

R1.01

INITIATION AU DÉVELOPPEMENT

Cours 5, partie 3 : Vecteurs non triés

✓ recherche associative séquentielle

Algorithmes de parcours qui peuvent être partiels

Hervé Blanchon & Anne Lejeune

Université Grenoble Alpes


IUT 2 – Département Informatique

Sommaire

 L'accès : par position et **associatif**

 La **recherche associative séquentielle non triée**

 dans un **vecteur quelconque** (non trié éventuellement)

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

 La **condition d'itération composée**

 **&&** et **|**

 Notion de vecteur trié dans l'ordre naturel

 Le **coût** de la recherche associative séquentielle


ACCÈS PAR POSITION & ASSOCIATIF

Accès à un élément de vecteur

 Accès par position dans un `ArrayList<E>` v (*déjà vu*)

 **`v.get(i)`**


 **Accès associatif** (recherche associative)

 soit `val` une variable de même type que les éléments contenus dans `v[0..v.size()-1]`

 Déterminer

 si il existe un indice $i \in [0..v.size()-1]$
tel que `v[i] = val` : fonction booléenne (prédicat)

 résultat = oui si `val` présente, faux sinon

 la valeur de l'indice $i \in [0..v.size()-1]$
tel que `v[i] = val` : fonction entière

 résultat = i si `val` présente, `v.size()` ou **-1** sinon

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

Un entier est-il présent dans un vecteur d'entiers ?

RECHERCHE ASSOCIATIVE DANS UN VECTEUR D'Integer QUELCONQUE









Le problème

- Étant donné un vecteur **v quelconque** d'entiers 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 estEntierPresentNonTrie(ArrayList<Integer> v,
                                int val)

