Compléments de Programmation

Benoit Donnet
Année Académique 2024 - 2025

LIÈGE université

Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- Chapitre 4: Récursivité
- Chapitre 5: Types Abstraits de Données
- Chapitre 6: Listes
- Chapitre 7: Piles
- Chapitre 8: Files
- Chapitre 9: Elimination de la Récursivité

Agenda

- Chapitre 2: Construction de Programme
  - Schéma Global
  - Approche Constructive
  - Exemples

Agenda

- Chapitre 2: Construction de Programme
  - Schéma Global
  - Approche Constructive
  - Exemples

Schéma Global

- Il est (très) difficile d'écrire un programme correct du premier coup, de la première à la dernière ligne
- Il est préférable d'adopter une *démarche méthodologique*
- **Schéma méthodologique**

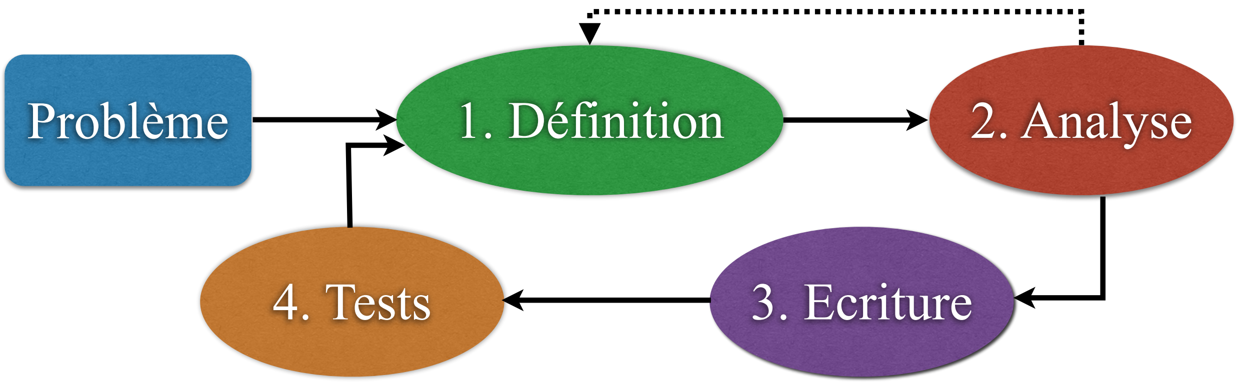


Schéma Global (2)

- 1ère étape: *Définition du problème*
  - définir précisément ce qu'on attend du programme
  - prendre connaissance des informations nécessaires à la résolution du problème
    - ✓ quelles sont les données en entrée?
    - ✓ quels sont les résultats attendus et sous quelle forme?
  - si problème de petite taille, peut mener à l'écriture de l'interface
    - ✓ prototype
    - ✓ spécification

Schéma Global (3)

- 2ème étape: *Analyse du problème*
  - découper le problème en parties plus petites et plus facile à appréhender
    - **découpe en sous-problèmes** (ou **approche systémique**)
      - chaque sous-problème pourra être résolu indépendamment
      - il est possible de généraliser un sous-problème (cfr. INFO0946 et INFO0030)
      - un sous-problème peut admettre plusieurs solutions
  - structurer le problème
    - comment les sous-problèmes s'emboîtent
  - pour chaque sous-problème, éventuellement recommencer l'étape 1

Schéma Global (4)

- 3ème étape: *Ecriture du code*
  - écriture des Invariants et Fonctions de Terminaison
  - implémentation des différents sous-problèmes
  - mise en commun des sous-problèmes
- 4ème étape: *Tests*
  - vérifier que l'implémentation résout bien le problème
    - tests d'implémentation
    - tests d'intégration
    - tests unitaires
      - cfr. INFO0030
  - peut nécessiter de revenir à la 1ère étape en cas d'erreur

Schéma Global (5)

- Typiquement, les différents sous-problèmes sont implémentés dans des fonctions/procédures
- Comment s'assurer d'une implémentation rigoureuse d'une fonction/procédure?
- Différentes méthodes de développement
  - *programmer + tester*
    - ✓ "testing can only show the presence of bugs, not their absence"
      - E. W. Dijsktra. "Notes on Structured Programming". August 1969. [http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF]
  - *programmer + prouver*
    - ✓ augmente la quantité de travail
    - ✓ difficile
    - ✓ quid si on ne réussit pas prouver?
  - *approche constructive*
    - ✓ D. G. Kourie, B. W. Watson. *The Correctness-By-Construction Approach to Programming*. Springer Ed. January 2012. [https://link.springer.com/book/10.1007/978-3-642-27919-5]

Agenda

- Chapitre 2: Construction de Programme
  - Schéma Global
  - Approche Constructive
    - ✓ Principe
    - ✓ Structure Séquentielle
    - ✓ Structure Conditionnelle
    - ✓ Structure Itérative
  - Exemples

Principe

- On peut voir le code comme un triplet
  - {P} `Code` {Q}
    - ✓ `Code`, un fragment de code exécutable
    - ✓ *P*, une précondition
    - ✓ *Q*, une postcondition
- **<u>Triplet de Hoare</u>**
  - R. W. Floyd. *Assigning Meanings to Programs*. In Proc. American Mathematical Society Symposia on Applied Mathematics. 19, pg. 19-31. 1967
  - C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*. In Communications of the ACM. 12(10), pg. 576-580. October 1969.
- Si `Code` débute son exécution dans un état satisfaisant *P*, alors
  - `Code` termine au bout d'un temps fini
  - et `Code` termine dans un état satisfaisant *Q*

Principe (2)

- *P* et *Q* sont des prédicats
- *P* et *Q* portent sur
  - les valeurs courantes des variables de la fonction/ procédure
  - les inputs/outputs de cette fonction/procédure
- {P} `Code` {Q} est valide ssi
  - dans tous les cas où l'on exécute `Code` lorsque *P* est valide
  - alors, après cette exécution, *Q* est toujours vrai

Principe (3)

- Exemples

{P} Code {Q}

```
{10 ≤ x ≤ 20} x++; {11 ≤ x ≤ 21}
```

```
{x ∈ [-2^31, 2^31-1]}
if(x<0)
  x=0;
{x ∈ [0, 2^31-1]}
```

{P} Code {Q}

```
{True} n'importe quel code {True}
```

{P} Code {Q}

Principe (4)

- **<u>Approche constructive</u>**
  - dériver des conditions et des actions d'un fonction/procédure qu'on construit
  - à partir des spécifications et assertions résultant d'étapes de raisonnement
- Nécessite d'écrire au préalable les spécifications
  - si possible de manière formelle
- Permet de faire le lien entre spécifications et Invariant de Boucle
  - l'Invariant devra être écrit de manière formelle

Principe (5)

- Cette approche doit être faite pour des instructions
  - séquentielles
    - ✓ **<u>structure séquentielle</u>**
  - conditionnelles
    - ✓ **<u>structure conditionnelle</u>**
  - itératives
    - ✓ **<u>structure itérative</u>**

Structure Séquentielle

- Principe
  - chercher des instructions $\texttt{I}_1$, $\texttt{I}_2$, ..., $\texttt{I}_{k+1}$
  - chercher des assertions intermédiaires $Q_1, Q_2, ..., Q_k$
  - telles que

$$\{\text{Pré}\}\ \texttt{I}_1\ \{Q_1\}$$

$$\{Q_1\}\ \texttt{I}_2\ \{Q_2\}$$

$$\dots$$

$$\frac{\{Q_{k-1}\}\ \texttt{I}_k\ \{Q_k\}}{\{Q_k\}\ \texttt{I}_{k+1}\ \{\text{Post}\}}$$

