R1.01

Initiation au développement

Cours 5, partie 2 : vecteurs

✓ algorithmes de parcours complet

Hervé Blanchon & Anne Lejeune
Université Grenoble Alpes
IUT 2 – Département Informatique

Sommaire

- Somme des éléments d’un vecteur d’entiers
  - `int getSommeVectInt(ArrayList<Integer> v)`

- Nombre d’éléments supérieurs à une valeur donnée dans un vecteur d’entiers
  - `int getNbSupVal(ArrayList<Integer> v, int val)`

Premier algorithme de parcours complet

SOMME DES ÉLÉMENTS D’UN VECTEUR D’ENTIERS (`Integer`)

Le problème

- Étant donné un vecteur **v** d'entiers représenté sous la forme d'un `ArrayList` de `Integer`
- On veut écrire une fonction qui retourne la somme des entiers contenus dans **v**
- Entête de la fonction à écrire :

```
private static int getSommeVectInt(ArrayList<Integer> v)
// { v quelconque } =>
// { résultat = somme des éléments de v ;
//                 0 si v est vide }
```

Première analyse du problème

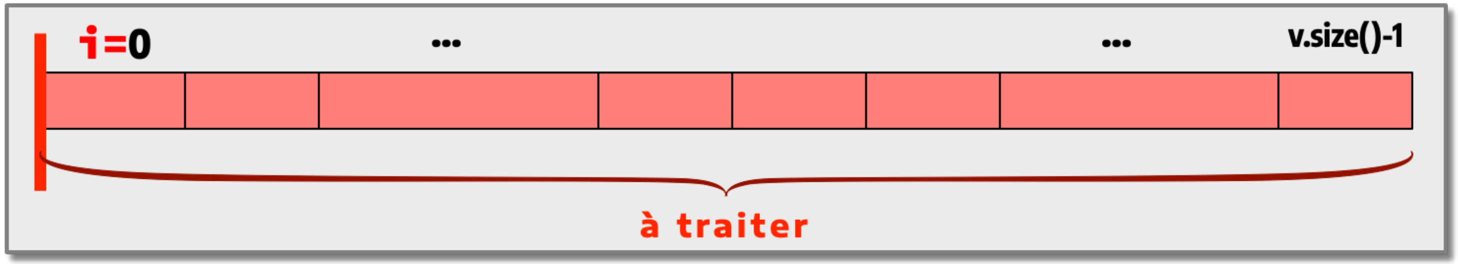
- Pour résoudre ce problème, on propose 3 étapes :
  1) initialiser les variables de l'algorithme
  2) faire un **parcours complet** (traiter tous les éléments) de **v** en partant du premier indice (`0`) jusqu'au dernier indice (`v.size()-1`) en accumulant les valeurs rencontrées dans un accumulateur
     - c'est donc une itération qui se fera avec un indice `i` qui va parcourir l'intervalle `[0 .. v.size()-1]`
     - à chaque pas de l'itération on va :
       - traiter v[`i`] (accumuler)
       - puis avancer (incrémenter `i`)
  3) retourner, comme résultat, la valeur accumulée

*on propose donc un **algorithme itératif de parcours complet***

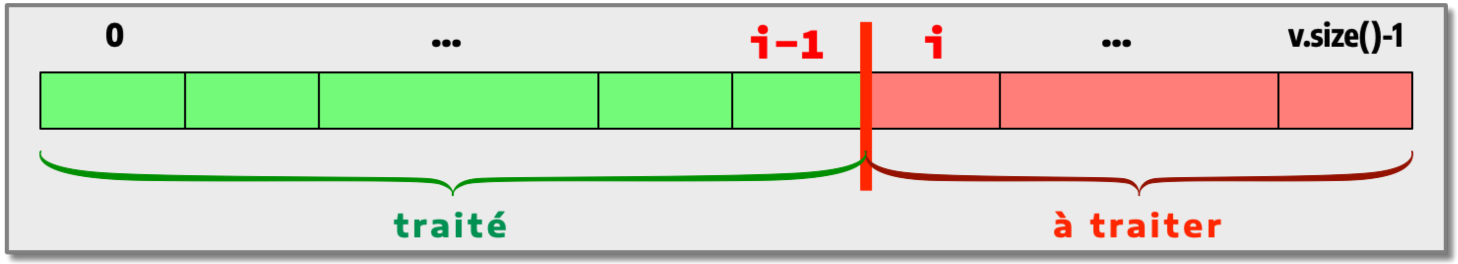
Effort de formalisation (parcours complet)

- Points intéressants dans le déroulement de l'algorithme ?