// {v quelconque} =>
// { résultat = vrai si val est dans v ; faux sinon}
```


Première analyse du problème

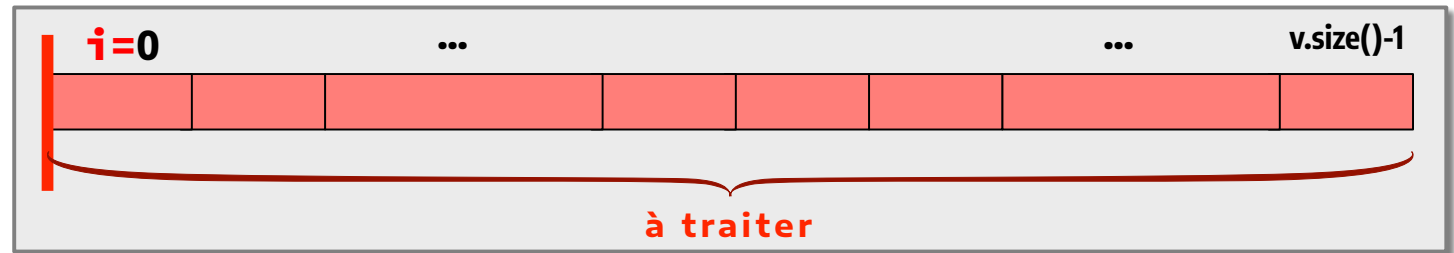
-  On propose un algorithme itératif
 -  examen successif des éléments de **v** tant que l'on a pas pu prendre de décision et qu'il reste un élément à traiter
-  **Ce n'est pas obligatoirement un parcours complet**
 -  un parcours qui peut être partiel
-  En effet, dès que l'on a trouvé, il faut arrêter la recherche puisque la fonction peut retourner vrai
 -  *l'objectif est toujours de faire le minimum de traitements*
 -  Si on trouve, c'est forcément dans l'intervalle $[0 .. v.size()-1]$
 -  Si on ne trouve pas, on aura examiner tous les éléments du vecteurs

Formalisation (parcours qui peut être partiel)

Points intéressants dans le déroulement de l'algorithme ?

 situation initiale

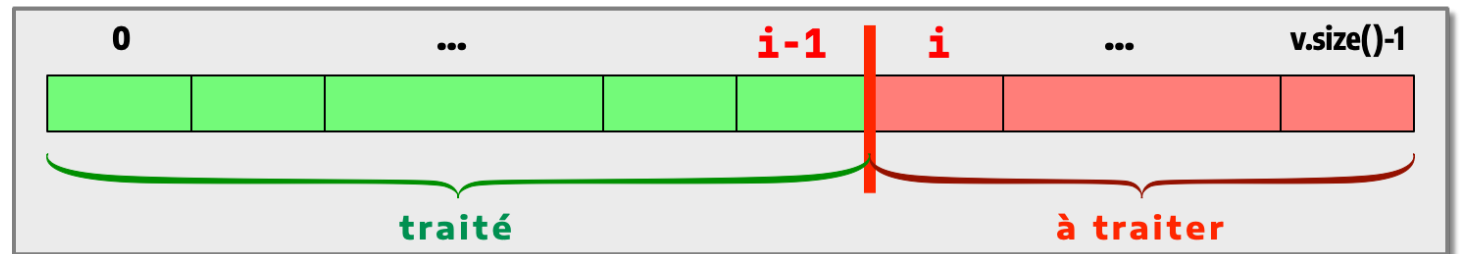
 $i = 0$



 situation intermédiaire de parcours qui peut être partiel

 v traité jusqu'à $i - 1$

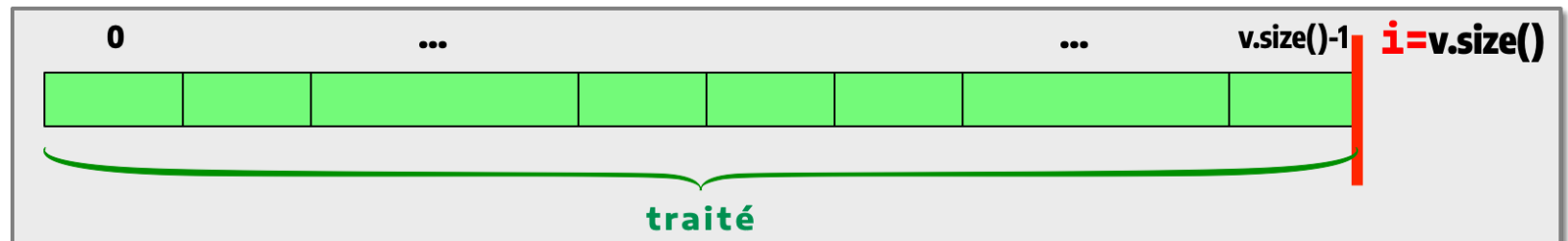
 $v[i]$ permet peut être de répondre vrai



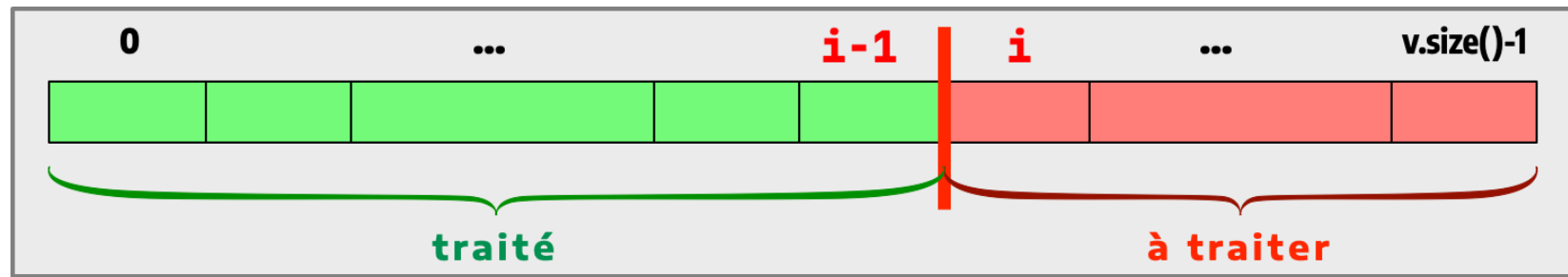
