

R1.01

INITIATION AU DÉVELOPPEMENT

Cours 6, partie 2 : Vecteurs triés

✓ recherche associative séquentielle triée

Algorithmes de parcours qui peuvent être partiels

Hervé Blanchon & Anne Lejeune

Université Grenoble Alpes

IUT 2 – Département Informatique


Sommaire

 Recherche associative séquentielle

 Notion de vecteur trié dans l'ordre naturel

 **Recherche associative séquentielle triée**

 dans un **vecteur trié**

 vecteur d'entiers (Integer) et vecteur de chaînes (String)

RECHERCHE ASSOCIATIVE SÉQUENTIELLE

Recherche associative séquentielle

 séquentielle

équivalent à

 le vecteur est parcouru

 avec un indice strictement croissant de une unité à partir de l'indice le plus petit

 parcours de gauche à droite

 avec un indice strictement décroissant de une unité à partir de l'indice le plus grand


 parcours de droite à gauche

NOTION DE VECTEUR TRIÉ DANS L'ORDRE NATUREL


Définition formelle : croissant au sens large

 soit **v** un **ArrayList<MaClasse>** avec **MaClasse** munie d'un ordre naturel

 **v** est dit **trié croissant au sens large** si

 **v** est vide ($v.size() == 0$)

 **v** ne contient qu'un seul élément ($v.size() == 1$)

 **v** contient au moins deux éléments sur l'intervalle d'indices $[0 .. v.size() - 1]$

et

$\forall i \in [1 .. v.size() - 1]$, **v[i-1]** est inférieur ou égal à **v[i]**

 en java avec **MaClasse** de type classe enveloppe d'entiers ou réels

 **v.get(i-1) <= v.get(i)**


 en java avec les autres classes

 **v.get(i-1).compareTo(v.get(i)) <= 0**


Définition formelle : croissant au sens strict

 soit **v** un **ArrayList<MaClasse>** avec **MaClasse** munie d'un ordre naturel

 **v** est dit **trié croissant au sens strict** si

 **v** est vide ($v.size() == 0$)

 **v** ne contient qu'un seul élément ($v.size() == 1$)

 **v** contient au moins deux éléments sur l'intervalle d'indices $[0 .. v.size() - 1]$

et

$\forall i \in [1 .. v.size() - 1]$, **v[i-1]** est strictement inférieur à **v[i]**

 en java avec **MaClasse** de type classe enveloppe d'entiers ou réels

 **v.get(i-1) < v.get(i)**

 en java avec les autres classes

 **v.get(i-1).compareTo(v.get(i)) < 0**

Un entier est-il présent dans un vecteur d'entiers trié dans l'ordre croissant ?

RECHERCHE ASSOCIATIVE DANS UN VECTEUR D'Integer TRIÉ (*DANS L'ORDRE NATUREL*)

Le problème

- Étant donné un vecteur d'entiers **v trié croissant dans l'ordre naturel** représenté sous la forme d'un ArrayList de Integer
- On veut écrire une fonction qui retourne vrai si une valeur `val` est présente dans `v`, faux sinon
- Entête de la fonction à écrire :

```
private static
boolean estEntierPresentTrie(ArrayList<Integer> v, int val)
// spécification {v trié croissant}
//           => { résultat = vrai si val est dans v ;
//           faux sinon}
```

Première analyse du problème



On propose un algorithme itératif



examen successif des éléments de v tant que l'on a pas pu prendre de décision et qu'il reste un élément à traiter



Ce n'est pas obligatoirement un parcours complet



un parcours qui peut être partiel



En effet :



dès que l'on a trouvé, il faut arrêter la recherche puisque la fonction peut retourner vrai



dès que l'on n'a plus d'espoir de trouver, il faut arrêter la recherche puisque la fonction peut retourner faux