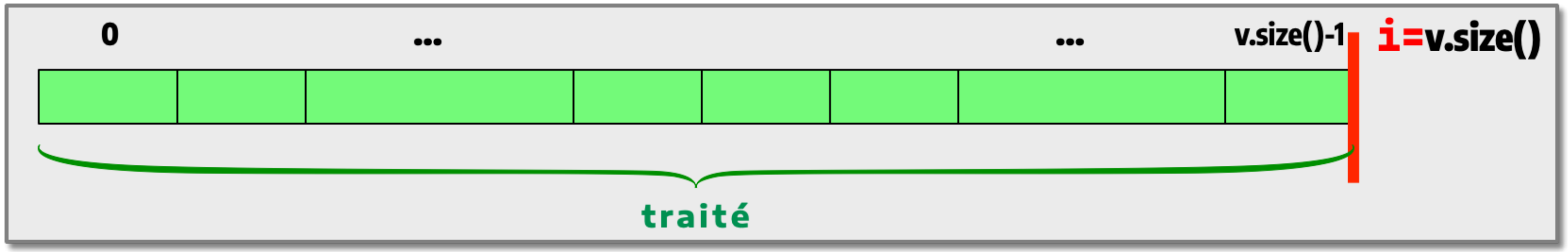
- **situation initiale**
  - `i = 0`



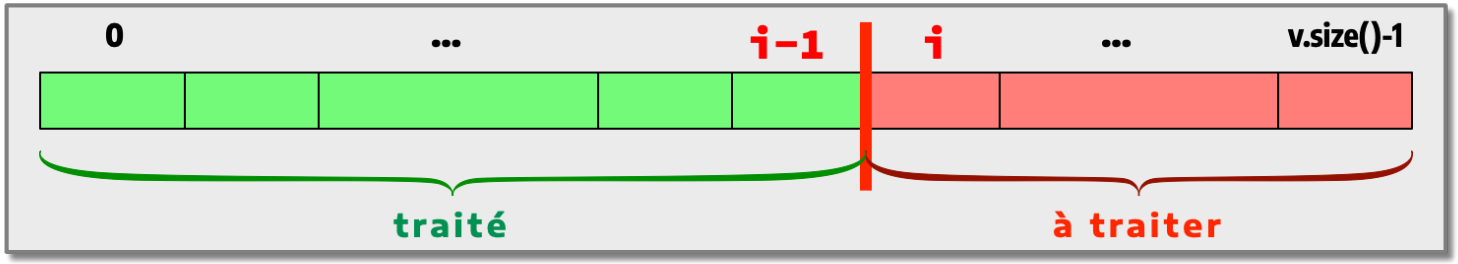
- **situation intermédiaire de parcours complet**
  - **v traité jusqu'à `i - 1`**
  - **v[`i`] à traiter**



- **situation finale de parcours complet**
  - **`i` = v.size()**
  - **v entièrement traité**



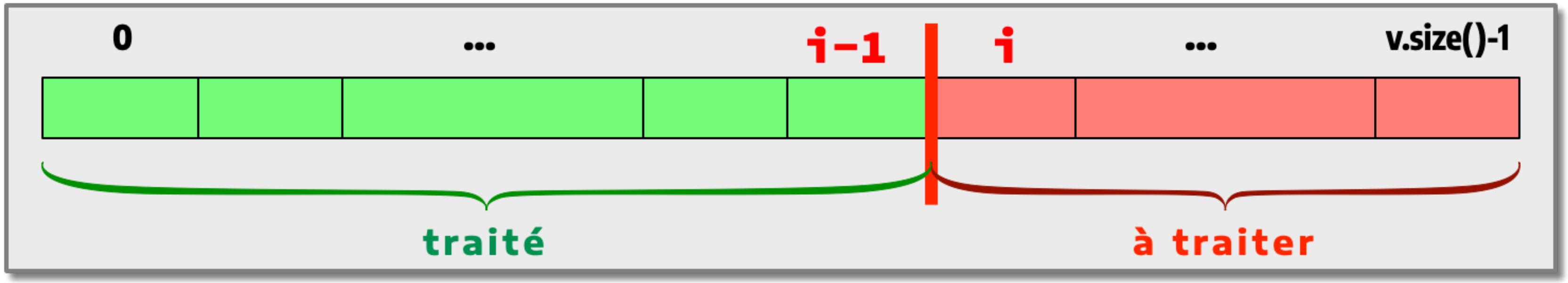
Qu’a-t-on fait sur la zone traitée ?