 situation finale du parcours si il est complet

 $i = v.size()$

 v traité



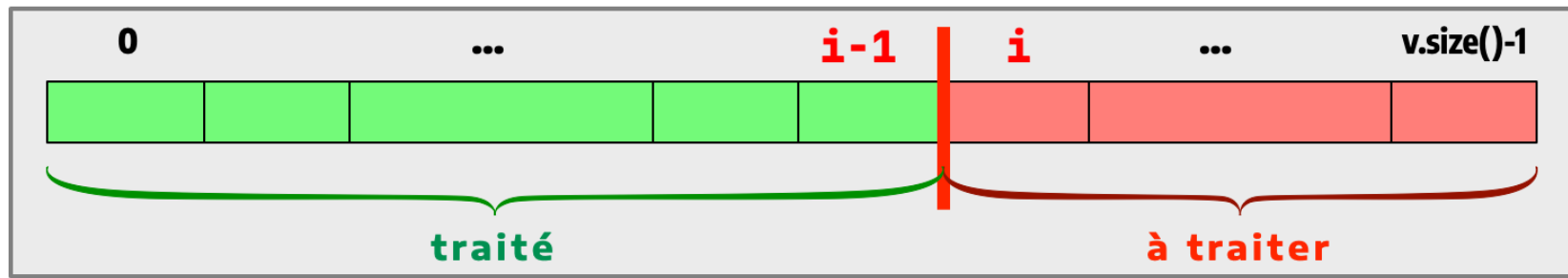
Qu'a-t-on fait sur la zone traitée ?



- Pour répondre au problème posé, on exprime ce qu'a fait l'algorithme sur la zone traitée :
 - l'algorithme n'a pas encore rencontré `val` sur l'intervalle `[0 .. i-1]`
 - si il avait rencontré `val`, il aurait déjà répondu vrai
- En formalisant :
 - $val \notin v[0 .. i-1]$
- On appelle cette formule l'**invariant** de l'algorithme
 - il permet de construire 1) l'initialisation, 2) l'itération

Récapitulons

🖼 Situation intermédiaire complètement décrite :



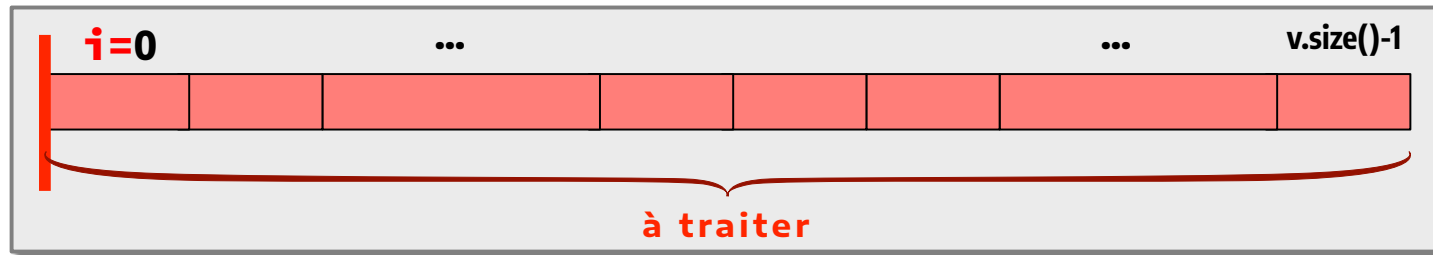
$val \notin v[0 .. i-1]$

invariant

Invariant et initialisation

■ Rappel de l'invariant : $val \notin v[0 .. i-1]$

■ 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

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

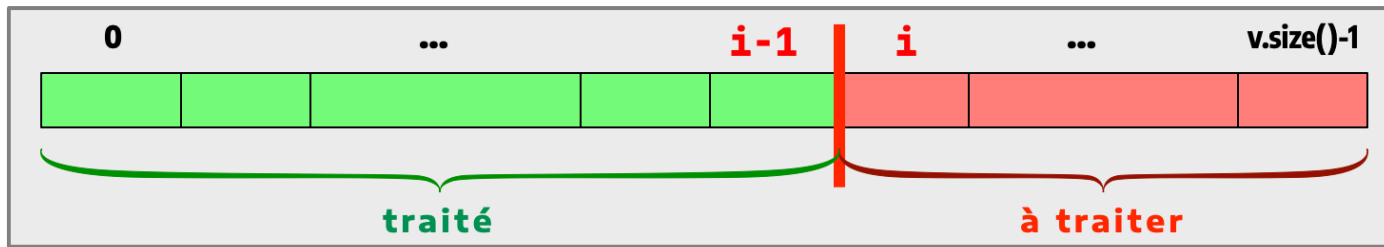
■ on est certain que val n'est pas présente dans un vecteur vide

■ **l'invariant est bien vérifié**

Invariant et itération de parcours qui peut être incomplet

■ Rappel de l'invariant : $val \notin v[0 .. i-1]$

