arrayHasCell(int[][] tab, int row, int col){
        int rows = tab.length;
        int cols = tab[0].length;
        return row >= 0 && col >= 0 && row < rows && col < cols;
    }
    public static void main(String[] args) {
        int[][] tab = {{1, 2, 3}, {10, 11, 12}, {3, 2, 1}};
        System.out.println(arrayHasCell(tab, 0, 0)); //true
        System.out.println(arrayHasCell(tab, 5, 5)); // false
    }
}
```

3. Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne) et une valeur entière. L'algorithme met la valeur donnée dans la case indiquée pour autant que la case contienne actuellement la valeur nulle. Dans le cas contraire, l'algorithme ne fait rien.

```java
import java.util.Arrays;
public class AssignerUneCase {

    public static void mutateZeroCellOfArray(int[][] tab, int row, int col, int
     ↪  val){
        if (tab[row][col] == 0) {
            tab[row][col] = val;
        }
    }

    public static void main(String[] args) {
        int[][] tab = {{1, 2, 3, 4}, {0, 1, 2, 3}};
        mutateZeroCellOfArray(tab, 0, 0, 11); // does nothing
        mutateZeroCellOfArray(tab, 1, 0, 11); // modifies array
        System.out.println(Arrays.deepToString(tab));     // modified array
    }
}
```

4. Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un bord du tableau.

```java
public class BordDuTableau {
    public static boolean isBorder(int[][] tab, int row, int col){
        int top = 0;
        int down = tab.length - 1;
        int left = 0;
        int right = tab[0].length - 1;

        return col == left || col == right || row == top || row == down;

    }
    public static void main(String[] args) {
        int[][] tab  = {{1, 2, 3, 4}, {0, 1, 0, 1}, {11, 11, 11, 11}};
        System.out.println(isBorder(tab, 0, 3)); // true
        System.out.println(isBorder(tab, 1,  2)); // false
        System.out.println(isBorder(tab, 2, 3)); // true
        System.out.println(isBorder(tab, 1, 1)); // false
    }
}
```

5. Écrire un algorithme qui reçoit un tableau d'entiers (à n lignes et m colonnes) ainsi que les coordonnées d'une case (ligne, colonne). L'algorithme doit indiquer si la case donnée est ou non sur un des 4 coins du tableau.

```java
public class CoinDuTableau {

    public static boolean isCorner(int[][] tab, int row, int col){
        int top = 0;
        int down = tab.length - 1;
        int left = 0;
        int right = tab[0].length - 1;
        return (row == top && (col == left || col == right))
                || (row == down && (col == left || col == right));
    }

    public static void main(String[] args) {
        int[][] tab = {{1, 2, 3, 4}, {3, 2, 1, 0}, {0, 0, 0, 0}, {1, 1, 1, 1}};
        System.out.println(isCorner(tab, 0, 0)); // true
```

```
        System.out.println(isCorner(tab, 3, 0)); // true
        System.out.println(isCorner(tab, 2, 2)); // false
        System.out.println(isCorner(tab, 3, 3)); // true
    }
}
```

2.3 Parcours

On s'intéresse à comment on peut parcourir un tableau, pour chercher des valeurs par exemple. On va d'abord parcourir une dimension, ligne ou colonne. Ensuite on verra comment parcourir en deux dimensions. Pour parcourir, on va utiliser l'itération, c'est-à-dire une boucle `for`. On peut aussi utiliser le `foreach`, une itération où on utilise les éléments directement plutôt que d'accéder aux éléments via leur index.

2.3.1 Parcours d'une dimension

Pour parcourir une ligne, on va fait changer le numéro de colonne et pour parcourir une colonne, on fait varier le numéro de ligne. Voici quelques exemples.

```
public static void afficherLigne(int[][] tab, int lg) {
    for (int col = 0; col < tab[0].length; col++) {
    System.out.println(tab[lg][col]);
    }
}
```

```
public static void afficherColonne(int[][] tab, int col) {
    for (int lg = 0; lg < tab.length; lg++) {
        System.out.println(tab[lg][col]);
    }
}
```

Plus précisément, on accède un tableau à deux dimensions, on "bloque" une ligne qu'on décide de parcourir ou on "bloque" une colonne qu'on décide de parcourir.

On peut aussi parcourir les diagonales des tableaux carrés, qui ont le même nombre de lignes et de colonnes. La différence avec les exemples précédent est qu'on accède à **deux** index en même temps.

Voici un exemple avec la première diagonale.

