VECTEURS EN `ArrayList` – ÉLÉMENTS DE SYNTHÈSE DES CONCEPTS ET DE LA DÉMARCHE ÉTUDIÉS

RAPPELS :

- ACCÈS À UN ÉLÉMENT D'UN VECTEUR EN `ArrayList`
- COMPARAISON DE DEUX ÉLÉMENTS D'UN VECTEUR EN `ArrayList`

FONCTIONS DE RECHERCHE SÉQUENTIELLE DANS UN VECTEUR EN `ArrayList`:

PRINCIPES DE BASE

DÉMARCHE DE CONSTRUCTION DE L'ALGORITHME

EXEMPLE COMPLET

NOTE : ce document n'a pas vocation à remplacer l'étude sérieuse des cours 1 à 6, mais peut vous aider à prendre du recul et, si possible, vous être utile…

RAPPELS

Soit v un `ArrayList<T>`, où T est un **type** classe enveloppe, `String` ou `classe` définie par l'utilisateur

- Accès à un élément du vecteur v - cf. cours 5 – partie 1

```
v.get(i) // i >= 0 et i < v.size()
```

- Comparaison entre un élément du vecteur v et un autre élément de type T

  ✓ Si T est une classe enveloppe - cf. cours 4 – partie 1
  un élément de v peut être comparé avec un autre élément de type T en utilisant les opérateurs :
  <, >, <=, >=, ==, !=

  ✓ Si T est la classe `String` - cf. cours 3
  un élément de v peut être comparé avec un autre élément de type `String` avec la méthode `compareTo` de la classe `String`

  ✓ Si T est une classe définie par l'utilisateur
  - si la classe T est munie d'un ordre naturel (elle implémente alors la classe `Comparable<T>`) :
    un élément de v peut être comparé avec un autre élément de type T avec la méthode `compareTo` de la classe T - cf. cours 6 – partie 1
  - sinon, la comparaison doit se faire en fonction du besoin et en respectant le type des éléments comparés

FONCTIONS DE RECHERCHE SÉQUENTIELLE DANS UN VECTEUR EN `ArrayList`

PRINCIPES DE BASE

LIRE ATTENTIVEMENT L'EN-TÊTE DE LA FONCTION À CODER
- Type du résultat
- Type du vecteur où s'effectue la recherche
- Précondition : contraintes relatives à ses paramètres
- Postcondition : ce qu'elle doit renvoyer

FAIRE DES DESSINS

UTILISER UNE STRUCTURE DE BOUCLE ADAPTÉE POUR PARCOURIR LE VECTEUR DE RECHERCHE
- Si le parcours doit être COMPLET : une boucle `for` peut être employée
- Si le parcours de recherche peut être PARTIEL :
  - S'INTERDIRE UNE BOUCLE `for` + `return` dans le corps de l'itération
  - PRIVILÉGIER UNE BOUCLE `while` et retourner le résultat quand la condition de maintien dans l'itération n'est plus vraie

DÉMARCHE ET CONSTRUCTION DE L'ALGORITHME
(avec parcours du vecteur de la gauche vers la droite)

EXEMPLES DE FONCTIONS

```java
public static boolean existeDansVint(ArrayList<Integer> vInt, int unInt) {
    // {  } => { résultat = vrai si unInt existe dans vInt
    //                      faux, sinon}
```

```java
public float valMin(ArrayList<Float> vFloat) {
    // { vFloat non vide } => { résultat = valeur minimum dans vFloat }
```

```java
public static int indiceDansVStringTrie(ArrayList<String> vStringTrie, String uneString) {
    // { vStringTrie trié croissant } =>
    //    { résultat = indice de la 1ère occurrence de uneString dans vStringTrie,
    //                 -1 si uneString n'est pas dans vStringTrie }
```

```java
public static ArrayList<Point> symetriquesAxeH(ArrayList<Point> vPoint) {
    // { vPoint non vide } =>
    //        { résultat = vecteur contenant les symétriques par rapport
    //          à l'axe horizontal, de chaque Point de vPoint  }
```

```java
public static int rechPremIndSeq(ArrayList<Polar> vPolar, int an, String aut) {
    // {vPolar trié dans l'ordre (annee, auteur)} =>
    //  {*  s'il y a dans vPolar au moins un élément d'année an et d'auteur aut,
    //      résultat = indice du premier de ces éléments
    //   *  sinon, résultat = -1}
    // LA RECHERCHE EST SÉQUENTIELLE !!!
```