Structure Séquentielle (2)

- Le séquencement résulte, généralement, d'un processus de décomposition en sous-objectifs
- L'ordre est guidé par
  - la dépendance entre résultats intermédiaires
  - la structure des données à traiter

Structure Séquentielle (3)

- Exemple
  - permutation de deux variables
    - ✓ j'ai deux variables entières, `x` et `y`
    - ✓ je veux placer le contenu de `x` dans `y` et le contenu de `y` dans `x`
- Spécification

```
/*
 * @pre: /
 * @post: x et y ont été permutés
 */
```

Structure Séquentielle (4)

- Modification des spécifications
  - objectif?
    - ✓ formalisme pouvant être utilisé dans les assertions intermédiaires

```
/*
 * @pre: x = x0 ∧ y = y0
 * @post: x = y0 ∧ y = x0
 */
```


Structure Séquentielle (6)

- Construction instruction par instruction

```
{x = x0 ∧ y = y0}                      {Pré}
int tmp = x;
{x = x0 ∧ tmp = x0 ∧ y = y0}           {Q1}
x = y;
{x = y0 ∧ tmp = x0 ∧ y = y0}           {Q2}
y = tmp;
{x = y0 ∧ tmp = x0 ∧ y = tmp}
=> {x = y0 ∧ y = x0}                   {Post}
```

Structure Conditionnelle

- Principe
  - chercher des instructions $\texttt{I}_1$, $\texttt{I}_2$
  - telles que

$$\frac{\{\text{Pré} \wedge B\}\ \texttt{I}_1\ \{\text{Post}\} \qquad \{\text{Pré} \wedge \neg B\}\ \texttt{I}_2\ \{\text{Post}\}}{\{\text{Pré}\}\ \textbf{if}\ \texttt{B}\ \textbf{then}\ \texttt{I}_1\ \textbf{else}\ \texttt{I}_2\ \{\text{Post}\}}$$

- Il peut être nécessaire de distinguer {Pré ∧ B} et {Pré ∧ ¬ B}

Structure Conditionnelle (2)

- Exemple
  - le maximum de 2 entiers
    - ✓ j'ai deux variables entières, `x` et `y`
    - ✓ je veux placer le maximum de `x` et `y` dans `z`
- Spécification

```
/*
 * @pre: /
 * @post: z contient le maximum de x et y
 */
```

Structure Conditionnelle (3)

- Modification des spécifications
  - objectif?
    - ✓ formalisme pouvant être utilisé dans les assertions intermédiaires

```
/*
 * @pre: x = x0 ∧ y = y0
 * @post: z = max(x0, y0) ∧ x = x0 ∧ y = y0
 */
```


Structure Conditionnelle (4)

- Construction instruction par instruction

```
{x = x0 ∧ y = y0}
if(x > y) B
  {x = x0 ∧ y = y0 ∧ x > y}          {Pré ∧ B}
  z = x;
  {z = x ∧ x = x0 ∧ y = y0 ∧ x > y}
  => {z = max(x0, y0) ∧ x = x0 ∧ y = y0}
else
  {x = x0 ∧ y = y0 ∧ ¬(x > y)}       {Pré ∧ ¬ B}
  ⇒ {x = x0 ∧ y = y0 ∧ x ≤ y}
  z = y;
  {z = y ∧ x = x0 ∧ y = y0 ∧ x ≤ y}
  => {z = max(x0,y0) ∧ x = x0 ∧ y =y0}
```

Structure Itérative

- Souvent, la forme d'une assertion ou la combinaison Pré-Post suggère l'introduction d'une boucle
- Dans ce cas
  1. déterminer une situation générale caractérisée par un Invariant {Inv}
  2. Construire une initialisation `INIT` tel que
     {Pré} `INIT` {Inv}
  3. trouver le Critère d'Arrêt ¬`B`
  4. établir que l'on peut faire progresser la situation générale vers la situation finale

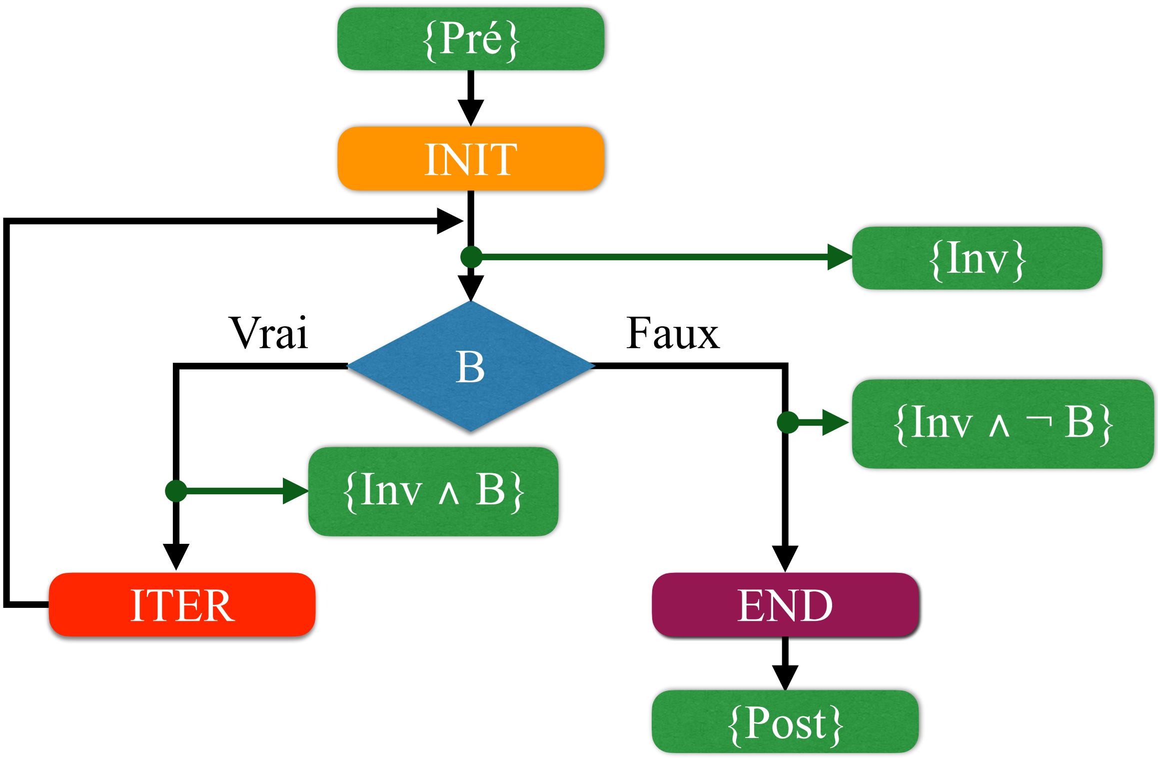
Structure Itérative (2)

4. Comment établir que l'on peut faire progresser la situation générale vers la situation finale?
   a. découverte des instructions `ITER` telles que
      {Inv ∧ B} `ITER` {Inv}
   b. découverte d'une fonction *f* garantissant la terminaison
   c. découverte des instructions `END` telles que
      {Inv ∧ ¬ B} `END` {Post}

Structure Itérative (3)

- Il est important de noter que
  - les points 1 & 3 portent sur des situations
  - les points 2 & 4 concernent des actions
- L'Invariant est l'étape clé autour de laquelle s'articule la conception des boucles
  - c'est la colle qui lie les autres constituants entre eux
- La construction d'une boucle commence <u>toujours</u> par la recherche d'un Invariant

Structure Itérative (4)



Structure Itérative (5)

- Comment déterminer un Invariant?
  - intuition
  - raisonnement inductif
  - combiner Input/Précondition et Output/Postcondition
  - travailler sur l'Output/Postcondition pour en tirer une situation générale
    - ✓ **éclatement de la Postcondition**
    - ✓ **constant relaxation**
      - C. A. Furia, B. Meyer, S. Velder. *Loop Invariants: Analysis, Classification, and Examples*. In ACM Computing Surveys, 46(3), pg. 1-51. January 2014.
      - G. Brieven, S. Liénardy, L. Malcev, B. Donnet. *Graphical Loop Invariant Based Programming*. In Proc. Formal Method Teaching (FMTea). March 2023.