l'objectif est toujours de faire le minimum de traitements



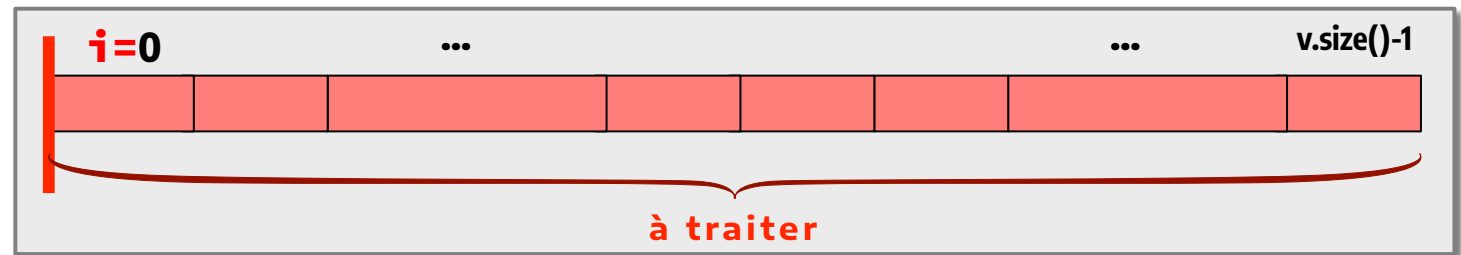
Si on ne trouve pas, on n'aura donc pas forcément examiné tous les éléments du vecteur

Formalisation (parcours qui peut être partiel)

Points intéressants dans le déroulement de l'algorithme ?


 situation initiale

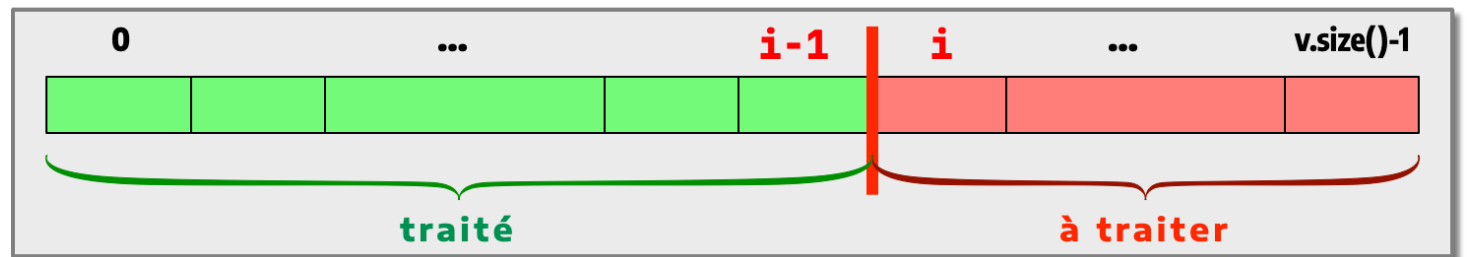
 $i = 0$



 situation intermédiaire de parcours qui peut être partiel

 v traité jusqu'à $i - 1$

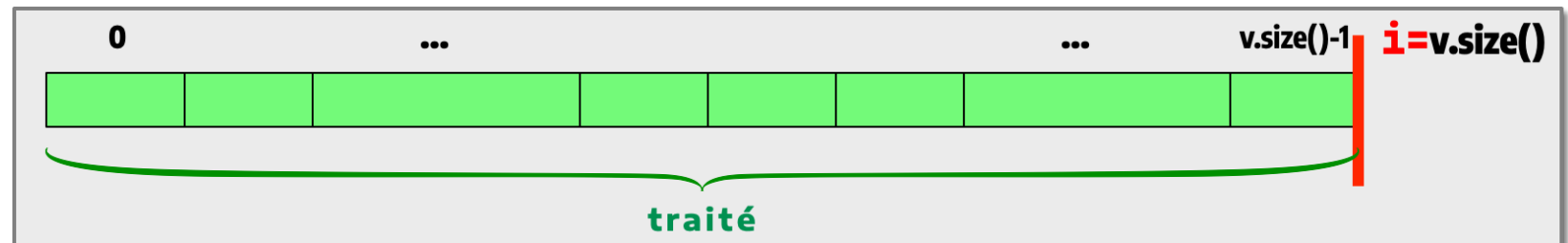
 $v[i]$ permet peut être de prendre une décision (vrai ou faux)