ÉTAPE 1 : TOUT SAVOIR SUR LE VECTEUR DANS LEQUEL S'EFFECTUE LA RECHERCHE

Questions ...

Q1. Le vecteur peut-il être vide ?

Q2. Le vecteur est-il ou n'est-il pas trié ?

Q3. Quel est le type des éléments du vecteur (objet de type classe enveloppe, `String` ou autre classe) ?

Q4. Existe-t-il un ordre naturel sur le type des éléments du vecteur ?

Q5. Dans le cas où *le vecteur est trié*, comment sont triés ses éléments ?

5.1. selon l'ordre naturel de leur type, ou selon d'autres critères ?

5.2. par ordre croissant ou par ordre décroissant ?

Pour répondre à ces questions ? LIRE ATTENTIVEMENT LA PRÉCONDITION DE LA FONCTION À CODER

	Précondition	Q1. *Vecteur non vide ?*	Q2. *Vecteur trié ?*	Q3. *Type des éléments ?*	Q4. *Type des éléments du vecteur doté d'un ordre naturel ?*	Q5. *Si le vecteur est trié, comment sont triés ses éléments ?*
FONCTION ❶	*pas d'info.*	pas d'info	?	classe enveloppe	OUI : ordre naturel des types primitifs et classe enveloppe - cours 6_P1 page 3	*sans objet*
FONCTION ❷	*vecteur non vide*	OUI	?	classe enveloppe	OUI : ordre naturel des types primitifs et classe enveloppe - cours 6_P1 page 3	*sans objet*
FONCTION ❸	*vecteur trié croissant*	pas d'info	OUI	`String`	OUI : ordre naturel de la classe `String` - cours 6_P1 page 5	ordre naturel croissant
FONCTION ❹	*vecteur non vide*	OUI	?	`Point`	NON : aucun ordre naturel pour la classe `Point` - cours 4_P2 page 3)	*sans objet*
FONCTION ❺	*vecteur trié ordre(...)*	pas d'info	OUI	`Polar`	OUI : ordre naturel (annee, auteur) de la classe `Polar` - TP6(B)	ordre naturel croissant

ÉTAPE 2 : DESSINER UN VECTEUR (NON VIDE) REPRÉSENTATIF DU VECTEUR DANS SA SITUATION INITIALE

NOTE : pas besoin d'être bon en dessin !...

EXEMPLES DE DESSINS

FONCTION ❶

	0	1	2	3	4	5	6
vInt	5	6	7	8	12	17	9

FONCTION ❷

	0	1	2	3
vFloat	5.2	0.5	7.0	1.8

FONCTION ❸

	0	1	2	3	4	5
vStringTrie	"ananas"	"banane"	"kiwi"	"poire"	"poire"	"pomme"

FONCTION ❹

	0	1	2	3	4
vPoint	3 0	-3 12	0 0	4 -7	25 14

FONCTION ❺

	0	1	2	3	4
vPolar	2002 "ABC" "xcvbn !"	2002 "ABC" "gef"	2002 "GEF" "truc"	2005 "GEF" "machin"	2005 "XYZ" "bidule"