- Pour répondre au problème posé, on exprime ce qu’a fait l’algorithme sur la zone traitée :
  - l’algorithme a fait la somme des éléments sur l’intervalle [`0` .. `i-1`]
- En formalisant :
  - $s = \sum v[0 \,..\, i-1]$
- On appelle cette formule l’invariant de l’algorithme
  - il permet de construire 1) l’initialisation, 2) l’itération et 3) la production du résultat

Récapitulons

- Situation intermédiaire complètement décrite :

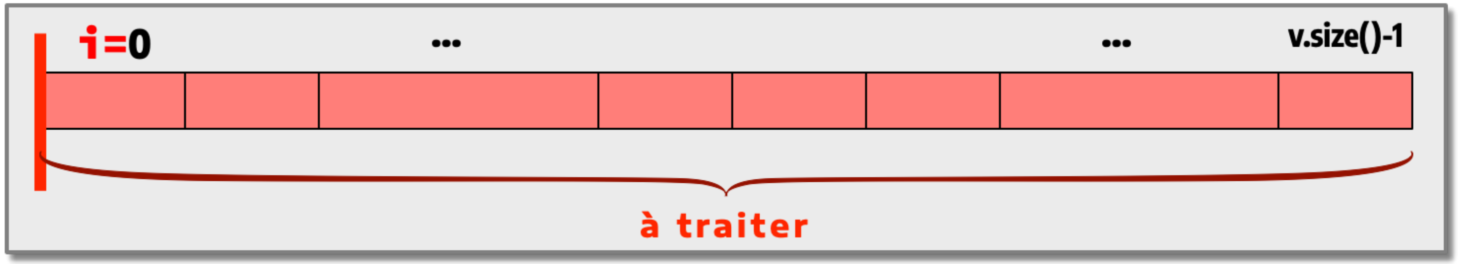


$$s = \sum v[0 \,..\, i-1]$$

invariant

Invariant et initialisation

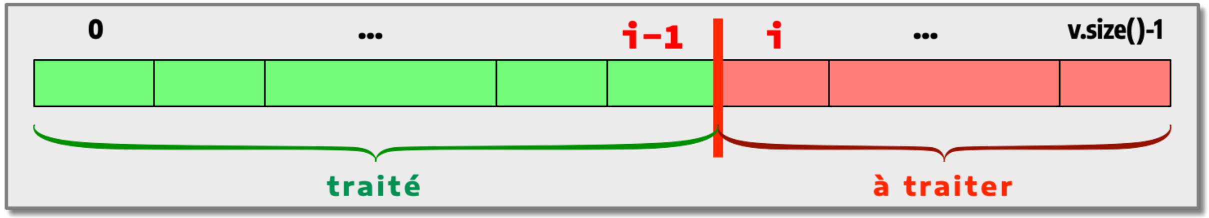
- Rappel de l'invariant : $s = \sum v[0 .. i-1]$
- Rappel du dessin de la situation initiale :



- La situation initiale est la situation dans laquelle on se trouve après l'initialisation :
  - **initialisation** de `i` pour commencer le parcours : `i = 0;`
  - s doit être initialisé pour y accumuler les valeurs des éléments de v
    - *en remplaçant la valeur de i dans l'invariant, on obtient*
      - $s = \sum v[0 .. -1]$ *{v[0 .. −1] est un vecteur vide inf > sup}*
    - *s est la somme des éléments d'un vecteur vide, la spécification du problème dit que cette somme vaut 0*
  - **initialisation** de `s` pour satisfaire l'invariant : `s = 0;`

Invariant et itération de parcours complet

- Rappel de l'invariant : $s = \sum v[0 .. i-1]$
- Rappel du dessin de la situation intermédiaire :



- Construction de l'itération
  - *On sait que* `i` *est initialisé à* `0`
  - *Le parcours étant complet, on doit poursuivre les traitements tant que* `i ≤ v.size()-1` *(ou* `i < v.size()`*) en vérifiant l'invariant*
- Condition de l'itération : `(i < v.size())`
- Corps de l'itération

1) traiter v[i] : `s = s + v.get(i);`
2) avancer : `i = i + 1;   // invariant vérifié`

Invariant et itération de parcours complet

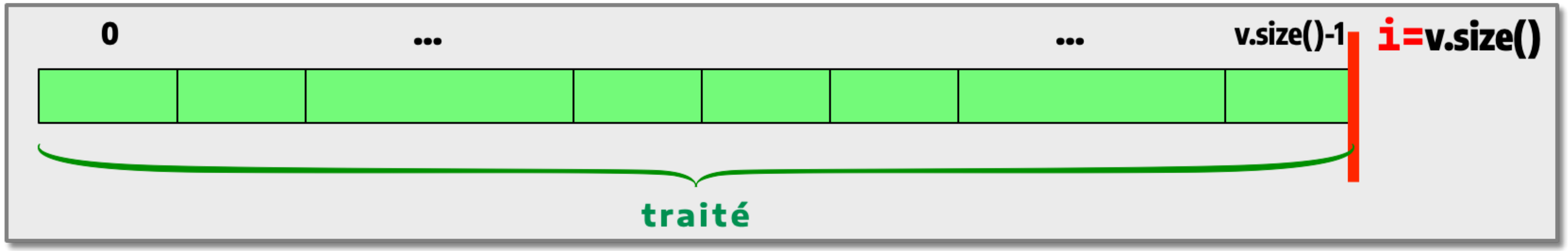
- Pour récapituler on a :

```
// i = 0 et s = 0 => invariant vérifié avant l'itération
// itération : parcours complet
while (i < v.size()) {
    s = s + v.get(i);         // traiter v[i]
    // s = Σv[0 .. i]
    i = i + 1;                // avancer
    // s = Σv[0 .. i-1] => invariant vérifié en fin de bloc
}
// i = v.size() : négation de la condition d'itération
```