Structure Itérative (6)

- Étapes pour l'éclatement de la Postcondition

A. formaliser le problème à l'aide de notation
  - ✓ cfr. Chap. 1, Slides 34 → 39

B. spécifier la fonction/procédure
  - ✓ spécifications formelles
  - ✓ utiliser les notations introduites en 1.

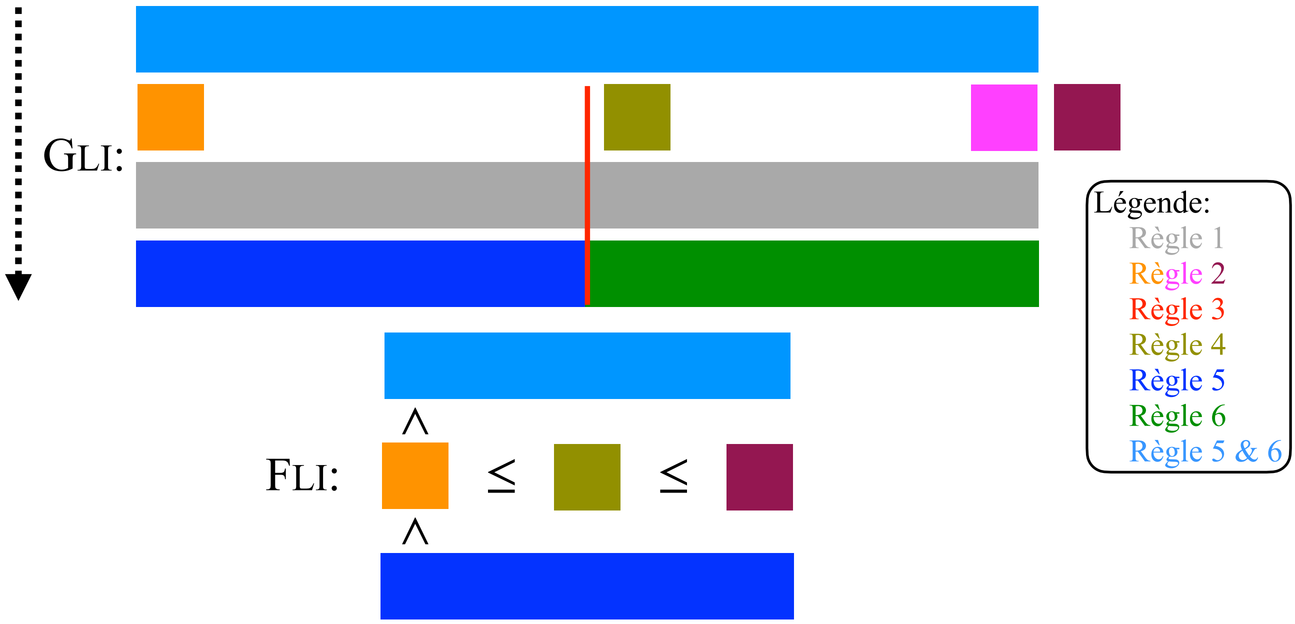
C. représenter graphiquement la Postcondition
  - ✓ s'appuyer sur les règles 1, 2 et 5 du GLI

D. dériver le GLI de l'étape 3
  - ✓ utiliser les notations introduites en 1 sur un résultat partiel

E. dériver le FLI de l'étape 4
  - ✓ utiliser les notations introduites en 1 sur un résultat partiel

Structure Itérative (7)

- Comment passer de l'étape D à l'étape E?
  - GLI simple avec propriété(s) conservée(s)



Structure Itérative (8)

- Exemple
  - calcul d'une puissance
    - ✓ j'ai deux variables entières, `x` et `n`
    - ✓ je veux calculer la valeur x^n
- Trouver un Invariant

A. Formalisation du problème
  - $power(x, n) \equiv x^n$

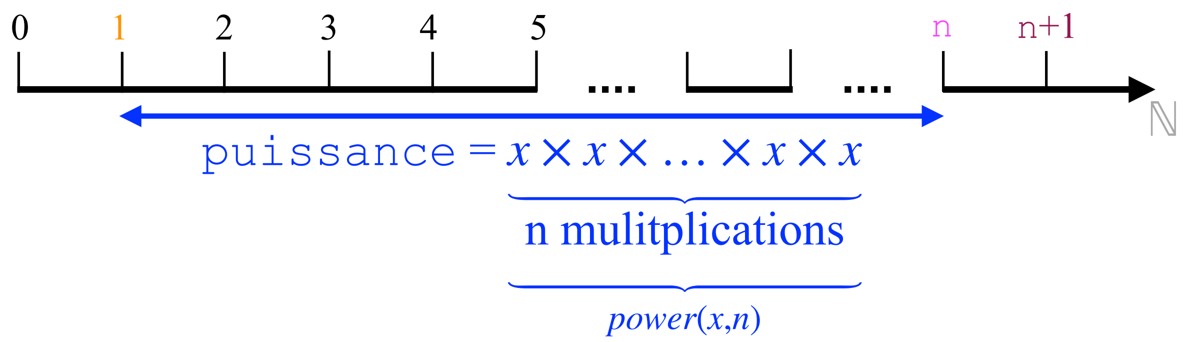
B. Spécifications

```
/*
 * @pre: x = x0 ∧ n ≥ 0
 * @post: n = n0 ∧ x = x0 ∧ puissance = power(x, n)
 */
int puissance(int x, int n);
```

Structure Itérative (9)

- Exemple (cont')

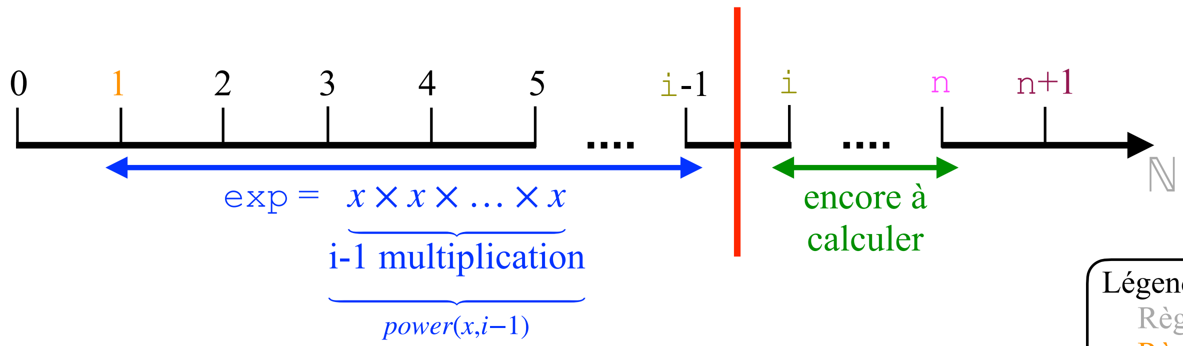
C. Représentation graphique de la Postcondition