ÉTAPE 3 : PRENDRE CONSCIENCE DU RÉSULTAT QUE DOIT RETOURNER LA FONCTION

Indicateurs : TYPE DU RÉSULTAT, PARAMÈTRES (AUTRES QUE LE VECTEUR DE RECHERCHE) ET POSTCONDITION

	Type du résultat	Autres paramètres	POSTCONDITION : résultat attendu
FONCTION ❶	`boolean`	**`unInt`** (type `int`)	• **`true`** si **`unInt`** est dans le vecteur **`vInt`** • **`false`** sinon
FONCTION ❷	`float`	_	valeur minimum du vecteur **`vFloat`**
FONCTION ❸	`int`	**`uneString`** (type **`String`**)	• indice le plus à gauche de **`uneString`** dans le vecteur **`vStringTrie`** • **`-1`** si **`uneString`** n'est pas dans le vecteur
FONCTION ❹	`ArrayList<Point>`	_	un `ArrayList<Point>` dont les éléments sont les symétriques des éléments de **`vPoint`** par rapport à l'axe horizontal
FONCTION ❺	`int`	- **`an`** (type `int`) - **`aut`** (type **`String`**)	• indice le plus à gauche dans le vecteur **`vPolar`** d'un élément dont l'attribut **`annee`** est égal à **`an`** et dont l'attribut **`auteur`** est égal à **`aut`** • **`-1`** si aucun élément n'a ces caractéristiques

ÉTAPE 4 : DÉTERMINER SI LE PARCOURS EST FORCÉMENT COMPLET OU S'IL PEUT ÊTRE PARTIEL

Indicateurs : PRÉCONDITION ET POSTCONDITION

	Infos sur le vecteur (PRÉCONDITION)	POSTCONDITION : *Quand obtient-on le résultat ?*	PARCOURS
FONCTION ❶	aucune	• dès le départ si **`vInt`** est vide • dès qu'on a trouvé **`unInt`**, ou après avoir étudié tous les éléments de **`vInt`** sans en trouver un dont la valeur est égale à **`unInt`**	PEUT ÊTRE PARTIEL
FONCTION ❷	**`vFloat`** non trié non vide	après avoir étudié tous les éléments de **`vFloat`** pour trouver le minimum	COMPLET
FONCTION ❸	**`vStringTrie`** trié pouvant être vide	• dès le départ si **`vStringTrie`** est vide • dès qu'on a trouvé **`uneString`**, ou dès que le dernier élément examiné dans **`vStringTrie`** est supérieur à **`uneString`** dans l'ordre lexicographique	PEUT ÊTRE PARTIEL
FONCTION ❹	**`vPoint`** non trié non vide	après avoir étudié tous les éléments de **`vPoint`** et ajouté *au fur et à mesure* dans le vecteur qui sera retourné par la fonction, leur symétrique par rapport à l'axe horizontal	COMPLET
FONCTION ❺	**`vPolar`** trié pouvant être vide	• dès le départ si **`vPolar`** est vide • dès qu'on a trouvé dans **`vPolar`** un élément d'attribut **`annee`** égal à **`an`** et d'attribut **`auteur`** égal à **`aut`**, ou dès que le dernier élément examiné dans **`vPolar`** est supérieur dans l'ordre (Annee, Auteur) cf. page 8	PEUT ÊTRE PARTIEL

Expérimenter : JOUER AVEC LES DESSINS PRODUITS À L'ÉTAPE 2

EXEMPLES :

FONCTION ❶

	0	1	2	3	4	5	6
`vInt`	5	6	7	8	12	17	9

`unInt` = **-1** ⇒ parcours COMPLET du vecteur pour savoir **qu'`unInt`** n'est pas dans le vecteur
Résultat : **`false`**

`unInt` = **7** ⇒ parcours PARTIEL du vecteur car **`unInt`** est trouvé à l'indice 2
Résultat : **`true`**

FONCTION ❸

	0	1	2	3	4	5
`vStringTrie`	"ananas"	"banane"	"kiwi"	"poire"	"poire"	"pomme"

`uneString` a pour valeur "fraise" ⇒ parcours PARTIEL du vecteur car à l'indice 2, on trouve "kiwi" supérieur dans l'ordre lexicographique
Résultat : -1

`uneString` a pour valeur "poire" ⇒ parcours PARTIEL du vecteur jusqu'à l'indice 3, on a trouvé…
Résultat : 3