- On voit que l'invariant est vérifié à la fin du bloc d'instructions de l'itération
- **C'est important !**

Invariant et production du résultat

- Rappel de l'invariant : $s = \sum v[0 \,..\, i-1]$
- Rappel du dessin de la situation finale



- Bilan
  - après l'itération `i` est égal à `v.size()`
  - en remplaçant la valeur de `i` dans l'invariant, on obtient :

    $s = \sum v[0 \,..\, v.size() - 1]$ `// somme des valeurs de v`

- Retour de la fonction : `return s;`

La fonction `getSommeVectInt`

```
private static int getSommeVectInt(ArrayList<Integer> v) {
// spécification {} => {résultat = somme des valeurs de v;
//                                 0 si v est vide}
  int i, s;                 // déclaration des deux variables
  // initialisation
  i = 0;
  s = 0;                    //somme d'un vecteur vide
  // s = Σv[0 .. i-1]  => invariant vérifié avant l'itération
  // itération
  while (i < v.size()) {               // parcours complet
    s = s + v.get(i);       // traiter v[i]
    // s = Σv[0 .. i]
    i = i + 1;              // avancer
    // s = Σv[0 .. i-1] => invariant vérifié à la fin du bloc
  }
  // i = v.size() : négation de la condition d'itération
  // s = Σv[0 .. v.size()-1], donc s = somme des éléments de v
  // production du résultat
  return s;
}
```

Classe d'utilisation

Classe

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

public class SommeVectInt {
  private static int getSommeVectInt(ArrayList<Integer> v) {
  // spécification {} => {résultat = somme des valeurs de v;
  //                                  0 si v est vide}

   -> voir planche précédente

  }

  public static void main(String[] args) {
     // déclaration et initialisation à partir d'une collection d'entiers
     ArrayList<Integer> unVectEnt = new ArrayList<>(Arrays.asList(12, 14, 4
                                                   45, 17, 23, 65, 18, 36, 8, 67));
     System.out.println("le vecteur : " + unVectEnt);
     int somme = getSommeVectInt(unVectEnt);
     System.out.println("Somme des éléments de v : " + somme);
  }
}
```

Trace

```
le vecteur : [12, 14, 4, 45, 17, 23, 65, 18, 36, 8, 67]
Somme des éléments de v : 309
```

Deuxième algorithme de parcours complet

NOMBRES D'ÉLÉMENTS SUPÉRIEUR À UNE VALEUR DONNÉE DANS UN VECTEUR D'ENTIERS (`Integer`)

Le problème

- Étant donné un vecteur **v** d'entiers représenté sous la forme d'un `ArrayList` de `Integer`
  - On veut écrire une fonction qui retourne le nombre d'éléments de **v** supérieurs à une valeur **`val`** donnée
  - Entête de la fonction à écrire :