Structure Itérative (10)

- Exemple (cont')

D. GLI

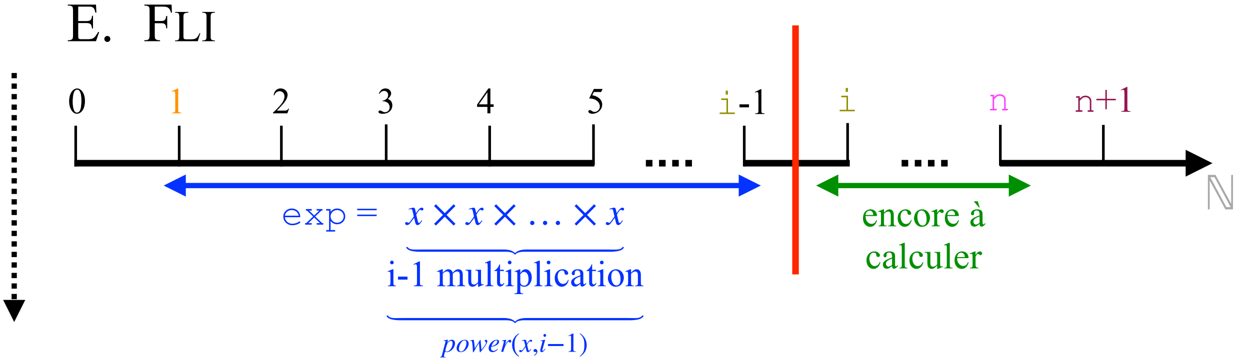


Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

Structure Itérative (11)

- Exemple (cont')

E. FLI



$$1 \leq i \leq n+1$$
$$\wedge$$
$$exp = power(x, i-1)$$
$$\wedge$$
$$x = x_0 \wedge n = n_0$$

Structure Itérative (12)

2. Trouver l'initialisation
   - on part de la Précondition
     - ✓ Pré $\equiv x = x_0 \wedge n \geq 0$
   - on doit arriver à une situation qui rende vrai l'Invariant
     - ✓ Inv $\equiv x = x_0 \wedge n = n_0 \wedge exp = power(x, i-1) \wedge 1 \leq i \leq n+1$
   - soit
     - ✓ déclarer un indice `i` et l'initialiser à la valeur 1
     - ✓ déclarer une variable `exp` et l'initialiser à la valeur 1

```
{x = x0 ∧ n ≥ 0}                          {Pré}
                                    INIT
int i = 1;
{x = x0 ∧ n ≥ 0 ∧ i = 1}                  {Q1}
int exp = 1;
{x = x0 ∧ n ≥ 0 ∧ i = 1 ∧ y = 1}          {Q2}
⇒ {x = x0 ∧ n = n0 ∧ exp = power(x, i-1) ∧ 1≤i≤n+1}   {Inv}
```

Structure Itérative (13)

3. Trouver le Critère d'Arrêt (¬ `B`)
   - on peut se baser sur l'Invariant, notamment l'élément qui est supposé varier à chaque tour
     - ✓ indice `i` varie entre 1 et `n`
     - ✓ quand il vaudra `n`+1, la boucle devra être terminée
       - confirmé par l'Invariant!!
     - ✓ $\neg B \equiv i = n + 1$

```
while(i<=n)  B
```


Structure Itérative (14)

4. Trouver des instructions qui font progresser la boucle
   a. trouver `ITER`
      - ✓ le raisonnement suggère le contenu d'`ITER`
        - la variable `exp` doit être mise à jour avec une nouvelle multiplication
        - la variable `i` doit être incrémentée d'une unité
   b. trouver la Fonction de Terminaison, *f*
      - ✓ *n-i+1*

```
{Inv ≡ x = x0 ∧ n = n0 ∧ exp = power(x, i-1)) ∧ 1≤i≤n+1}
while(i<=n){
  {Inv ∧ B ≡ x = x0 ∧ n = n0 ∧ exp = power(x, i-1) ∧ 1≤i≤n}
  exp *= x;
  {x = x0 ∧ n = n0 ∧ exp = power(x, i) ∧ 1 ≤ i ≤ n}
  i++;
  {Inv ≡ x = x0 ∧ n = n0 ∧ exp = power(x, i-1) ∧ 1≤i≤n+1}
}//fin while
```

Structure Itérative (15)

4. Trouver des instructions qui font progresser la boucle (suite)
   a. trouver ITER
   b. trouver la Fonction de Terminaison, *f*
   c. trouver END
      - ✓ rien à faire
      - ✓ il faut seulement s'assurer que {Inv ∧ ¬ B} ⇒ {Post}

```
{Inv ≡ x = x0 ∧ n = n0 ∧ exp = power(x, i-1) ∧ 1≤i≤n+1}
while(i<=n){
  //code
}//fin while
{Inv ∧ ¬ B ≡ x = x0 ∧ n = n0 ∧ exp = power(x,i-1)  ∧
1≤i≤n+1 ∧ i=n+1}
⇒ {x = x0 ∧ n = n0 ∧ exp = power(x, n+1-1)}
⇒ {x = x0 ∧ n = n0 ∧ puissance = power(x, n)}   {Post}
```


Structure Itérative (16)

- Code complet

```
int puissance(int x, int n){
  {Pré ≡ x = x0 ∧ n ≥ 0}
  int i = 1;
  int exp = 1;

  {Inv ≡ x = x0 ∧ n = n0 ∧ y = power(x, i-1) ∧ 1≤i≤n+1}
  while(i<=n){
    exp *= x;
    i++;
  }//fin while

  return exp;
  {Post ≡ x = x0 ∧ n = n0 ∧ puissance = power(x, n)}
}//fin puissance()
```

Agenda

- Chapitre 2: Construction de Programme
  - Schéma Global
  - Approche Constructive
  - Exemples
    - ✓ Somme des Eléments d'un Tableau
    - ✓ Nombre d'Inversions
    - ✓ Problème des Plateaux
    - ✓ Tri par Bulles

Somme Tableau

- Soit `a[0, ..., n-1]`, un tableau initialisé à `n` valeurs entières
- Ecrire une fonction qui retourne la somme des éléments de `a`

Somme Tableau (2)

- Formalisation du problème
  - Soit *S(a, n)*, une notation telle que
    - ✓ *a* est un tableau d'entiers
    - ✓ *n* est la taille du tableau ($n \geq 0$)
  - On a
    - ✓ $S(a, n) \equiv \sum_{i=0}^{n-1} a[i]$
  - Soit SS(a, i, n), une notation pour la somme des éléments d'un sous-tableau de a
    - ✓ $SS(a, i, n) \equiv 0 \leq i \leq n, \sum_{j=0}^{i-1} a[j]$

Somme Tableau (3)

- Définition du problème
  - Input
    - ✓ `a`, tableau
    - ✓ `n`, taille du tableau
  - Output
    - ✓ la somme des éléments du tableau, $S(a, n)$
  - Caractérisation des Inputs
    - ✓ `a` est un tableau d'entiers
      - `int *a;`
    - ✓ `n` est une valeur entière
      - `int n;`