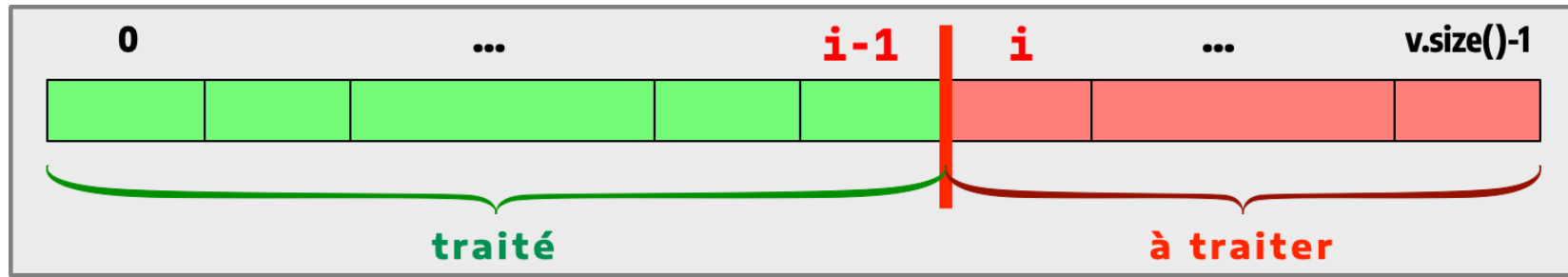
 situation finale du parcours si il est complet

 $i = v.size()$

 v traité



Qu'a-t-on fait sur la zone traitée ?



■ Pour répondre au problème posé, on exprime ce qu'a fait l'algorithme sur la zone traitée :

- l'algorithme a rencontré des valeurs $< \text{val}$ sur l'intervalle $[0 .. i-1]$
- si il avait trouvé `val`, il aurait déjà répondu vrai
- si il avait trouvé une valeur $> \text{val}$, il aurait déjà répondu faux

■ En formalisant :

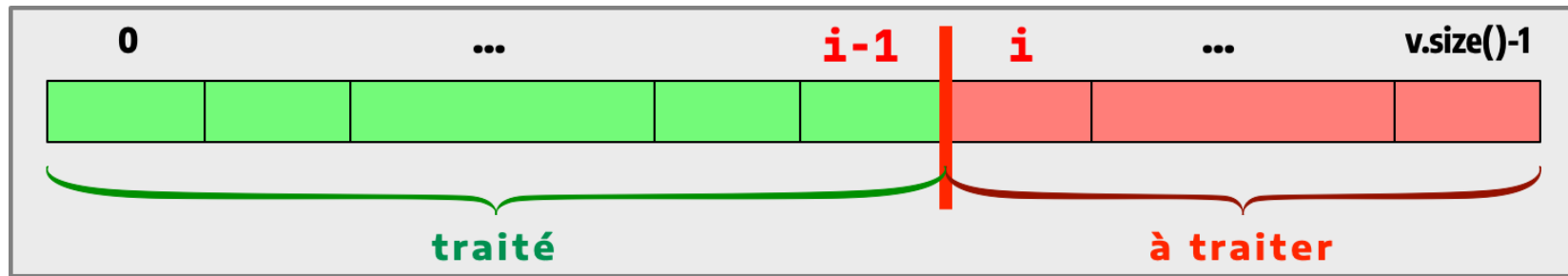
■ $v[0 .. i-1] < \text{val}$

■ On appelle cette formule l'**invariant** de l'algorithme

- il permet de construire 1) l'initialisation, 2) l'itération

Récapitulons

🎨 Situation intermédiaire complètement décrite :