```
public static void afficherDiagonaleDescendante(int[][] tab) {
    for (int i = 0; i < tab.length; i++) {
        System.out.println(tab[i][i]);
    }
}
```

Voici un exemple, plus complexe, avec la seconde diagonale. Ici on doit se méfier parce qu'on commence de 0 au niveau des lignes mais on doit commencer par la dernière colonne. Donc, on aura une incrémentation des lignes mais une décrémentation des colonnes.

```
// Version avec deux variables
public static void afficherDiagonaleMontanteV1(int[][] tab) {
    int col = tab[0].length - 1;
    for (int lg = 0; lg < tab.length; lg++) {
        System.out.println(tab[lg][col]);
        col--;
    }
}

// En Java, on peut placer les 2 variables dans le for
public static void afficherDiagonaleMontanteV2(int[][] tab) {
    for (int lg = 0, col = tab[0].length - 1; lg < tab.length; lg++, col--) {
```

```
        System.out.println(tab[lg][col]);
    }
}
```

2.3.2 Parcours des deux dimensions

Dans les exemples précédents, on parcourait en une dimension, c'est-à-dire qu'on définissait une ligne ou une colonne sur lesquelles itérer et puis seulement on les parcourait. Ici, on va visiter une tableau case par case. Ce qui nécessite d'accéder aux deux dimensions et deux boucles `for`, donc on parlera de parcours à deux dimensions.

Voici un exemple des parcours, avec une stratégie à deux dimensions, on imprime ici chaque cellule.

Pour les lignes :

```
public static void afficherLigneParLigne(int[][] tab) {
    for (int lg = 0; lg < tab.length; lg++) {
        for (int col = 0; col < tab[0].length; col++) {
            System.out.println(tab[lg][col]);
        }
    }
}
```

Pour les colonnes :

```
public static void afficherColonneParColonne(int[][] tab) {
    for (int col = 0; col < tab[0].length; col++) {
        for (int lg = 0; lg < tab.length; lg++) {
            System.out.println(tab[lg][col]);
        }
    }
}
```

Mais on est pas forcément limité par cette dernière stratégie, en effet, on peut aussi utiliser la taille du tableau $(n * m)$ pour identifier le nombre d'éléments et puis itérer.

```
public static void afficherLigneParLigneV2(int[][] tab) {
    int nbÉléments = tab.length * tab[0].length;
    int lg = 0;
    int col = 0;
    for (int i = 0; i < nbÉléments; i++) {
        System.out.println(tab[lg][col]);
        col++;
        if (col == tab[0].length) {
            col = 0;
            lg++;
        }
    }
}
```

2.3.3 Interruption du parcours

Dans certains cas, on voudra interrompre le parcours du tableau, quand on rencontre une certaine condition par exemple. On préféra alors utilise un parcours avec `while`, en prenant soin de correctement placer l'interruption. Dans les exemples suivants, on imagine la situation où on cherche un élément particulier au sein d'un tableau.

```
public static boolean chercherLigneParLigneV1(int[][] tab, int elt) {
    int lg, col;
    boolean trouvé;
    trouvé = false;
    lg = 0;
    while (lg < tab.length && !trouvé) {
```

```
        col = 0;
        while (col < tab[0].length && !trouvé) {
            if (tab[lg][col] == elt) {
                trouvé = true;
            } else {
                col++;
            }
        }
        if (!trouvé) {
            lg++;
        }
    }
    return trouvé;
}
```

Il existe une version de cette algorithme avec une seule boucle, la ligne et la colonne s'arrêtent exactement au bon endroit.

```
public static boolean chercherLigneParLigneV2(int[][] tab, int elt) {
    int nbÉléments = tab.length * tab[0].length;
    int lg = 0;
    int col = 0;
    int i = 1;
    while (i <= nbÉléments && tab[lg][col] != elt) {
        col++;
        if (col == tab[0].length) {
            col = 0;
            lg++;
        }
        i++;
    }
    return i <= nbÉléments;
}
```

Enfin, un exercice plus compliqué est présenté, avec un parcours en serpent, il est entièrement détaillé dans le syllabus.

2.3.4 Exercices

1. Affichage