■ 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

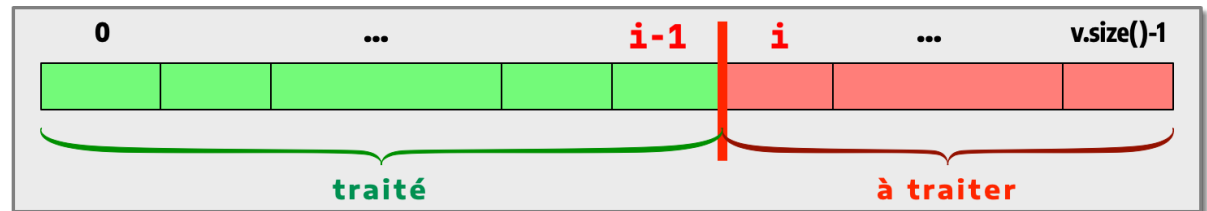
■ **Dans le cas d'un parcours qui peut être incomplet, la suite des traitements dépend de la valeur de $v[i]$ ($v.get(i)$), en effet :**

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


■ lorsque $v[i] \neq val$ ($v.get(i) != val$), il faut continuer (avancer)

Invariant et itération de parcours qui peut être incomplet


Situation intermédiaire

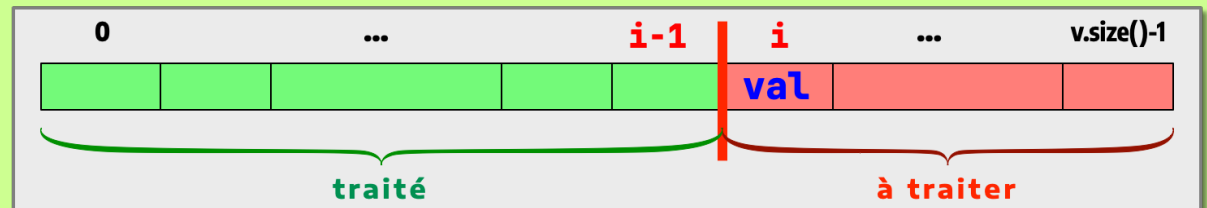


Cas 1 : $v[i] = \text{val}$ ($v.get(i) == \text{val}$)

 la condition d'itération doit devenir fausse

 **s'arrêter et répondre vrai**

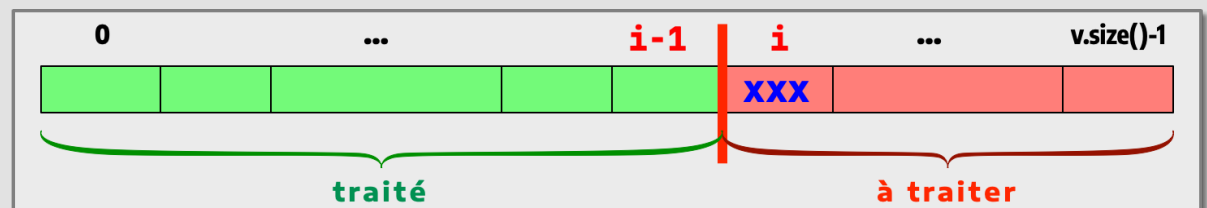
 c'est la situation finale
si le parcours est partiel



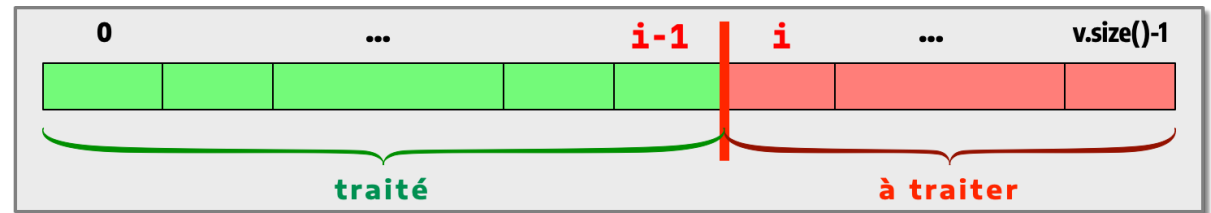
Cas 2 : $v[i] \neq \text{val}$ ($v.get(i) != \text{val}$)

 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 ≤ 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`)
 - 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;
// val ∉ v[0 .. -1] -> invariant vérifié avant 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
```