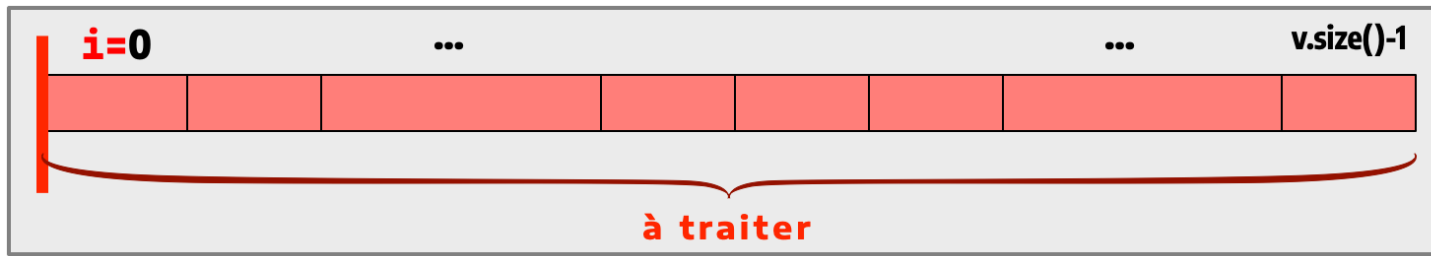
$val > v[0 .. i-1]$

invariant

Invariant et initialisation

🎨 Rappel de l'invariant : $v[0 .. i-1] < \text{val}$

🎨 Rappel du dessin de la situation initiale



🎨 La situation initiale est la situation dans laquelle on se trouve après l'initialisation :

🎨 **initialiser i pour commencer le parcours : $i = 0$;**

🎨 en remplaçant la valeur de i dans l'invariant, on obtient

🎨 $v[0 .. i-1] < \text{val}$ $\{v[0 .. -1] \text{ est un vecteur vide } \text{inf} > \text{sup}\}$

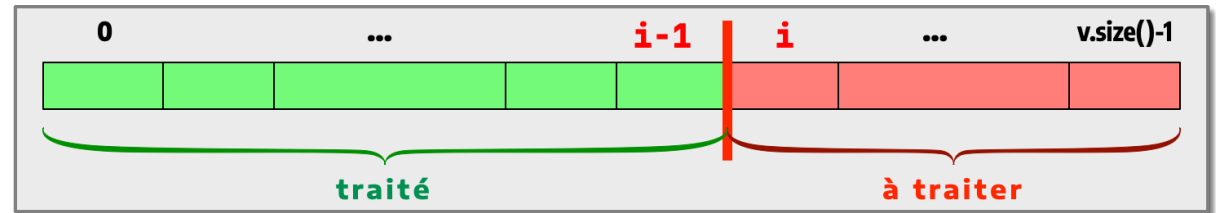
🎨 on peut admettre que cette formule est vraie

🎨 **l'invariant est bien vérifié**

Invariant et itération de parcours qui peut être incomplet

📦 Rappel de l'invariant : $v[0 .. i-1] < val$

📦 Rappel du dessin de la situation intermédiaire



📦 On sait que **i** est initialisé à 0

📦 On sait que $i \leq v.size()-1$ (ou $i < v.size()$)

📦 il reste au moins un $v[i]$ à traiter


📦 **Dans le cas de ce parcours qui peut être incomplet, la suite des traitements dépend de la valeur de $v[i]$ ($v.get(i)$), en effet :**


📦 lorsque $v[i] = val$ ($v.get(i) == val$), il faut s'arrêter (on a trouvé)

📦 lorsque $v[i] > val$ ($v.get(i) > val$), il faut s'arrêter (on ne trouvera pas)


📦 lorsque $v[i] < val$ ($v.get(i) < val$), il faut continuer (avancer)

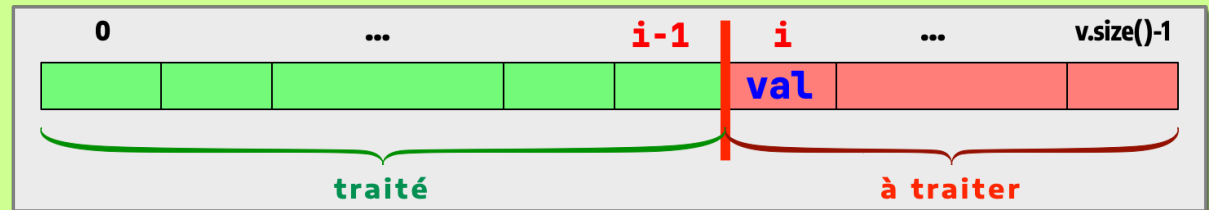
Invariant et itération de parcours qui peut être incomplet