```
public class AfficherTableau {
    public static void ligneParLigne(int[][] tab){
        int rows = tab.length;
        int cols  = tab[0].length;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(tab[i][j] + " ");
            }
            System.out.println();
        }
    }
    public static void colonneParcolonne(int[][] tab){
        int rows = tab.length;
        int cols = tab[0].length;
        for (int i = 0; i < cols; i++) {
            for (int j = 0; j < rows; j++) {
                System.out.println(tab[j][i]);
            }
        }
    }
```

```
    public static void main(String[] args) {
        int[][] tab = {{1, 2, 3}, {4, 4, 7}, {9, 8, 1}};
        System.out.println("Impression ligne par ligne");
        System.out.println();
        ligneParLigne(tab);
        System.out.println();
        System.out.println("Impression colonne par colonne");
        System.out.println();
        colonneParcolonne(tab);
    }
}
```

2. Cases adjacentes

```
public class CaseAdjacente {
    public static void printAdjacentCell(int[][] tab, int row, int col) {
        int[] rowPos = {-1, -1, -1, 0, 1, 1, 1, 0}; // all relatives rows
        ↪ starting from top left
        int[] colPos = {-1, 0, 1, 1, 1, 0, -1, -1}; // relative columns
        ↪ starting from top left
        String[] names = {"top left", "top", "top right", "right", "bottom
        ↪ right", "bottom", "bottom left", "left"};
        int n = rowPos.length;
        for (int i = 0; i < n; i++) {
            int r = row + rowPos[i];
            int c = col + colPos[i];
            if (r < 0 || r >= tab.length || c < 0 || c >= tab[0].length) {
                System.out.println("(" + r + ", " + c + ")" + " -> " + "out!");
            } else {
                System.out.println(names[i] + " : (" + r + ", " + c + ")");
            }
        }
    }
}
```

3. Les nuls

```
public class LesNuls {
    public static double nulRatio(int[][] tab){
        int total = 0; // Use total instead of size in case of unequal row
        ↪ lengths
        int count = 0;
        for (int[] rows : tab) {
            for (int cell : rows) {
                if (cell == 0) {
                    count++;
                }
                total++;
            }
        }
        return (double) count / total ;
    }
}
```

4. Tableau de cotes

```
public class TableauCotes {
    public static double passingPercentage(int[][] tab){
        int nStudents = tab.length;
        int countPassed = 0;
        int passingGrade = 10 * tab[0].length; // 10/20 * nbr of grades
        for (int[] row : tab){
            int rowSum = 0;
```

```
            for(int cell : row){
                rowSum += cell;
            }
            if (rowSum > passingGrade){
                countPassed++;
            }
        }
        return (double) 100 * countPassed/nStudents; // convert to percentage
    }
}
```

5. Triangle de Pascal

```
public class TriangleDePascal {
    public static int[][] makeTriangle(int n){
        int[][] arr = new int[n+1][n+1];
        for(int i = 0; i <= n; i++){
            for (int j = 0; j <=i; j++) {
                int topRow = i - 1;
                int leftCol = j - 1;
                if(topRow < 0 || leftCol < 0 || i == j){
                    arr[i][j] = 1;
                }
                else {
                    arr[i][j] = arr[topRow][j] + arr[topRow][leftCol];
                }
            }
        }
        return arr;
    }
}
```

6. Tous positifs

```
public class TousPositifs {
    public static boolean allPositive(int[][] arr){
        // return statement breaks loop
        for(int[] row : arr){
            for(int val : row){
                if(val <= 0){
                    return false;
                }
            }
        }
        return true;
    }

    public static boolean allPositiveWhile(int[][] arr){
        int row = 0;
        int col = 0;
        while (row < arr.length && arr[row][col] > 0){
            col++;
            if (col == arr[0].length){
                row++;
                col = 0;
            }
        }
        // The row index of arr.length is reached only if all are positive
        return row == arr.length;
    }
}
```

7. Toute une ligne de valeurs non nulles

```java
public class TousPositifsLignesColonnes {
    public static boolean positiveRow(int[][] arr, int row) {
        for (int elem : arr[row]){
            if (elem <= 0) {
                return false;
            }
        }
        return true;
    }

    public static boolean positiveCol(int[][] arr, int col){
        for (int i = 0; i < arr.length; i++) {
            if (arr[i][col] <= 0){
                return false;
            }
        }
        return true;
    }
}
```

8. Carré magique
   Attention, ce code peut échouer si chaque somme de ligne est égale à chaque somme de colonne pour chaque ligne et colonne. Ce qui ne devrait pas arriver mais est un cas extrême qui fait échouer l'algorithme.