```
private static int getNbSupVal(ArrayList<Integer> v,
                               int val)
// { v vide ou non } =>
// { résultat = nombres de valeurs de v
//              supérieures ou égales à val de v }
```

Première analyse du problème

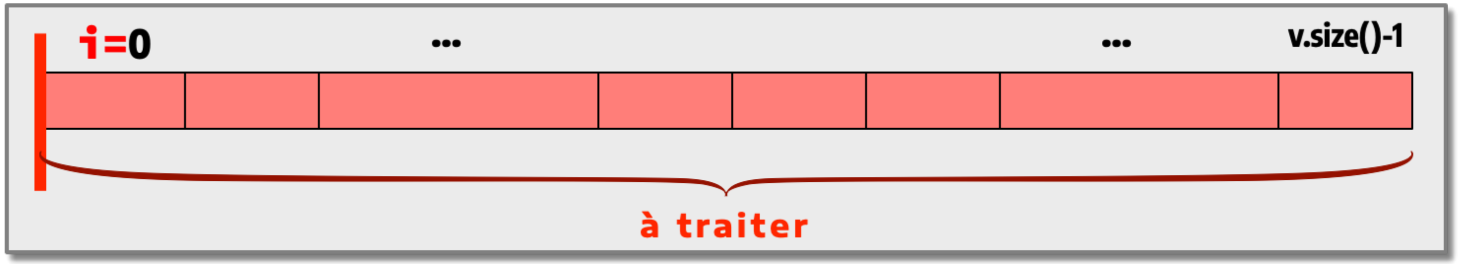
- Pour résoudre ce problème, on propose 3 étapes :
  1) initialiser les variables de l'algorithme
  2) faire un **parcours complet** (traiter tous les éléments) de **v** en partant du premier indice (`0`) jusqu'au dernier indice (`v.size()-1`) en comptant le nombre de valeurs supérieures ou égales à `val`
     - c'est donc une itération qui se fera avec un indice `i` qui va parcourir l'intervalle `[0 .. v.size()-1]`
     - à chaque pas de l'itération on va :
       - traiter v[`i`] (vérifier s'il faut le compter ou non)
       - puis avancer (incrémenter `i`)
  3) retourner, comme résultat, le comptage

*on propose donc un **algorithme itératif de parcours complet***

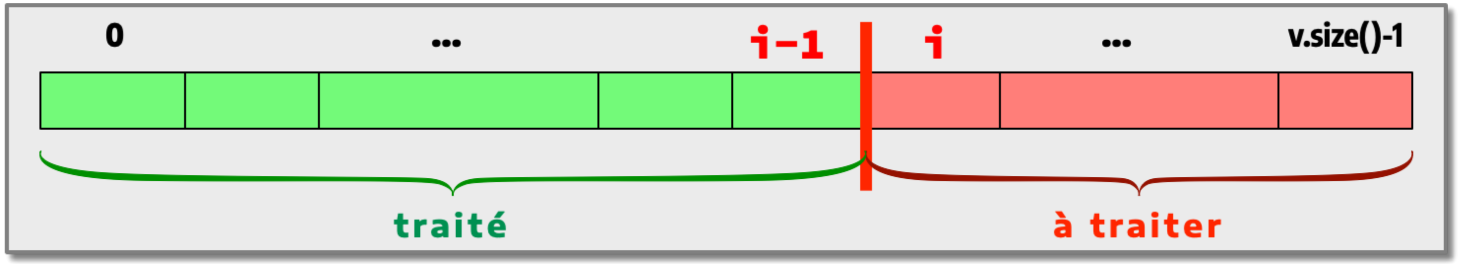
Effort de formalisation (parcours complet)

Points intéressants dans le déroulement de l'algorithme ?

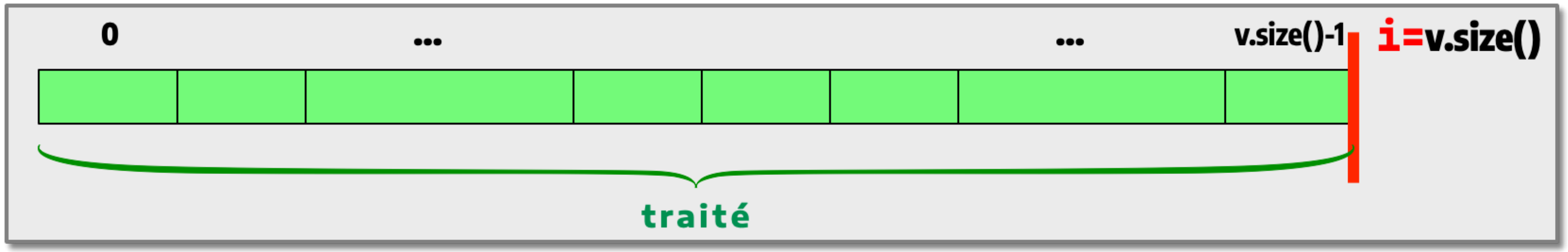
- **situation initiale**
  - `i = 0`



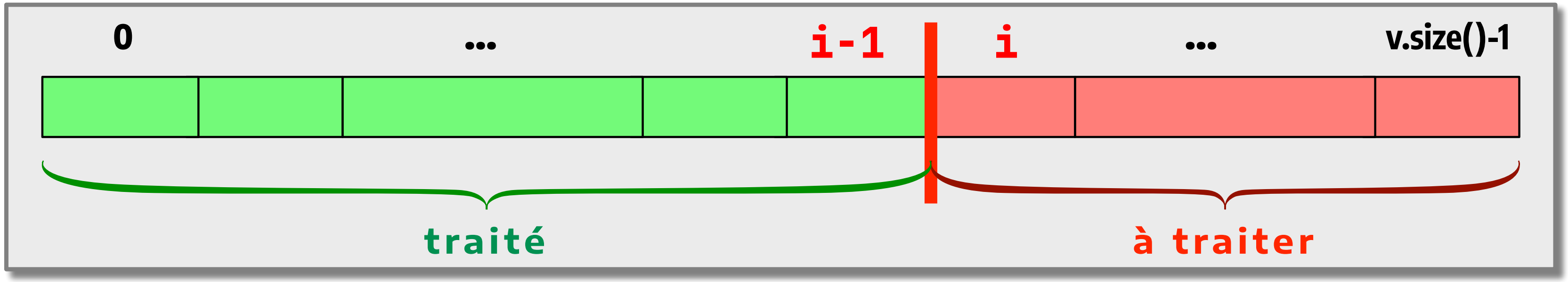
- **situation intermédiaire de parcours complet**
  - **v traité jusqu'à `i - 1`**
  - **v[`i`] à traiter**



- **situation finale de parcours complet**
  - **`i` = v.size()**
  - **v entièrement traité**



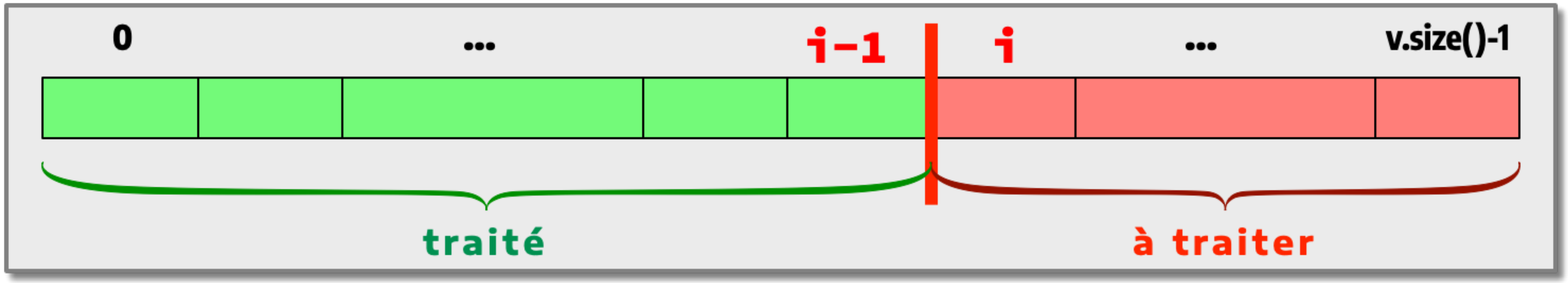
Qu'a-t-on fait sur la zone traitée ?



- Pour répondre au problème posé, on exprime ce qu'a fait l'algorithme sur la zone traitée :
  - l'algorithme a compté le nombre d'éléments > `val` sur l'intervalle [0 .. `i`-1]
  - ce nombre est mémorisé dans une variable que l'on nomme **nb**