 Cas 1 : $v[i] = \text{val}$ ($v.\text{get}(i) == \text{val}$)

 la condition d'itération doit devenir fausse

 **s'arrêter et répondre vrai**


 c'est une situation finale
si le parcours est partiel

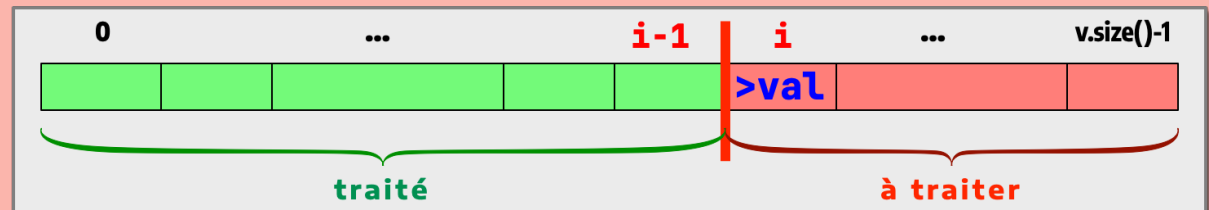



 Cas 2 : $v[i] > \text{val}$ ($v.\text{get}(i) > \text{val}$)

 la condition d'itération doit devenir fausse

 **s'arrêter et répondre faux**

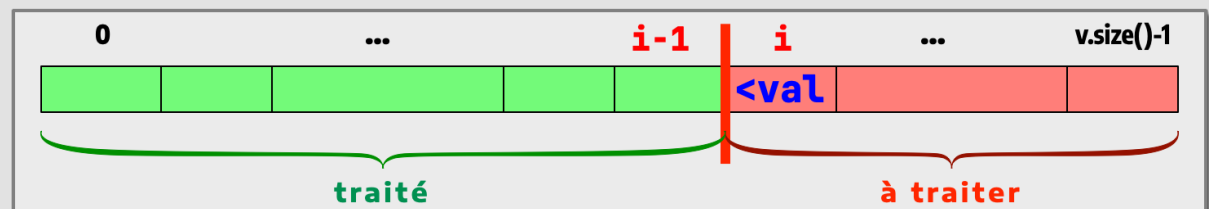
 c'est une situation finale
si le parcours est partiel



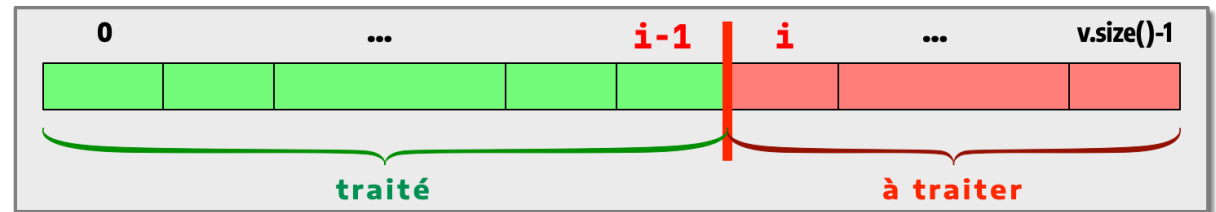
 Cas 3 : $v[i] < \text{val}$ ($v.\text{get}(i) < \text{val}$)

 la condition d'itération doit rester vraie

 **poursuivre la recherche :
incrémenter i**



Invariant et itération de parcours qui peut être incomplet



- On sait que $i \leq v.size() - 1$ (ou $i < v.size()$)
- Dans le cas d'un parcours qui peut être incomplet, la suite des traitements dépend de la valeur de `v[i]` (`v.get(i)`), en effet :
 - lorsque `v[i] == val` (`v.get(i) == val`)
 - trouvé, la condition d'itération doit devenir fausse, situation finale
 - lorsque `v[i] > val` (`v.get(i) > val`)
 - on ne trouvera pas, la condition d'itération doit devenir fausse, situation finale
 - lorsque `v[i] < val` (`v.get(i) < val`)
 - pas encore trouvé, la condition d'itération doit restée vraie (continuer en avançant)
 - `i = i + 1; // invariant vérifié`
- L'itération semble devoir s'écrire :
 - ```
while (i < v.size() & v.get(i) < val) {
 bloc/corps de l'itération
}
```

# Invariant, initialisation et itération

 Pour récapituler on aurait :

Attention ce n'est pas la version définitive

```
// déclaration de i et initialisation
int i = 0;
// v[0 .. -1] < val -> invariant vérifié avant itération
// itération
while (i < v.size() & v.get(i) < val) {
 // parcours qui peut être partiel
 // v[0 .. i] < val -> v[i] traité dans la condition
 i = i + 1; // avancer
 // v[0 .. i-1] < val -> invariant en fin de bloc
}
// négation de la condition d'itération vérifiée
```