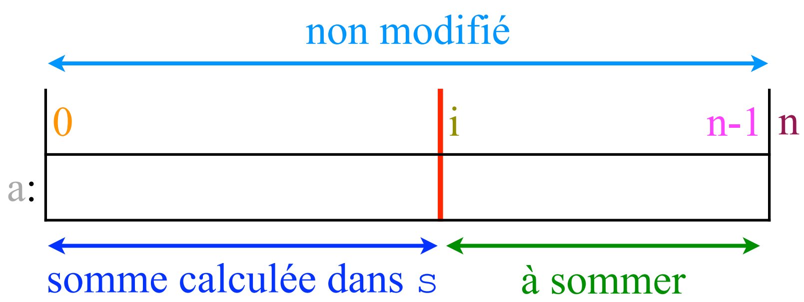
Somme Tableau (4)

- Spécification

```
/*
 * @pre: a initialisé ∧ n ≥ 0
 * @post: a = a0 ∧ n = n0 ∧ somme_tableau = S(a, n)
 */
int somme_tableau(int *a, int n);
```


Somme Tableau (5)

- Invariant Graphique (GLI)



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Invariant Formel (FLI)

$a = a_0 \wedge n = n_0$

$\wedge$

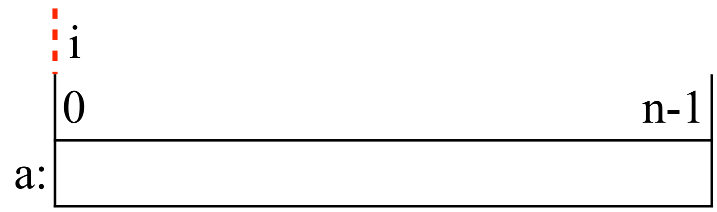
$0 \leq i \leq n$

$\wedge$

$s = SS(a, i, n)$

Somme Tableau (6)

- {Pré} `INIT` {Inv}
  - situation à établir



`s` = SS(a, 0, n) = 0

Somme Tableau (7)

- {Pré} `INIT` {Inv}
  - Instructions

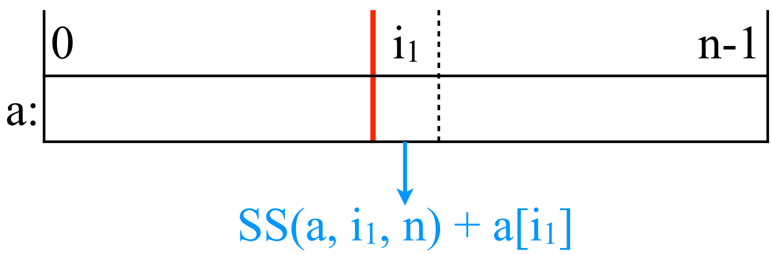
```
{Pré ≡ a initialisé ∧ n ≥ 0}
int i = 0;
{a = a_0 ∧ n = n_0 ∧ i=0}
int s = 0;
{a = a_0 ∧ n = n_0 ∧ i=0 ∧ s = 0}
⇒ {a = a_0 ∧ n = n_0 ∧ i=0 ∧ s = SS(a, 0, n)}
⇒ {a = a_0 ∧ n = n_0 ∧ i=0 ∧ s = SS(a, i, n)}
⇒ {Inv ≡ a = a_0 ∧ n = n_0 ∧ 0 ≤ i ≤ n ∧ s = SS(a, i, n)}
```

Somme Tableau (8)

- Critère d'Arrêt ($\neg$B)
  - `i == n`
- {Inv $\wedge$ $\neg$B} `END` {Post}
  - rien à faire, hormis retourner `s`
  - établir {Inv $\wedge$ $\neg$B} $\Rightarrow$ {Post}
    - ✓ $a = a_0 \wedge n = n_0 \wedge 0 \leq i \leq n \wedge s = SS(a, i, n) \wedge i = n$
    - ✓ $a = a_0 \wedge n = n_0 \wedge somme\_tableau = SS(a, n, n) = S(a, n)$

Somme Tableau (9)

- {Inv $\wedge$ B} `ITER` {Inv}



Il faut établir

`s = s + a[i`$_1$`]`

faire progresser `i` (`i = i`$_1$`+1`)

Somme Tableau (10)

- {Inv ∧ B} `ITER` {Inv}
  - instructions

```
{Inv ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n ∧ s = SS(a, i, n)}
while(i<n){
  {Inv ∧ B ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n-1 ∧ s = SS(a,i,n)}
  s += a[i];
  {a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n-1 ∧ s = SS(a, i+1, n)}
  i++;
  {Inv ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n ∧ s = SS(a, i, n)}
}//fin while
```


Somme Tableau (11)

- Fonction de Terminaison
  - *n-i*
- Code complet

```
int somme_tableau(int *a, int n){
  {Pré ≡ a initialisé ∧ n ≥ 0}
  int i = 0;
  int s = 0;
  {Inv ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n ∧ s = SS(a, i, n)}
  while(i<n){
    s += a[i];
    i++;
  }//fin while
  return s;
  {Post ≡ a = a0 ∧ n = n0 ∧ somme_tableau = S(a, n)}
}//fin somme_tableau()
```

Inversion

- Soit `a[0, ..., n-1]`, un tableau initialisé à `n` valeurs entières et uniques
- Ecrire une fonction qui retourne le nombre d'inversions dans `a`
- Valeurs uniques?
  - $unique(a, n) \equiv$
    - ✓ $\forall\, j,\ 0 \leq j \leq n-1,\ \nexists\, i,\ 0 \leq i \leq n-1$
    - ✓ $\wedge\, i \neq j,\ a[j] = a[i]$
- (`i`, `j`) est une *inversion* de `a` ssi, par définition,
  - $0 \leq i \leq n\text{-}1 \wedge 0 \leq j \leq n\text{-}1$
  - $i < j$
  - $a[i] > a[j]$

Inversion (2)

- Exemple (`n`=4)

	0	1	2	3
a:	1	3	0	2

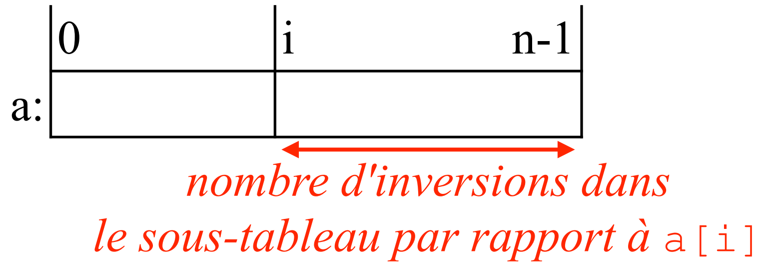
(1, 3) est une inversion car
1 < 3
`a`[1] > `a`[3]

(1, 2) est une inversion car
1 < 2
`a`[1] > `a`[2]

(0, 2) est une inversion car
0 < 2
`a`[0] > `a`[2]

Inversion (3)

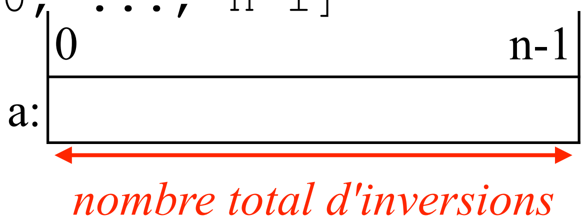
- Formalisation du problème
  - *inversion(a, i, n)* décrit le nombre d'inversions dans le sous-tableau `a[i, ..., n-1]` par rapport à `a[i]`



$$inversion(a, i, n) \equiv 0 \leq i \leq n-1, \#j \cdot (i+1 \leq j \leq n-1 \,|\, a[i] > a[j])$$