- En formalisant :
  - $\text{nb} = \text{nombre d'élements} > \text{val dans v}[0 \,..\, \text{i}-1]$
- On appelle cette formule l'invariant de l'algorithme
  - l'invariant permet de construire 1) l'initialisation, 2) l'itération et 3) la production du résultat

Récapitulons

- Situation intermédiaire complètement décrite :

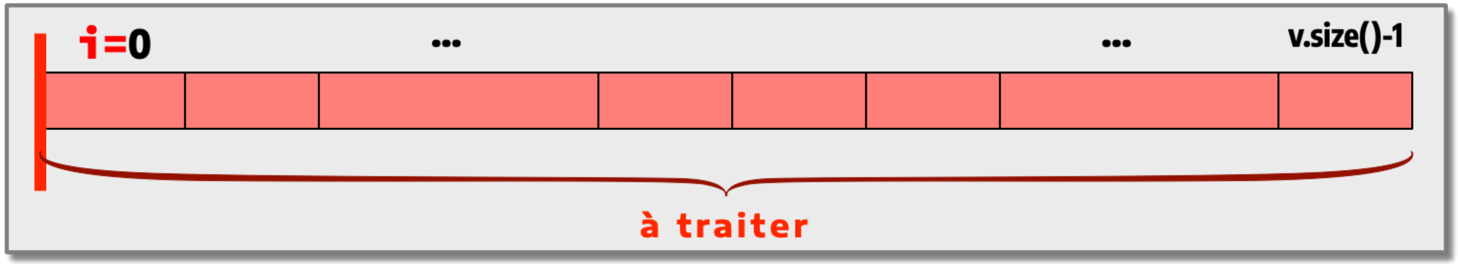


nb = nombre d'élements > val dans v[0 .. i−1]

invariant

Invariant et Initialisation

- Rappel de l'invariant : nb = nombre d'élements > val dans v[0 .. i−1]
- Rappel du dessin de la situation initiale



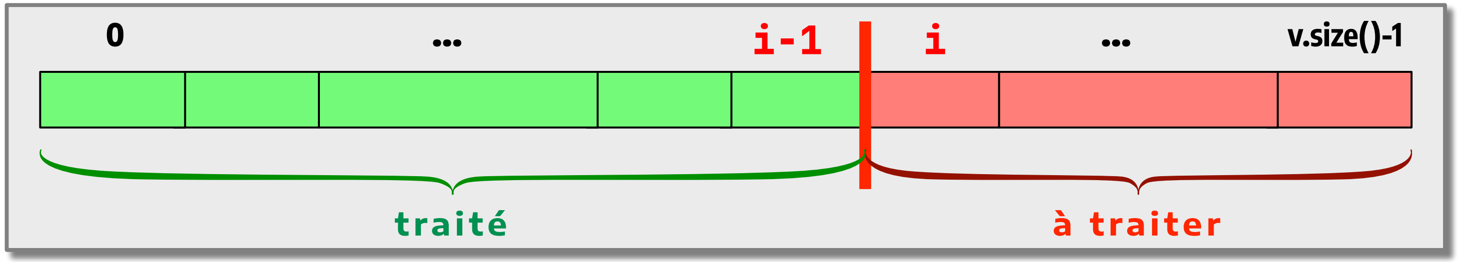
- La situation initiale est la situation dans laquelle on se trouve <u>après</u> l'initialisation :
  - initialisation de `i` pour commencer le parcours : `i = 0;`
  - **nb** doit être initialisé pour compter les valeurs > `val`

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

    nb = nombre d'élements > val dans v[0 .. −1]   // v[0 .. −1] est un vecteur vide
  - initialisation de **nb** pour satisfaire l'invariant : `nb = 0;`

Invariant et itération de parcours complet

- Rappel de l'invariant : nb = nombre d'éléments > `val` dans `V[0..i-1]`
- Rappel du dessin de la situation intermédiaire



- Construction de l'itération
  - *On sait que `i` a été initialisé à `0`*
  - *Le parcours étant complet, on doit poursuivre les traitements tant que `i <= V.size()-1` (`i < V.size()`) en vérifiant l'invariant*

- Condition de l'itération : `(i < V.size())`
- Corps de l'itération (bloc d'instructions):

1) traiter **V**[i] : `if (V.get(i) > val) {nb = nb+1};`
2) avancer : `i = i + 1;   // invariant vérifié`

Invariant et itération de parcours complet

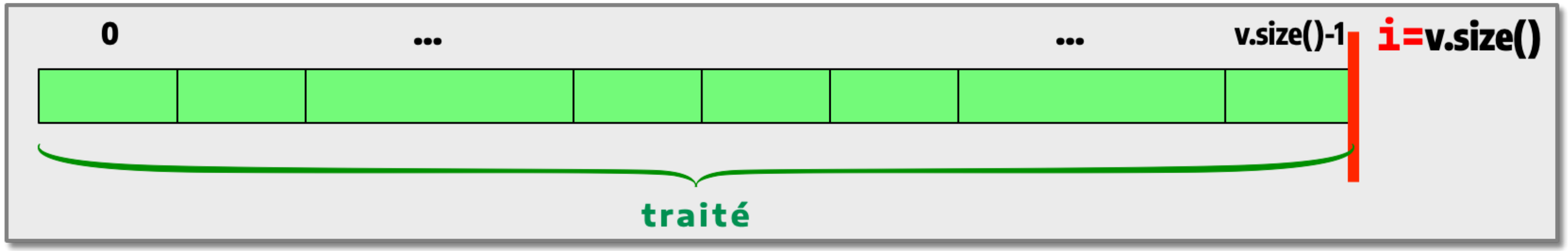
- Pour récapituler on a :