 On voit que l'invariant est vérifié avant l'itération et à la fin du bloc d'instructions de l'itération

 **C'est important !**


# Correction de la condition d'itération

---


 Rappel de la proposition :

 `while ( i < v.size() & v.get(i) < val ) {`  
    bloc/corps de l'itération  
`}`

 Proposition corrigée (la seule acceptable) :

 `while ( i < v.size() && v.get(i) < val ) {`  
    bloc/corps de l'itération  
`}`

 Lecture en français :

 tant que **i** est strictement inférieur à **v.size()** et sous réserve que **i** soit strictement inférieur à **v.size()** tant que **v[i]** est strictement inférieur à **val**

# Invariant, initialisation et itération

 Pour récapituler on a :

version définitive

```
// déclaration de i et initialisation
int i = 0;
// v[0 .. -1] < val -> invariant vérifié avant itération
// itération
while (i < v.size() && v.get(i) < val) {
 // parcours qui peut être partiel
 // v[0 .. i] < val -> v[i] traité dans la condition
 i = i + 1; // avancer
 // v[0 .. i-1] < val -> invariant en fin de bloc
}
// négation de la condition d'itération vérifiée
```


 On voit que l'invariant est vérifié avant l'itération et à la fin du bloc d'instructions de l'itération

 **C'est important !**

# Production du résultat

(assertion en sortie d'itération)

## À la sortie de l'itération


 `while (i < v.size() && v.get(i) < val) {`  
    bloc/corps de l'itération  
`}`

## La négation de la condition d'itération est vraie (assertion) :

 `!(i < v.size() && v.get(i) < val)`

 `!(i < v.size()) || !(v.get(i) < val)`

 `i >= v.size() || v.get(i) >= val`

 or i croît depuis 0, la première fois que `i >= v.size()` est vrai c'est lorsque `i == v.size()`

 `i == v.size() || v.get(i) >= val`

## Se lit en français

 `i == v.size()`

ou sinon

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

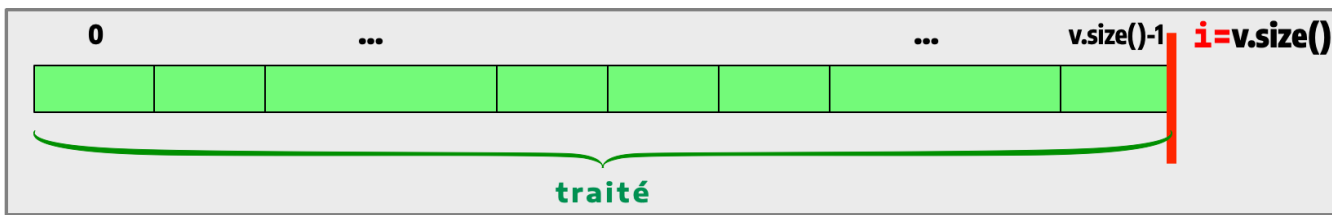
ce sont les trois situations  
finales que l'on a repéré

# Production du résultat

(situations finales et  
assertion en sortie d'itération)