Inversion (4)

- Formalisation du problème (cont.)
  - *tot_inversion(a, n)* décrit le nombre total d'inversions dans `a[0, ..., n-1]`



$$tot\_inversion(a, n) \equiv \#i \cdot \left[0 \leq i \leq n-1 \;|\; \left(\#j \cdot (i+1 \leq j \leq n-1 \;|\; a[i] > a[j])\right)\right]$$
$$\equiv \sum_{i=0}^{n-1} inversion(a, i, n))$$

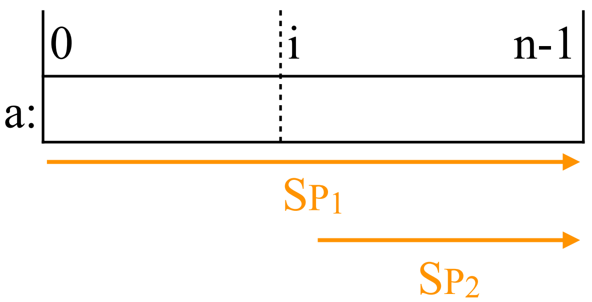
- Formalisation du problème (cont.)
  - *tot_inversion_stab(a, i, n)* décrit le nombre total d'inversions dans un sous-tableau `a[0…i-1]`
  - $tot\_inversion\_stab(a, i, n) \equiv \sum_{j=0}^{i-1} inversion(a, j, n)$

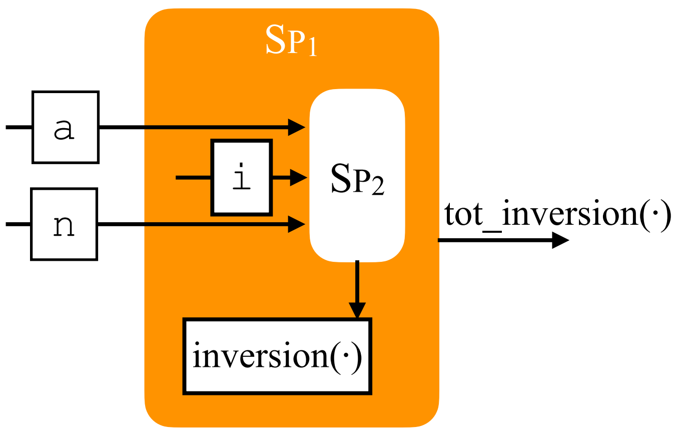
Inversion (5)

- Définition du problème
  - Input
    - ✓ `a`, tableau d'entiers à valeurs uniques
    - ✓ `n`, la taille de `a`
  - Output
    - ✓ le nombre d'inversion dans `a[0, ..., n-1]`, *tot_inversion(a, n)*
  - Caractérisation des Inputs
    - ✓ `a`, tableau d'entiers
      - `int *a;`
    - ✓ `n`, valeur entière
      - `int n;`

Inversion (6)

- Analyse du problème
- On peut envisager 2 SPs
  - SP$_1$: calcul de *tot_inversion(a, n)*
  - SP$_2$: calcul de *inversion(a, i, n)*
- Enchaînement
  - SP$_2$ ⊂ SP$_1$





Inversion (7)

- Spécification pour le SP_1

```
/*
 * @pre: a initialisé ∧ n ≥ 0 ∧ unique(a, n)
 * @post: a = a0 ∧ n = n0
 *        ∧ nb_inversion = tot_inversion(a, n)
 */
int nb_inversion(int *a, int n);
```

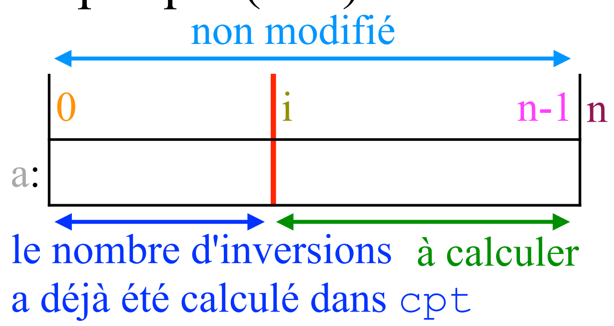

Inversion (8)

- Spécification pour le SP_2

```
/*
 * @pre: a initialisé ∧ n ≥ 0 ∧ unique(a, n)
 *       ∧ 0 ≤ i ≤ n-1
 * @post: a = a0 ∧ n = n0
 *        ∧ inversion_ss_tab = inversion(a, i, n)
 */
int inversion_ss_tab(int *a, int i, int n);
```

Inversion (9)

- Construction du SP_1
- Invariant Graphique (GLI)



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Invariant Formel (FLI)

$a = a_0 \wedge n = n_0$

$\wedge$

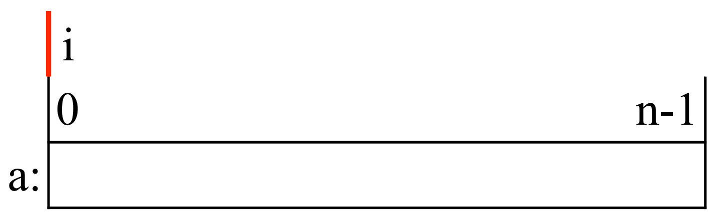
$0 \leq i \leq n$

$\wedge$

$cpt = tot\_inversion\_stab(a, i, n)$

Inversion (10)

- {Pré} `INIT` {Inv}
  - situation à établir



`cpt` = tot_inversion_stab(a, 0, n) = 0

Inversion (11)

- {Pré} `INIT` {Inv}
  - instructions

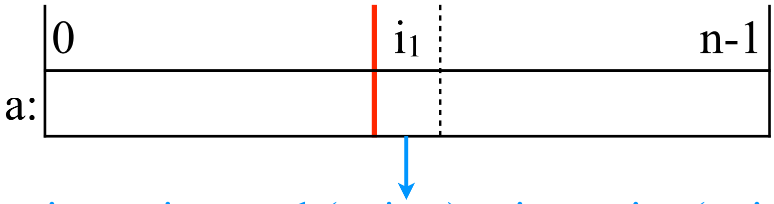
```
{Pré ≡ a initialisé ∧ n ≥ 0 ∧ unique(a, n)}
int i = 0;
{a = a0 ∧ n = n0 ∧ i=0}
int cpt = 0;
{a = a0 ∧ n = n0 ∧ i=0 ∧ cpt = 0}
⇒ {a = a0 ∧ n = n0 ∧ i=0 ∧ cpt=tot_inversion_stab(a,0,n)}
⇒ {a = a0 ∧ n = n0 ∧ i=0 ∧ cpt=tot_inversion_stab(a,i,n)}
⇒ {Inv ≡ a = a0 ∧ n = n0 ∧ cpt=tot_inversion_stab(a,i,n)
    ∧ 0 ≤ i ≤ n}
```


Inversion (12)

- Critère d'Arrêt (¬ B)
  - `i == n`
- {Inv ∧ ¬ B} `END` {Post}
  - rien à faire hormis retourner `cpt`
  - établir {Inv ∧ ¬ B} ⇒ {Post}
    - ✓ $a = a_0 \wedge n = n_0 \wedge 0 \leq i \leq n \wedge cpt = tot\_inversion\_stab(a, i, n) \wedge i = n$
    - ✓ $a = a_0 \wedge n = n_0 \wedge cpt = tot\_inversion\_stab(a, i, n) \wedge i = n$
    - ✓ $a = a_0 \wedge n = n_0 \wedge cpt = tot\_inversion(a, n)$
    - ✓ $a = a_0 \wedge n = n_0 \wedge nb\_inversion = tot\_inversion(a, n)$
      - {Post}