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

 **C'est important !**

# CONDITION COMPOSÉE

## &, | ET &&, ||

# Retour sur & et |

 Dans l'évaluation des expressions  $a \ \& \ b$  ou  $a \ | \ b$

  $a$  et  $b$  sont d'abord toutes les deux évaluées

 puis le résultat de  $a \ \& \ b$  ou  $a \ | \ b$  est calculé en utilisant les tables déjà vues

| a & b |       | et       |
|-------|-------|----------|
| a     | b     | Résultat |
| true  | true  | true     |
| true  | false | false    |
| false | true  | false    |
| false | false | false    |

| a   b |       | ou (inclusif) |
|-------|-------|---------------|
| a     | b     | Résultat      |
| true  | true  | true          |
| true  | false | true          |
| false | true  | true          |
| false | false | false         |

# Utilisation de l'instruction d'itération proposée

 Soit  $v = [45, 12, 28, 85, 18]$  et  $val = 42$


 Imaginons que nous ayons écrit la fonction avec la condition proposée :  
`while (  $i < v.size()$  &  $v.get(i) \neq val$  )`

 *nous n'avons vu que l'initialisation et l'itération pour l'instant*

 À l'exécution nous aurions le message d'erreur suivant :

Exception in thread "main" java.lang.IndexOutOfBoundsException:  
Index 5 out of bounds for length 5


 Décodage du message :

 ce message signale que l'on veut consulter  $v$  à l'indice 5 qui n'est pas autorisé étant donné que le  $v$  est de longueur 5, l'intervalle des indices autorisés est  $[0 .. 4]$

 Explication :

  $i$  est initialisé à 0 et tant que l'on ne trouve 42 pas  $i$  est incrémenté








 42 n'étant pas dans  $v$ ,  $i$  va prendre les valeurs 0, 1, 2, 3, 4, 5

 pour les valeurs dans l'intervalle  $[0 .. 4]$  il n'y a pas de problème à l'exécution de  $v.get(i)$

 par contre  $v.get(i)$  avec  $i = 5$  est interdit !