## Situations finales

 situation finale du parcours complet

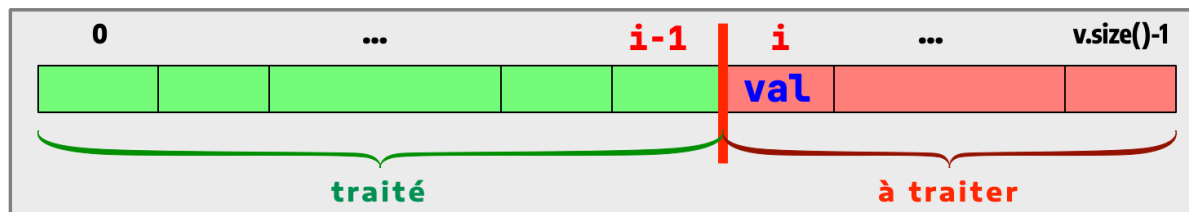



## Assertion à la sortie de l'itération

 `i == v.size()`

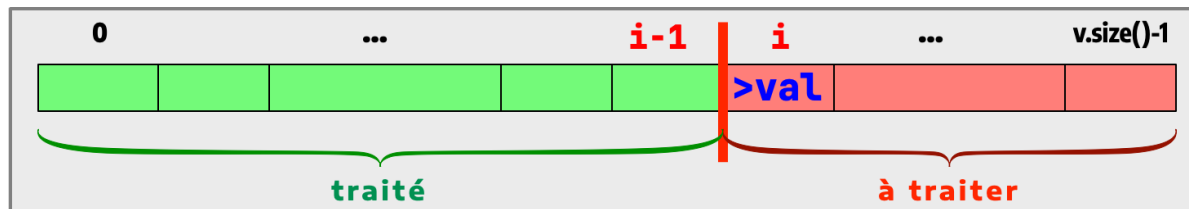
ou sinon

 situations finales du parcours partiel



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

`v.get(i) == val`



`v.get(i) > val`

# Production du résultat

🏠 État du code produit :

```
// déclaration de i et initialisation
int i = 0;
// val ∉ v[0 .. -1] -> invariant vérifié avant itération
// itération
while (i < v.size() && v.get(i) < val) {
 // parcours qui peut être partiel
 // val ∉ v[0 .. i] -> v[i] traité dans la condition
 i = i + 1; // avancer
 // val ∉ v[0 .. i-1] -> invariant en fin de bloc
}
// négation de la condition d'itération vérifiée
// i == v.size() || v.get(i) == val

// instruction(s) pour produire le résultat
```

# Production du résultat

(préparation du tableau de sortie)

Pour écrire la ou les instructions pour rendre le résultat, une méthode infaillible consiste à examiner toutes les situations en faisant un tableau de sortie

 condition de sortie



`i == v.size()` || `v.get(i) >= val`

 préparation du tableau (3 lignes cf. table du ||)

| <code>i == v.size()</code> | <code>v.get(i) &gt;= val</code> | résultat |
|----------------------------|---------------------------------|----------|
|                            |                                 |          |
|                            |                                 |          |
|                            |                                 |          |



# Production du résultat

(remplissage du tableau de sortie)

## Rappel

-  si `val` est dans `v` rendre vrai (`true`)
-  si `val` n'est pas dans `v` rendre faux (`false`)

## Tableau de sortie (3 cas, 1., 2., 3.)

| <code>i == v.size()</code>            | <code>v.get(i) &gt;= val</code>       | résultat                                                             |
|---------------------------------------|---------------------------------------|----------------------------------------------------------------------|
| vrai                                  | <b>non examinée</b>                   | <code>false</code>                                                   |
| faux ( <code>i &lt; v.size()</code> ) | vrai                                  | <code>&gt;&gt; = : true</code><br><code>&gt;&gt; &gt; : false</code> |
| faux ( <code>i &lt; v.size()</code> ) | faux ( <code>v.get(i) == val</code> ) | <b>impossible</b> ( <code>while</code> )                             |

1.

2.

3.

- Sortie du vecteur et toutes inférieures → {on n'a pas trouvé} résultat = **false**
- 2.1 Dans le vecteur et = → {on a trouvé} résultat = **true**
- 2.2 Dans le vecteur et > → {on ne peut pas trouver} résultat = **false**
3. Dans le vecteur et pas encore trouvé → {l'itération n'est pas terminée} **impossible**

# Production du résultat

(lecture du tableau de sortie)

🧩 Quelle instruction pour retourner le résultat attendu ?

| <code>i == v.size()</code>            | <code>v.get(i) &gt;= val</code>       | résultat                  |
|---------------------------------------|---------------------------------------|---------------------------|
| vrai                                  | <del>non examinée</del>               | false                     |
| faux ( <code>i &lt; v.size()</code> ) | vrai                                  | <code>= : true</code>     |
| faux ( <code>i &lt; v.size()</code> ) | faux                                  | <code>&gt; : false</code> |
| faux ( <code>i &lt; v.size()</code> ) | faux ( <code>v.get(i) == val</code> ) | impossible (while)        |