```java
import java.util.Arrays;
public class CarreMagique {
    public static boolean isMagicSquare(int[][] arr){
        int n = arr.length;
        int[] rowSums = new int[n];
        int[] colSums = new int[n];
        int firstDiagSum = 0;
        int secondDiagSum = 0;
        for (int i = 0; i < n; i++) {
            int k = n - 1 - i;
            firstDiagSum += arr[i][i];
            secondDiagSum += arr[i][k];
            for (int j = 0; j < n; j++) {
                rowSums[i] += arr[i][j];
                colSums[j] += arr[i][j];
            }
        }
        return Arrays.equals(rowSums, colSums) && firstDiagSum ==
         ↪  secondDiagSum;
    }

    public static void main(String[] args) {
        int[][] tab = {{2, 7, 6},
                       {9, 5, 1},
                       {4, 3, 8}};
        int[][] tab2 = {{4, 7, 6},
                        {9, 8, 1},
                        {2, 3, 5}};
        System.out.println(isMagicSquare(tab)); // true
        System.out.println(isMagicSquare(tab2)); // false
    }
}
```

Voici une version qui ne devrait pas poser le même problème :

```java
public static boolean isMagicSquare(int[][] arr){
    int n = arr.length;
    int[] rowSums = new int[n];
    int[] colSums = new int[n];
```

```
    int firstDiagSum = 0;
    int secondDiagSum = 0;
    for (int i = 0; i < n; i++) {
        int k = n - 1 - i;
        firstDiagSum += arr[i][i];
        secondDiagSum += arr[i][k];
        for (int j = 0; j < n; j++) {
            rowSums[i] += arr[i][j];
            colSums[j] += arr[i][j];
        }
    }
    int check = rowSums[0];
    if (firstDiagSum != secondDiagSum || check != firstDiagSum){
        return false;
    }
    for (int i = 0; i < n; i++) {
        if (rowSums[i] != check || colSums[i] != check)
            return false;
    }
    return true;
}
```

3 L'orienté-objet

On peut se poser la question de pourquoi on utilise l'orienté-objet en algorithmique. C'est parce qu'on étudie les structures de données et leurs implémentations. De ce fait, on a besoin des notions d'orienté-objet pour implémenter et utiliser les structures de données nécessaires.

Donc, on peut utiliser les membres d'une classe pour accéder aux données (attributs) et appliquer des changements à ces données (méthodes). Aussi, l'orienté-objet n'est pas unique à Java, on le retrouve dans plusieurs autres langages, c'est une méthode de représentation.

L'entièreté des exemples et du code sont repris dans le chapitre 2 du syllabus, je ne les copierai pas ici mais reprendrai quelques explications.

De manière générale, chaque composante qui a un groupe de comportement à part entière peut être représentée sous forme d'objet. Ici, on prend l'exemple d'animaux, de jeux, etc. On peut voir chaque animal et son comportement propre comme une classe avec ses attributs et méthodes. Comme on peut voir un tableau de jeu ou bien un élément de jeu comme un objet propre. Cette manière de grouper les attributs et méthodes semblables dans un seul objet est typique de l'orienté objet.

Malgré tout, l'orienté-objet n'est pas le point final, c'est une forme de représentation qu'on doit utiliser à bon escient. La section suivante propose une méthode appelée Modèle-Vue-Contrôleur (MVC).

3.1 Commentaire sur le MVC (Modèle-Vue-Contrôleur)

Dans l'approche MVC, on découpe le code en trois parties. Le **modèle**, qui utilise et manipule les objets à proprement parler pour obtenir le comportement voulu. C'est ici qu'on crée un plateau de jeu et les pions ou les cartes quand on construit un jeu.

La **vue**, qui gère l'interaction avec l'utilisateur et l'affichage. C'est ici qu'on fera les demandes de lecture ou l'affichage du jeu.

Le **contrôleur**, qui lie la vue et le modèle et qui orchestre le code déjà écrit.

4 La liste

La liste est une structure de donnée de taille variable et de type homogène en Java. Elle est plus flexible que le tableau, surtout dans le cas où ne connaît pas à l'avance les tailles nécessaires pour recevoir les données. De plus, elle implémente des méthodes utiles.

4.1 Interface `List`

Une liste est une **interface**. C'est-à-dire qu'elle fournit des méthodes qu'il faut implémenter mais qu'on ne l'instancie pas. On utilisera généralement des implémentations comme `ArrayList` ou `LinkedList`. De plus, on utilise des *wrappers* avec les listes car celles-ci ne peuvent contenir que des objet et non pas des types primitifs. Les *wrappers* Ce sont des classes utilitaires qui enveloppent les types primitifs habituels pour leur donner plus de fonctionnalités ; comme `Integer` pour les `int` ou `Character` pour les `char`.

4.2 Manipulation de base et affichage

Voyons quelques exemples minimaux pour illustrer le fonctionnement des listes.