# Utilisation de l'instruction d'itération proposée

---

-  Soit  $v = [45, 12, 28, 85, 18]$  et  $val = 42$
-  Imaginons que nous ayons écrit la fonction avec la condition proposée : `while (  $i < v.size()$  &  $v.get(i) \neq val$  )`
-  Contrainte :
  -   `$v.get(i)$`  est autorisé **si et seulement si**  $i < v.size()$
-  Correction possible :
  -  interdire l'évaluation de  `$v.get(i) \neq val$`
  -  lorsque la condition  $i < v.size()$  devient fausse


# Voir autrement & et |

## Observation


| A    | B    | A & B |
|------|------|-------|
| vrai | vrai | vrai  |
| vrai | faux | faux  |
| faux | vrai | faux  |
| faux | faux | faux  |


| A    | B    | A   B |
|------|------|-------|
| vrai | vrai | vrai  |
| vrai | faux | vrai  |
| faux | vrai | vrai  |
| faux | faux | faux  |



 si A = vrai alors


 A & B = B


 si A = faux alors

 A & B = A (faux)


 Inutile d'examiner B



 si A = vrai alors

 A | B = A (vrai)

 Inutile d'examiner B

 si A = faux alors

 A or B = B


# Éviter les évaluations inutiles

---

 Deux nouveaux opérateurs booléens ...

 **&&** (se lit en français « **et alors** »)

 **||** (se lit en français « **ou sinon** »)

 ... qui réalisent ce que l'on appelle l'évaluation  
« court circuit » (“short-circuit evaluation” en  
anglais)


 en faire le moins possible !

# Table de vérité de && — « et alors »

 La table

| A    | B            | A && B |
|------|--------------|--------|
| vrai | vrai         | vrai   |
| vrai | faux         | faux   |
| faux | non examinée | faux   |

 Remarque

 on n'a que 3 lignes (3 situations) car lorsque l'expression **A** est fausse, l'expression **B** n'est pas examinée




# Table de vérité de $|$ — « ou sinon »

 La table

| A    | B            | A $ $ B |
|------|--------------|---------|
| vrai | non examinée | vrai    |
| faux | vrai         | vrai    |
| faux | faux         | faux    |

 Remarque

 on n'a que 3 lignes (3 situations) car lorsque l'expression **A** est vraie, l'expression **B** n'est pas examinée


# Négation d'expression booléenne


---

 Loi de Morgan étendue

  $!(A \ \& \ B) = !A \ | \ !B$

  $!(A \ | \ B) = !A \ \& \ !B$

  $!(A \ \&\& \ B) = !A \ || \ !B$


  $!(A \ || \ B) = !A \ \&\& \ !B$

Un entier est-il présent dans un vecteur d'entier ?


## **RECHERCHE ASSOCIATIVE** **DANS UN VECTEUR NON TRIÉ** *(suite et fin)*

# 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 différent de **val**

# Invariant, initialisation et itération

 Pour récapituler on aurait :

version définitive

```
// déclaration de i et initialisation
int i = 0;
// val ∉ v[0 .. -1] -> invariant vérifié avant 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
```