`uneString` a pour valeur "raisin" ⇒ parcours COMPLET du vecteur dont tous les éléments sont inférieurs à "raisin" dans l'ordre lexicographique
Résultat : -1

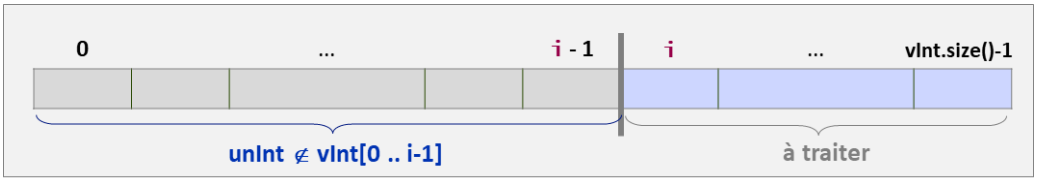
ÉTAPE 5 : CAS D'UN PARCOURS QUI PEUT ÊTRE PARTIEL (EXEMPLES : FONCTIONS ❶ ET ❸)

A – FAIRE UN DESSIN DE LA SITUATION INTERMÉDIAIRE ET TROUVER L'INVARIANT

> L'INVARIANT EST UNE CONDITION (PRÉDICAT) QUI DOIT ÊTRE VÉRIFIÉE AVANT L'ITÉRATION
> ET
> AU DÉBUT DE CHAQUE NOUVEAU CYCLE DE L'ITÉRATION

DESSIN DE LA SITUATION INTERMÉDIAIRE POUR LA FONCTION ❶

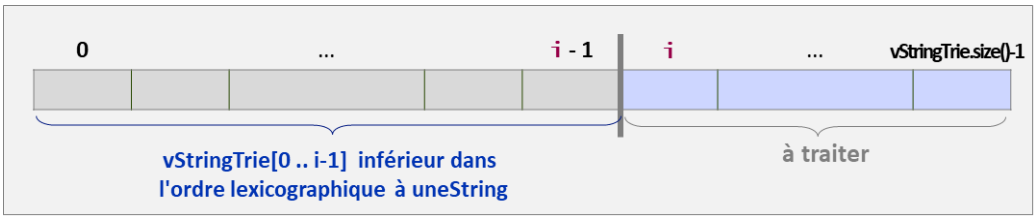
- **unInt** non trouvé dans vInt[0 .. i-1]
- vInt[i] à traiter



INVARIANT : $\text{unInt} \notin \text{vInt}[0\,..\,i-1]$

DESSIN DE LA SITUATION INTERMÉDIAIRE POUR LA FONCTION ❸

- vStringTrie[0 .. i-1] inférieur dans l'ordre lexicographique à uneString
- vStringTrie[i] à traiter



INVARIANT : vStringTrie[0 . . i − 1] inférieur dans l'ordre lexicographique à uneString

B – DÉDUIRE DE L'INVARIANT, L'INITIALISATION DE L'INDICE DE PARCOURS DU VECTEUR ET LA CONDITION DE L'ITÉRATION

FONCTION ❶

- INTIALISATION DE L'INDICE DE PARCOURS

  Aucun élément n'a encore été traité : i doit être initialisé à 0
  initialisation en java :

```java
int i = 0;  // vInt[0 .. -1] est un vecteur vide, unInt ∉ v[0 .. -1]
            // => l'invariant est vérifié
```

- CONDITION D'ITÉRATION
  - i doit être un indice du vecteur et l'invariant doit être vérifié à chaque nouveau cycle d'itération
  - on peut comparer des entiers (type des éléments du vecteur) avec <, <=, >, >=, == ou !=

  condition d'itération en java :

```java
(i < vInt.size() && vInt.get(i) != unInt)
// unInt ∉ v[0 .. -1] => invariant vérifié si on entre dans la boucle
```

FONCTION ❸

- INTIALISATION DE L'INDICE DE PARCOURS

  Aucun élément n'a encore été traité : i doit être initialisé à 0
  initialisation en java :

```java
int i = 0;  // vStringTrie[0 .. -1] est vide
            // => l'invariant est vérifié
```