🧩 Le résultat est

🧩 vrai si `i < v.size() && v.get(i) == val`

🧩 faux sinon

`return i < v.size() && v.get(i) == val;`

# La fonction estEntierPresentTrie

```
private static boolean estEntierPresentTrie(ArrayList<Integer> v,
 int val) {
 // déclaration de i et initialisation
 int i = 0;
 // v[0 .. -1] < val -> invariant vérifié avant itération

 while (i < v.size() && v.get(i) < val) {
 // parcours qui peut être partiel
 // v[0 .. i] < val -> v[i] traité dans la condition
 i = i + 1; // avancer
 // v[0 .. i-1] < val -> invariant en fin de bloc
 }
 // négation de la condition d'itération vérifiée
 // i == v.size() || v.get(i) >= val

 // instruction pour produire le résultat
 return i < v.size() && v.get(i) == val;
}
```

# Classe d'utilisation

## Classe

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

public class RechSequArrayList {
 private static int estEntierPresentTrie(ArrayList<Integer> v, int val) {
 // spécification {v trié croissant} => {résultat = vrai si val est dans v ;
 // faux sinon}
 -> voir planche précédente
 }

 public static void main(String[] args) {
 // déclaration et initialisation à partir d'une liste d'entiers
 ArrayList<Integer> vectTrieInteger = new ArrayList<>(Arrays.asList(10, 12, 28, 45, 85));
 System.out.println("Entiers : " + vectTrieInteger);
 System.out.println("9 ? : " + estEntierPresentTrie (vectTrieInteger, 9));
 System.out.println("10 ? : " + estEntierPresentTrie (vectTrieInteger, 10));
 System.out.println("27 ? : " + estEntierPresentTrie (vectTrieInteger, 27));
 System.out.println("85 ? : " + estEntierPresentTrie (vectTrieInteger, 85));
 System.out.println("90 ? : " + estEntierPresentTrie (vectTrieInteger, 90));
 }
}
```

## Trace

```
Entiers : [10, 12, 28, 45, 85]
9 ? : false
10 ? : true
27 ? : false
85 ? : true
90 ? : false
```

Une chaîne est-elle présente dans un vecteur de chaînes trié dans l'ordre lexicographique ?

## **RECHERCHE ASSOCIATIVE DANS UN VECTEUR DE String TRIÉ LEXICOGRAPHIQUEMENT (*DANS L'ORDRE NATUREL*)**

# Le problème










---

- 🧩 Étant donné un vecteur **v trié** de chaînes représenté sous la forme d'un ArrayList de String
- 🧩 On veut écrire une fonction qui retourne vrai si une chaîne `val` est présente dans `v`, faux sinon
  - 🧩 Cette recherche est sensible à la casse
- 🧩 Entête de la fonction à écrire :

```
private static
boolean estChainePresenteTrie(ArrayList<String> v,
 String val)
// spécification {v trié croissant lexicographiquement}
// => { résultat = vrai si val est dans v ;
// faux sinon}
```

# Le problème est-il nouveau ?

---

-  Nous avons déjà résolu le même problème avec un `ArrayList` de `Integer`.
-  On peut donc mettre en œuvre les mêmes étapes et obtenir le même squelette de fonction
  -  Seule la condition d'itération va être différente
-  Condition d'itération de la recherche triée d'un `int val`
  -  `i < v.size() && v.get(i) < val`
-  Condition d'itération de la recherche d'un `String val`
  -  Comment exprimer qu'une chaîne (`string1`) est strictement inférieure à une autre chaîne (`string2`) lexicographiquement
    -  `string1.compareTo(string2) < 0`
  -  `i < v.size() && v.get(i).compareTo(val) < 0`

# La fonction estChainePresenteTrie

---

```
private static boolean estChainePresenteNonTrie(ArrayList<String> v,
 String val) {
 // déclaration de i et initialisation
 int i = 0;
 // val ∉ v[0 .. -1] -> invariant vérifié avant itération

 while (i < v.size() && v.get(i).compareTo(val) < 0) {
 // parcours qui peut être partiel
 // val ∉ v[0 .. i] -> v[i] traité dans la condition
 i = i + 1; // avancer
 // val ∉ v[0 .. i-1] -> invariant en fin de bloc
 }
 // négation de la condition d'itération vérifiée
 // i == v.size() || v.get(i).compareTo(val) >= 0

 // instruction pour produire le résultat
 return i < v.size() && v.get(i).compareTo(val) == 0;
}
```