```java
import java.util.List;
import java.util.ArrayList;

List<Integer> myList = new ArrayList<>(); // vide
myList.add(1);                  // [1]
myList.add(4);                  // [1, 4]
myList.add(7);                  // [1, 4, 7]
myList.set(1, 33);              // [1, 33, 7]
myList.remove((int) 0);         // [33, 7] :: Enlève à l'index
myList.remove((Integer) 7);     // [33]    :: Enlève la valeur
myList.clear();                 // []      Vide la liste
```

On peut aussi accéder aux éléments d'une liste et itérer sur une liste.

```java
List<Integer> myList = new ArrayList<>();
myList.add(11);
myList.add(22);
myList.add(33);
Integer x = myList.get(1);  // 22
for(Integer el : myList)
    System.out.println(el); // Imprime les éléments de la liste

myList.contains(2);         // false
myList.contains(11);        // true
```

Comme toujours, le détail de toutes les méthodes se trouvent dans la Javadoc.

4.3 Exercices

Dans les exercices qui suivent, j'utilise fréquemment l'expression suivante :

```java
import java.util.Arrays;
import java.util.ArrayList;

List<Integer> myList = new ArrayList<>(Arrays.asList(1, 2, 4, 5)); // Par exemple
```

L'utilisation `Arrays.asList()` me permet de créer rapidement une liste non mutable. Ensuite, elle est simplement mise dans une `ArrayList` qui elle, est mutable. C'est donc une simplification pour créer une liste. À noter qu'insérer :

```java
import java.util.List;
import java.util.ArrayList;
var list = new ArrayList<>(List.of(1, 2, 3, 4, 5));
```

fonctionne tout aussi bien. Attention, `List.of()` et `Arrays.asList()` ne sont pas des expressions équivalentes mais pour donner peupler une `ArrayList`, je n'ai pas rencontré de problème.

J'utilise aussi le mot-clé `var` qui me permet d'inférer le type des variables, son fonctionnement est décrit dans le cours de DEV2.

4.3.1 Manipulation d'une liste

```java
import java.util.ArrayList;
public class ManipList {

    public static void makeList() {
        var list = new ArrayList<>();
        list.add(494);
        list.add(209);
        list.add(425);
        System.out.println("Taille de liste : " + list.size());
        System.out.println("La liste contient 425 : " + list.contains(425));
        list.remove((Integer) 209);
        list.add(0, 101);
        System.out.println(list);
    }

    public static void main(String[] args) {
        makeList();
    }
}
```

4.3.2 Liste des premiers entiers

```java
import java.util.ArrayList;
import java.util.List;

public class PremiersEntiers {
    public static List<Integer> makeDecreasing(int n){
        var list = new ArrayList<Integer>();
        for (int i = 1; i <= n; i++) {
            list.add(0, i);
        }
        return list;
    }

    public static void main(String[] args) {
        var l = makeDecreasing(10);
        System.out.println(l);
    }
}
```

4.3.3 Somme d'une liste

```java
import java.util.List;

public class SumList {
    public static int sumList(List<Integer> list){
        int sum = 0;
        for(Integer el : list){
            sum += el;
        }
        return sum;
    }