- CONDITION D'ITÉRATION
  - i doit être un indice du vecteur et l'invariant doit être vérifié à chaque nouveau cycle d'itération
  - une méthode de comparaison de la classe `String` (type des éléments du vecteur) est `compareTo()`

  condition d'itération en java :

```java
(i < vStringTrie.size() && vStringTrie.get(i).compareTo(uneString) < 0)
// vStringTrie[0 .. -1] inférieur dans l'ordre lexicographique à uneString
// => invariant vérifié si on entre dans la boucle
```

C – INSTRUCTIONS DU BLOC DE L'ITÉRATION

Il suffit d'avancer dans le vecteur : `i = i + 1;`

D – DÉDUIRE DE LA NÉGATION DE LA CONDITION D'ITÉRATION, LES DIFFÉRENTS CAS DE SORTIE DE L'ITÉRATION

FONCTION ❶

- *condition d'itération en java*

```
(i < vInt.size() && vInt.get(i) != unInt)
```

- *négation de la condition d'itération*

```
!(i < vInt.size()) || !(vInt.get(i) != unInt)
⇔ i >= vInt.size() || vInt.get(i) == unInt
// mais i ne peut pas être supérieur à vInt.size()
⇔ i == vInt.size() || vInt.get(i) == unInt
```

- *cas de sortie de l'itération*

CAS 1 : `i == vInt.size()`

CAS 2 : `i < vInt.size() && vInt.get(i) == unInt`

FONCTION ❸

- *condition d'itération en java*

```
(i < vStringTrie.size() && vStringTrie.get(i).compareTo(uneString) < 0)
```

- *négation de la condition d'itération*

```
!(i < vStringTrie.size()) || !(vStringTrie.get(i).compareTo(uneString) < 0))
⇔ i >= vStringTrie.size() || vStringTrie.get(i).compareTo(uneString) >= 0
// mais i ne peut pas être supérieur à vInt.size()
⇔ i == vStringTrie.size() || vStringTrie.get(i).compareTo(uneString) >= 0
```

- *cas de sortie de l'itération*

CAS 1 : `i == vStringTrie.size()`

CAS 2 : `i < vStringTrie() && vStringTrie.get(i).compareTo(uneString) > 0`

CAS 3 : `i < vStringTrie() && vStringTrie.get(i).compareTo(uneString) == 0`

E – PRODUIRE LE RÉSULTAT ATTENDU (CF. POSTCONDITION DE LA FONCTION)

FONCTION ❶

- *cas de sortie :* `i == vInt.size() // unInt n'est pas dans le vecteur => retourner faux`
  retour de la fonction : `return false;`
- *cas de sortie :* `i < vInt.size() && vInt.get(i) == unInt`
  `// unInt trouvé à l'indice i => retourner true`
  retour de la fonction : `return true;`
- RETOUR DU RÉSULTAT *:*

```
return (i < vInt.size()); // si on n'avait pas trouvé, i serait égal à v.size()
```

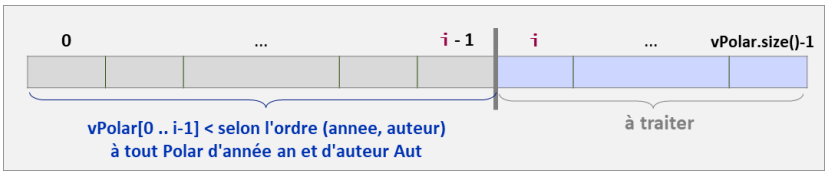
FONCTION ❸

- *cas de sortie :* `i == vStringTrie.size()`
  `// dernier élément de vStringTrie supérieur à uneString dans l'ordre`
  `// lexicographique => retourner -1`
  retour de la fonction : `return -1;`
- *cas de sortie :* `i < vStringTrie.size() && vStringTrie.get(i).compareTo(uneString) > 0`
  `// vStringTrie.geti(i) supérieur à uneString dans l'ordre lexicographique`
  `// => retourner -1`
  retour de la fonction : `return -1;`
- *cas de sortie :* `i < vStringTrie.size() && vStringTrie.get(i).compareTo(uneString) == 0`
  `// uneString trouvée à l'indice i => retourner i`
  retour de la fonction : `return i;`
- RETOUR DU RÉSULTAT *:*

```
if (i < vStringTrie.size() && vStringTrie.get(i).compareTo(uneString) == 0) {
    return i;
} else {
    return -1;
}
```