Inversion (13)

- {Inv ∧ B} `ITER` {Inv}



tot_inversion_stab(a, i_0,n) + inversion(a, i_1, n)

Il faut établir

`cpt = cpt + inversion(a, i₁, n)`

$\text{S}_{\text{P}2}$!!!

faire progresser `i` (`i = i₁+1`)

Inversion (14)

- {Inv ∧ B} `ITER` {Inv}
  - instructions

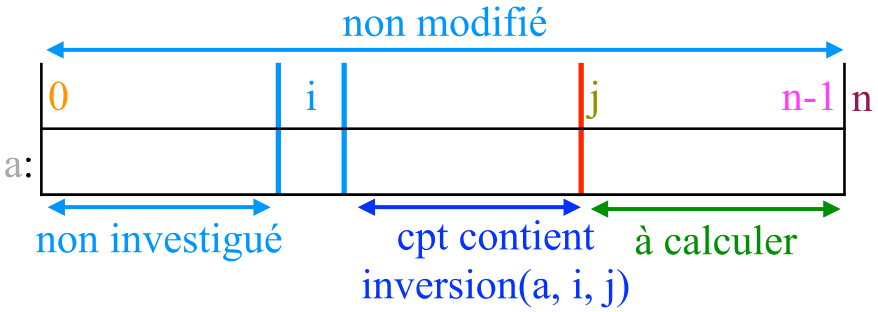
```
{Inv ≡ a=a0 ∧ n=n0 ∧ cpt=tot_inversion_stab(a,i,n) ∧ 0≤i≤n}
while(i<n){
  {Inv ∧ B ≡ a=a0 ∧ n=n0 ∧ cpt=tot_inversion_stab(a,i,n)
             ∧ 0 ≤ i ≤ n-1}
  {Inv ∧ B ⇒ PréSP2}     Sp2
  cpt += inversion_ss_tab(a, i, n);
                                            PostSp2
  {a = a0 ∧ n = n0 ∧ 0 ≤ i ≤n-1
  ∧ cpt=tot_inversion_stab(a,i,n) + inversion(a, i, n)}

  {a=a0 ∧ n=n0 ∧ 0≤i≤n-1 ∧ cpt=tot_inversion_stab(a,i+1,n)}
  i++;
  {Inv ≡ a = a0 ∧ n = n0 ∧ cpt = tot_inversion_stab(a,i,n)
         ∧ 0 ≤ i ≤ n}
}//fin while
```

Inversion (15)

- Fonction de Terminaison
  - *n-i*

Inversion (16)

- Construction SP2
- Invariant Graphique (GLI)



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Invariant Formel (FLI)

$a = a_0 \wedge n = n_0$

$\wedge$

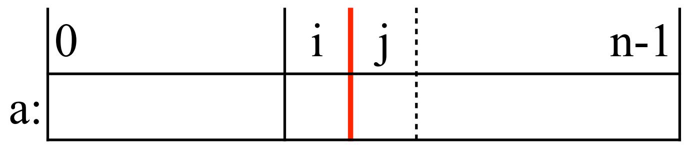
$0 \leq i < j \leq n$

$\wedge$

$cpt = inversion(a, i, j)$

Inversion (17)

- {Pré} `INIT` {Inv}
  - situation à établir



`cpt` = inversion(a, i, j)

= 0

Inversion (18)

- {Pré} `INIT` {Inv}
  - instructions

```
{Pré ≡ a init ∧ n ≥ 0 ∧ unique(a, n) ∧ 0 ≤ i ≤ n-1}
int j = i+1;
{a = a0 ∧ n = n0 ∧ 0 ≤ i ≤ n-1 ∧ j=i+1}
⇒ {a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n}
int cpt = 0;
{a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n ∧ cpt = 0}
⇒ {Inv ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n
         ∧ cpt = inversion(a, i, j)}
```

Inversion (19)

- Critère d'Arrêt ($\neg$ B)
  - `j == n`
- {Inv $\wedge \neg$ B} `END` {Post}
  - rien à faire hormis continuer SP1
  - établir {Inv $\wedge \neg$ B} $\Rightarrow$ {Post}
    - ✓ $a = a_0 \wedge n = n_0 \wedge 0 \leq i < j \leq n \wedge cpt = inversion(a, i, j) \wedge j = n$
    - ✓ $a = a_0 \wedge n = n_0 \wedge 0 \leq i \leq n\text{-}1 \quad cpt = inversion(a, i, n)$

Inversion (20)

- {Inv $\wedge$ B} `ITER` {Inv}

$inversion(a, i, j_0) + inversion(a, i, j_1)$

Il faut établir

`cpt = cpt + inversion(a, i, j`$_1$`)`

faire progresser `j` (`j = j`$_1$`+1`)

Inversion (21)

```
{Inv ≡ a = a0 ∧ n = n0 ∧ 0≤i<j≤n  ∧ cpt=inversion(a, i, j)}
while(j<n){
  {Inv ∧ B ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n-1
          ∧ cpt = inversion(a, i, j)}
  if(a[i] > a[j])
    {a = a0 ∧ n = n0 ∧ 0≤i<j≤n-1 ∧ cpt = inversion(a, i, j)
     ∧ a[i] > a[j]}
    cpt++;
    {a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n-1
     ∧ cpt = inversion(a, i, j) + 1 ∧ a[i] > a[j]}
  {a = a0 ∧ n = n0 ∧ 0≤i<j≤n-1 ∧ cpt = inversion(a, i, j+1)}
  j++;
  {Inv ≡ a = a0 ∧ n = n0 ∧ 0 ≤ i < j ≤ n
         ∧ cpt = inversion(a, i, j)}
}//fin while
```


Inversion (22)

- Fonction de Terminaison
  - *n-j*

Code Complet (23)

- Code complet

```
int inversion_ss_tab(int *a, int i, int n){
  int j = i+1;
  int cpt = 0;

  while(j<n){
    if(a[i] > a[j])
      cpt++;

    j++;
  }//fin while - j

  return cpt;
}//fin inversion_ss_tab()
```


Inversion (24)

- Code complet (cont.)

```
int nb_inversion(int *a, int n){
  int i=0;
  int cpt = 0;

  while(i<n){
    cpt += inversion_ss_tab(a, i, n);

    i++;
  }//fin while - i

  return cpt;
}//fin nb_inversion()
```

Plateaux

- Soit `a[0, ..., n-1]`, un tableau initialisé à `n` valeurs entières
- On désire connaître
  - le nombre de plateaux
  - la longueur du plus long plateau
- Plateau?
  - sous-tableau `a[i, ..., j]` tel que tous les éléments du sous-tableau sont identiques

Plateaux (2)

- Exemple

a:	1	1	3	2	2	2	6	6	4	1	1
	0	1	2	3	4	5	6	7	8	9	10

Plateaux (3)

- Formalisation du problème
- `a[i, ..., j]` est un **plateau** de `a` ssi, par définition,

  - $0 \leq i \leq j \leq n-1$
  - $a[i] = a[i+1] = \ldots = a[j]$
  - $i = 0 \vee a[i-1] \neq a[i]$
  - $j = n-1 \vee a[j] \neq a[j+1]$