    public static void main(String[] args) {
        var arr = List.of(1, 2, 3, 4, 5, 6);
        System.out.println(sumList(arr));
    }
}
```

4.3.4 Anniversaires

Cet exercice nécessitait de créer une classe `Personne` ainsi qu'une liste de personnes. Je met ici mon implémentation suivi de la solution.

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Personne {
    private final String firstName;
    private final String lastName;
    private final int birthMonth;

    public Personne(String firstName, String lastName, int birthMonth){
        this.firstName = firstName;
        this.lastName = lastName;
        if(birthMonth <= 0 || birthMonth > 12){
            throw new IllegalArgumentException("Month should be a value between 1 and
             ↪  12 " + birthMonth);
        }
        this.birthMonth = birthMonth;
    }

    public int getBirthMonth() {
        return birthMonth;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName + " " + birthMonth;
    }

    public static List<Personne> makePeople(){
        var p1 = new Personne("Johnny", "Boy", 5);
        var p2 = new Personne("Alice", "Randal", 8);
        var p3 = new Personne("Mojo", "Jojo", 3);
        var p4 = new Personne("Sally", "Brown", 11);
        var p5 = new Personne("Elton", "John", 4);
        var p6 = new Personne("Bob", "Ross", 12);
        var p7 = new Personne("Frank", "Boyle", 5);
        var p8 = new Personne("Angela", "Merkle", 4);
        var p9 = new Personne("Franky", "Frank", 7);
        return new ArrayList<>(Arrays.asList(p1, p2, p3, p4, p5, p6, p7, p8, p9));
    }
}
```

Ici la solution sélectionne ceux dont le mois de naissance est 4, soient Elton John et Angela Merkle.

```java
import java.util.ArrayList;
import java.util.List;

public class MoisAnniversaire {
    public static List<Personne> findBirthMonth(List<Personne> people, int
     ↪  birthMonth){
        var out = new ArrayList<Personne>();
        for(Personne p : people){
            if(p.getBirthMonth() == birthMonth){
                out.add(p);
            }
        }
        return out;
    }
```

```
14
15      public static void main(String[] args) {
16          var people = Personne.makePeople();
17          var out = findBirthMonth(people, 4);
18          System.out.println(out);
19      }
20  }
```

4.3.5 Concaténation de deux listes

```
1   import java.util.ArrayList;
2   import java.util.Arrays;
3   import java.util.List;
4
5   public class ConcatList {
6       public static List<Integer> concatList(List<Integer> l1, List<Integer> l2) {
7           for (Integer el : l2) {
8               l1.add(el);
9           }
10          return l1;
11      }
12
13      public static void main(String[] args) {
14          var l1 = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 7));
15          var l2 = new ArrayList<>(Arrays.asList(11, 33, 44, 77));
16          System.out.println(concatList(l1, l2));
17      }
18  }
```

4.3.6 Le nettoyage

```
1   import java.util.ArrayList;
2   import java.util.Arrays;
3   import java.util.List;
4
5   public class Nettoyage {
6       public static int cleanStrings(List<String> strings, String s){
7           int len = strings.size();
8           while(strings.contains(s)){
9               strings.remove(s);
10          }
11          return len - strings.size();
12      }
13
14      public static void main(String[] args) {
15          var strings = new ArrayList<>(Arrays.asList("hello", "hey", "hey", "ola",
            ↪  "bjr", "ok"));
16          System.out.println(cleanStrings(strings, "hey"));
17      }
18  }
```

4.3.7 Les extrêmes

```
1   import java.util.ArrayList;
2   import java.util.Arrays;
3   import java.util.List;
4
5   public class Extrema {
```

```
    public static List<Integer> removeExtrema(List<Integer> l){
        Integer min = l.get(0);
        Integer max = l.get(0);
        for(Integer i : l){
            if(i > max){
                max = i;
            }
            if(i < min){
                min = i;
            }
        }
        l.remove(max);
        l.remove(min);
        return l;
    }

    public static void main(String[] args) {
        var l = new ArrayList<>(Arrays.asList(0, 1, 2, -1, 4, 5, 3, 8, 1, 2, 11, -3,
        ↪  7));
        System.out.println(removeExtrema(l));
    }
}
```

4.3.8 Fusion de deux listes

Cette solution est probablement la plus compliquée, elle vient de de Stackoverflow.

En pratique, on détermine la taille finale de la liste qui de retour, qui vaut la somme des tailles des listes individuelles. Puis, on itère sur cette taille. On met deux variables à 0 qui nous permettent de tenir le compte des indices de chacune des deux listes qu'on reçoit.

Ensuite, on itère et on vérifie si on a atteint la fin de la première liste. Si c'est le cas, on ajoute à la réponse la valeur correspondante de l'**autre** liste et puis on incrémente son index. On fait exactement la même vérification pour la seconde liste.