ÉTAPE 5 COMPLÈTE : EXEMPLE FONCTION ❺ CF. TP6(B)

DESSIN DE LA SITUATION INTERMÉDIAIRE



INVARIANT : $vPolar[0\,..\,i-1]$ inférieur selon l'ordre(annee, auteur)à tout Polar d'année an et d'auteur aut

DÉDUIRE DE L'INVARIANT, L'INITIALISATION DE L'INDICE DE PARCOURS DU VECTEUR ET LA CONDITION DE L'ITÉRATION

- INTIALISATION DE L'INDICE DE PARCOURS
  initialisation en java :
  ```
  int i = 0;  // vPolar[0 .. -1] est vide => l'invariant est vérifié
  ```
- CONDITION D'ITÉRATION
  - la classe **Polar** (type des éléments du vecteur) est dotée d'un ordre naturel, l'ordre (annee, auteur)
  - pour comparer un élément du vecteur **vPolar** à un **Polar** dont l'attribut **annee** serait égal à **an** et l'attribut **auteur** serait égal à **aut**, il est utile de déclarer la variable suivante :
    ```
    Polar unPolar = new Polar(an, aut, ""); // le titre n'est pas un critère de tri
    ```
  condition d'itération en java :
  ```
  (i < vPolar.size() && vPolar.get(i).compareTo(unPolar) < 0)
  // vPolar[0 .. -1] inférieur dans l'ordre (annee, auteur) à unPolar
  ```

NÉGATION DE LA CONDITION D'ITÉRATION, LES DIFFÉRENTS CAS DE SORTIE DE L'ITÉRATION

- *négation de la condition d'itération*
  ```
  i == vPolar.size() || vPolar.get(i).compareTo(unPolar) >= 0
  ```
- *cas de sortie de l'itération*
  cas 1 : `i == vPolar.size()`
  cas 2 : `i < vPolar.size() && vPolar.get(i).compareTo(unPolar) == 0`
  cas 3 : `i < vPolar.size() && vPolar.get(i).compareTo(unPolar) > 0`

PRODUIRE LE RÉSULTAT ATTENDU (CF. POSTCONDITION DE LA FONCTION)

- *sortie sur* cas 1 : `return -1; // le dernier Polar de vPolar est supérieur à unPolar`
- *sortie sur* cas 2 : `return -1; // le dernier Polar examiné dans vPolar est supérieur à unPolar`
- *sortie sur* cas 3 : `return i; // i est l'indice du premier Polar d'année an et d'auteur aut`
- RETOUR DU RÉSULTAT :
  ```
  if (i < vPolar.size() && vPolar.get(i).compareTo(unPolar) == 0) {
      return i;
  } else {
      return -1;
  }
  ```

ALGORITHME

```
public static int rechPremIndSeq(ArrayList<Polar> vPolar, int an, String aut) {
    // {vPolar trié dans l'ordre (annee, auteur)} =>
    //  {*  s'il y a dans vPolar au moins un élément d'année an et d'auteur aut,
    //      résultat = indice du premier de ces éléments
    //   *  sinon, résultat = -1}
    // LA RECHERCHE EST SÉQUENTIELLE !!!
    int i = 0;
    Polar unPolar = new Polar(an, aut, "");
    while (i < vPolar.size() && vPolar.get(i).compareTo(unPolar) < 0) {
        i++;
    }
    if (i < vPolar.size() && vPolar.get(i).compareTo(unPolar) == 0) {
        return i;
    } else {
        return -1;;
    }
}
```