```
// i = 0 et nb = 0 => invariant vérifier avant l'itération
// itération de parcours complet
while (i < v.size()) {
    // nb = nombre d'éléments > val dans v[0 .. i-1] (invariant)
    if (v.get(i) > val) {       // traiter v[i]
      nb = nb + 1;
    }                           // sinon on ne fait rien
    // nb = nombre d'éléments > val dans v[0 .. i]
    i = i + 1;                  // avancer
    // nb = nombre d'éléments > val dans v[0 .. i-1] (invariant)
}
// i = v.size() : négation de la condition d'itération
```

- On voit que l'invariant est vérifié au début et à la fin du bloc d'instructions de l'itération
- **C'est important !**

Invariant et production du résultat

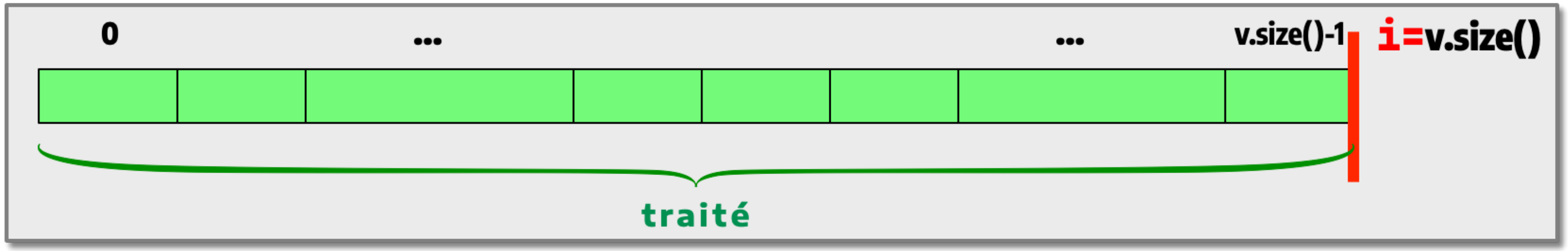
- Rappel de l'invariant : nb = nombre d'élements > val dans v[0 .. i−1]
- Rappel du dessin de la situation finale



- Bilan
  - on a : i = v.size()
  - en remplaçant la valeur de i dans l'invariant, on obtient
    - nb = nombre d'élements > val dans v[0 .. v.size()−1]
  - nb est le nombres d'éléments > val de v (c'est le résultat)
  - la fonction va donc retourner nb :
    - `return nb;`

Invariant et production du résultat

- Rappel de l'invariant : `nb` = nombre d'éléments > `val` dans `V[0..i-1]`
- Rappel du dessin de la situation finale



- Bilan

  ** après l'itération `i` est égal à `V.size()`*

  ** en remplaçant la valeur de `i` dans l'invariant, on obtient :*

  nb = nombre d'éléments > `val` dans `V[0..V.size()-1]` // c'est le résultat

- Retour de la fonction : **`return nb;`**

La fonction getSommeVectInt