Si on a atteint le bout d'aucune des deux listes entrées, on compare les deux listes à l'endroit où on est arrivé pour chacune d'elle car il n'est pas forcément le même. Si la première liste à une valeur plus faible à l'emplacement j, on ajoute sa valeur à la réponse. Si c'est plutôt la seconde liste qui a une valeur plus faible à l'emplacement k, alors on ajoute sa valeur à la réponse.

On comprend donc qu'une fois qu'une des deux listes est épuisée, on ajoute constamment les valeurs de l'autre jusqu'à la fin.

Attention, cette solution utilise aussi l'incrémentation. Il faut réaliser que lorsque qu'une variable est suivie de ++, c'est bien sa valeur avant l'incrémentation qui est utilisée et puis seulement elle est incrémentée. Cette subtilité est traitée dans le cours de DEV2 aussi.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class FusionListes {
    public static List<Integer> mergeSortedLists(List<Integer> l1, List<Integer> l2){
        var sorted = new ArrayList<Integer>();
        int finalSize = l1.size() + l2.size();
        int j = 0, k = 0;
        for (int i = 0; i < finalSize; i++) {
            if(j == l1.size()){
                sorted.add(l2.get(k++));
            }
            else if(k == l2.size()){
                sorted.add(l1.get(j++));
            }
            else if (l1.get(j).compareTo(l2.get(k)) < 0){
                sorted.add(l1.get(j++));
```

```
            }
            else {sorted.add(l2.get(k++));}
        }
        return sorted;
    }
    public static void main(String[] args) {
        var l1 = new ArrayList<>(Arrays.asList(1, 3, 7, 7));
        var l2 = new ArrayList<>(Arrays.asList(3,  9));
        var out = mergeSortedLists(l1, l2);
        System.out.println(out);
    }
```

4.3.9 Éliminer les doublons d'une liste

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class EliminerDoublon {
    // Partie 1
    public static List<Integer> removeDuplicates(List<Integer> l){
        var uniques = new ArrayList<Integer>();
        for(Integer el : l){
            if(!uniques.contains(el)){
                uniques.add(el);
            }
        }
        return uniques;
    }
    // Partie 2
    public static void removeDuplicatesInPlace(List<Integer> l){
        var seen = new ArrayList<Integer>();
        for (int i = 0; i < l.size(); i++) {
            if(seen.contains(l.get(i))){
                l.remove(i);
                i--; // Sinon la liste "avance" d'un élément de trop car elle
                 ↪  rétrécit
            }
            else {seen.add(l.get(i));}
        }
    }

    public static void main(String[] args) {
        var l = new ArrayList<>(Arrays.asList(1, 3, 3, 7, 8, 8, 11, 13, 11, 11));
        var other = new ArrayList<>(Arrays.asList(1, 3, 4, 4, 8, 8, 11, 13, 11, 11));
        var out  = removeDuplicates(l);
        removeDuplicatesInPlace(other);
        System.out.println(out);
        System.out.println(other);
    }
}
```

5 Les traitements de rupture

Dans ce chapitre, nous allons aborder une algorithme qui permet de résoudre un ensemble de problèmes, à travers des données classées. On parlera d'algorithme de rupture. Pour pouvoir parler de **rupture**, on doit d'abord définir les classements d'éléments.

5.1 Le classement complexe

Dans les chapitres précédents, nous avons souvent évoqués des éléments simples à classer comme des nombres ou des chaînes. Ici, nous analyserons plutôt des structures où la relation n'est pas forcément simple à décrire. On verra aussi qu'on traitera des données avec plusieurs **champs** possibles.

Le résumé du syllabus est totalement approprié, je me permet de le remettre ici.

> Les exemples ci-dessus constituent des exemples de **classement complexes**. On dira que des données sont classées sur la **clé composée** champ 1 - champ 2 - ... - champ n (où le i-ème champ est un attribut des données) si le classement se fait prioritairement depuis le champ 1 jusqu'au champ n. Autrement dit, si deux données ont tous leurs champs $1, 2, ..., i$ égaux $i < n$, le classement se fait en départageant sur le champ $i + 1$. L'indice du champ correspond au **niveau** du classement complexe.

5.2 La notion de rupture

5.3 Traitement de rupture dans une séquence ordonnée

5.3.1 Rupture de niveau 0

5.3.2 Rupture de niveau 1

5.3.3 Rupture de niveau 2

5.4 Exercices