 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  $i \geq v.size()$  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
```

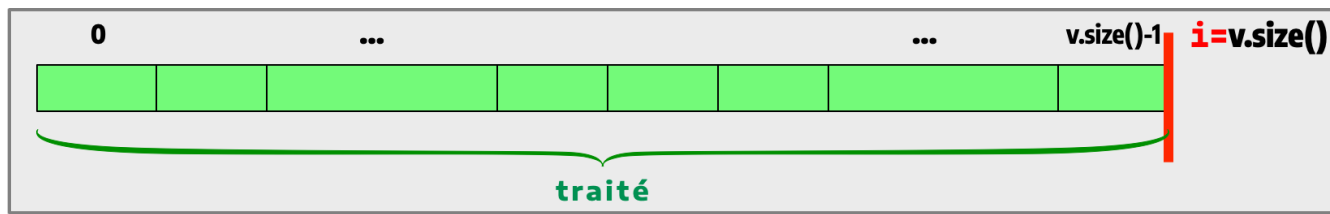
ce sont les deux situations  
finales que l'on a repéré

# Production du résultat

(situations finales et  
assertion en sortie d'itération)

## Situations finales

 situation finale du parcours  
complet

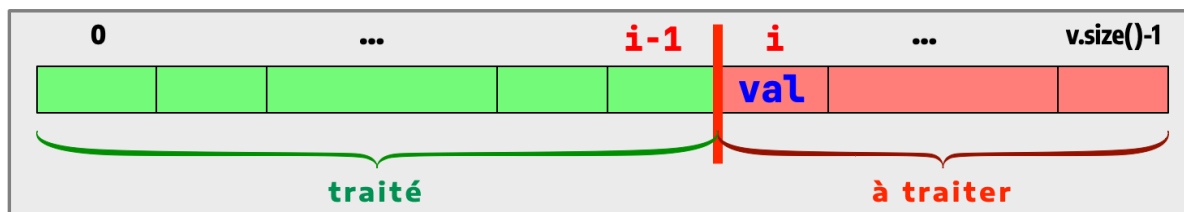



## Assertion à la sortie de l'itération

 `i == v.size()`

ou sinon

 situation finale du parcours  
partiel



 `i < v.size()` et  
`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
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 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) == 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) == val</code>          | résultat                                 |    |
|---------------------------------------|---------------------------------------|------------------------------------------|----|
| vrai                                  | <b>non examinée</b>                   | <code>false</code>                       | 1. |
| faux ( <code>i &lt; v.size()</code> ) | vrai                                  | <code>true</code>                        | 2. |
| faux ( <code>i &lt; v.size()</code> ) | faux ( <code>v.get(i) != val</code> ) | <b>impossible</b> ( <code>while</code> ) | 3. |

- Sortie du vecteur et pas encore trouvé → {on n'a pas trouvé} résultat = `false`
- Dans le vecteur et trouvé → {on a trouvé} résultat = `true`
- 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) == val</code>          | résultat           |
|---------------------------------------|---------------------------------------|--------------------|
| vrai                                  | non examinée                          | false              |
| faux ( <code>i &lt; v.size()</code> ) | vrai                                  | true               |
| faux ( <code>i &lt; v.size()</code> ) | faux ( <code>v.get(i) == val</code> ) | impossible (while) |

🧩 La colonne `v.get(i) == val` n'est pas utile pour produire le résultat

🧩 Le résultat est

|                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------|
|  vrai si <code>i &lt; v.size()</code> |
|  faux sinon                           |

————— `return i < v.size();`

# La fonction estEntierPresentNonTrie

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

// déclaration de i et initialisation
int i = 0;
// val ∉ v[0 .. -1] -> invariant vérifié avant 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 pour produire le résultat
return i < v.size();
}
```

# Classe d'utilisation

Classe

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

public class RechSequArrayList {

 private static
 boolean estEntierPresentNonTrie(ArrayList<Integer> v, int val) {
 // {v quelconque} => { résultat = vrai si val est dans v ; faux sinon}

 -> cf. code planche précédente
 }

 public static void main(String[] args) {
 ArrayList<Integer> vectInteger = new ArrayList<>(Arrays.asList(45, ..., 10));
 System.out.println("Entiers : " + vectInteger);
 System.out.println("45 ? : " + estEntierPresentNonTrie(vectInteger, 45));
 System.out.println("10 ? : " + estEntierPresentNonTrie(vectInteger, 10));
 System.out.println("42 ? : " + estEntierPresentNonTrie(vectInteger, 42));
 }
}
```

Trace

```
Entiers : [45, 12, 28, 85, 10]
45 ? : true
10 ? : true
42 ? : false
```

Une chaîne est-elle présente dans un vecteur de chaînes ?

## **RECHERCHE ASSOCIATIVE DANS UN VECTEUR DE `String` NON TRIÉ**

# Le problème

---

- Étant donné un vecteur **v quelconque** 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 estChainePresentsNonTrie(ArrayList<String> v,
 String val)

// {v quelconque} =>
// { 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 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 différente d'une autre de chaîne (`string2`) lexicographiquement
    - `string1.compareTo(string2) != 0`
  - `i < v.size() && v.get(i).compareTo(val) != 0`



# La fonction estChainePresenteNonTrie

```
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();
}
```