```
private static int getSommeVectInt(ArrayList<Integer> v) {
// spécification {} => {résultat = nombre d’éléments > val de v}
  int i, nb;                    // déclaration des deux variables
  // initialisation
  i = 0;
  nb = 0;
  // nb = nombre d’éléments > val de v[0 .. -1]   -> vecteur vide

  // itération
  while (i < v.size()) {        // parcours complet
    // nb = nombre d’éléments > val dans v[0 .. i-1] (invariant)
    if (v.get(i) > val) {       // traiter v[i]
      nb = nb + 1;
    }                           // sinon on ne fait rien
    // nb = nombre d’éléments > val dans v[0 .. i]
    i = i + 1;                  // avancer
    // nb = nombre d’éléments > val dans v[0 .. i-1] (invariant)
  }
  // i = v.size() : négation de la condition d’itération
  // s = Σv[0 .. v.size()-1]   -> somme de tous les éléments de v

  // production du résultat
  return nb;
}
```

Classe d'utilisation

Classe

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

public class NbSupVal {
  private static int getNbSupVal(ArrayList<Integer> v) {
  // spécification {} => {résultat = somme des valeurs de v;
  //                                 0 si v est vide}

   -> voir planche précédente

  }

  public static void main(String[] args) {
    // déclaration et initialisation à partir d'une collection d'entiers
    ArrayList<Integer> unVectEnt = new ArrayList<>(Arrays.asList(12, 14, 4
                                                   45, 17, 23, 65, 18, 36, 8, 67));
    System.out.println("le vecteur : " + unVectEnt);
    System.out.println("Nombre d'éléments > 20 : " +
                                               getNbSupVal(unVectEnt, 20));
    System.out.println("Nombre d'éléments > 45 : " +
                                               getNbSupVal(unVectEnt, 45));
  }
}
```

Trace

```
le vecteur : [12, 14, 4, 45, 17, 23, 65, 18, 36, 8, 67]
Nombre d'éléments > 20 : 5
Nombre d'éléments > 45 : 2
```

CONCLUSION

Répondre à un problème avec itération

1) Faire des dessins pour les situations
   - initiale
   - intermédiaire
   - finale
2) Trouver l'invariant avec la situation intermédiaire
3) Utiliser l'invariant pour écrire
   - l'initialisation des variables nécessaires
   - la condition de l'itération et son corps
   - la production du résultat si nécessaire (fonction)

- Pourquoi faire cela si le problème est trivial ?
  - pour prendre de bonnes habitudes
  - pour acquérir des automatismes
  - pour résoudre plus facilement des problèmes plus difficiles

Modèle du parcours complet sur v

- `while` (présenté sur les planches)
  - un indice `i` initialisé à 0 : `int i = 0;`
  - une condition d'itération : `i < v.size()`
    - à la sortie de l'itération on a : i = v.size()
      (le vecteur est entièrement traité)
- Le modèle

```
int i = 0;
//autres déclarations de variables et initialisations
…
while (i < v.size()) {
    // traiter v[i] -> v.get(i) en Java sur un ArrayList<E>
    …
    // avancer
    i = i + 1;
}
```

Vérification de l’invariant dans l’algorithme

```
// initialisation
…

// invariant vérifié après l'initialisation

// itération
while (i < v.size()) {
    // invariant vérifié en début du corps de l'itération

    // traiter v[i] -> v.get(i) en Java sur un ArrayList<E>
    …
    // avancer
    i = i + 1;

    // invariant vérifié en fin du corps de l'itération
}
// condition d'itération devenue fausse (i == v.size())
// invariant vérifié après l'itération

// production du résultat pour une fonction
…
```