Plateaux (4)

- Formalisation du problème (cont.)

$$
\begin{aligned}
plateau(a, i, j, n) \equiv\ & 0 \leq i \leq j \leq n-1 \\
& \wedge \\
& \forall k, i \leq k \leq j-1, a[k] = a[k+1] \\
& \wedge \\
& (i = 0 \vee a[i-1] \neq a[i]) \\
& \wedge \\
& (j = n-1 \vee a[j] \neq a[j+1]
\end{aligned}
$$

- $lg(a, i, j, n) \equiv \begin{cases} (j-i)+1 & \text{si } plateau(a, i, j, n) \\ 0 & \text{sinon.} \end{cases}$

Plateaux (5)

- Formalisation du problème (cont.)
- Exemple pour *lg(...)*
  - lg(plateau(a, 0, 1, 11)) = 2
  - lg(plateau(a, 3, 5, 11)) = 3
  - lg(plateau(a, 8, 10, 11)) = 0

	0									10
a:	1	1	3	2	2	2	6	6	4	1	1

Plateaux (6)

- Formalisation du problème (cont.)
- $Np(a, n) \equiv \#(i, j) \cdot (i, j \in 0, \ldots, n-1 \mid plateau(a, i, j, n))$
  - nombre de plateaux de `a`
  - $Np(a, 11) = 6$
- $Mlp(a, n) \equiv max_{(i, j) \in 0, \ldots, n-1} (lg(plateau(a, i, j, n)))$
  - maximum des longueurs des plateaux de `a`
  - $Mlp(a, 11) = 3$

	0									10
a:	1	1	3	2	2	2	6	6	4	1	1

Plateaux (7)

- Formalisation du problème (cont.)
- *Np(a, i, n)*
  - nombre de plateaux dans le sous-tableau `a[0, ..., i]`
  - *Np(a, 5, 11)* = 3
- *Mlp(a, i, n)*
  - maximum des longueurs des plateaux dans le sous-tableau `a[0, ..., i]`
  - *Mlp(a, 5, 11)* = 3

	0									10
a:	1	1	3	2	2	2	6	6	4	1	1

Plateaux (8)

- Problème des plateaux
  - trouver *Np(a, n)*
  - trouver *Mlp(a, n)*
- Contrainte
  - la solution doit être linéaire par rapport à la taille du tableau
  - *O(n)*

Plateaux (9)

- Pour résoudre ce problème, on va utiliser les éléments suivants

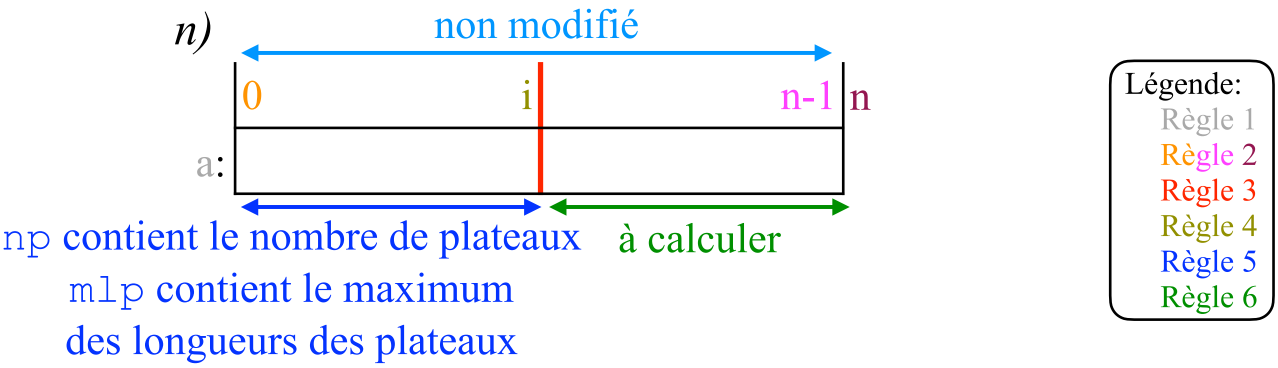
```
#define n ...
int a[n];
int np, mlp;
```

- Spécification

```
/*
 * @pre: a initialisé
 * @post: a = a₀ ∧ n = n₀ ∧ np = Np(a, n) ∧ mlp = Mlp(a, n)
 */
```


Plateaux (10)

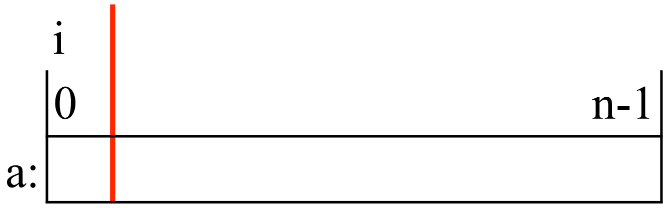
- Invariant Graphique (GLI)
  - parcourir `a` de gauche à droite
  - maintenir la condition `np` = *Np(a, i, n)* ∧ `mlp` = *Mlp(a, i, n)*



- Invariant Formel (FLI)
  - $a = a_0 \wedge n = n_0 \wedge 0 \leq i \leq n-1 \wedge np = Np(a, i, n) \wedge mlp = Mlp(a, i, n)$

Plateaux (11)

- {Pré} `INIT` {Inv}
  - la situation à établir est la suivante



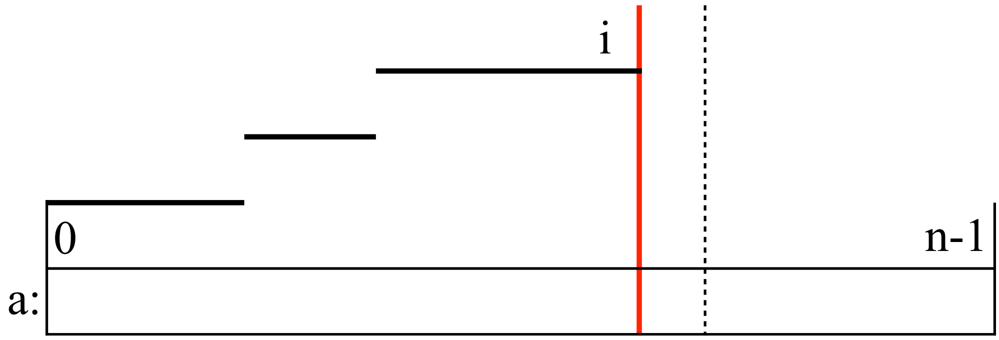
Plateaux (12)

- {Pré} `INIT` {Inv}
  - instructions

```
{Pré ≡ a initialisé}
int i = 0;
{a=a_0 ∧ i=0 ∧ n = n_0}
np = 1;
{a=a_0 ∧ i=0 ∧ np=1 ∧ n = n_0}
⇒ {a=a_0 ∧ i=0 ∧ np = Np(a, 0, n) ∧ n = n_0}
⇒ {a=a_0 ∧ i=0 ∧ np = Np(a, i, n) ∧ n = n_0}
mlp = 1;
{a=a_0 ∧ i=0 ∧ np = Np(a, i, n) ∧ mlp = 1 ∧ n = n_0}
⇒ {a=a_0 ∧ i=0 ∧ np = Np(a, i, n) ∧ mlp = Mlp(a, 0, n)
   ∧ n = n_0}
⇒ {a=a_0 ∧ i=0 ∧ np = Np(a, i, n) ∧ mlp = Mlp(a, i, n)
   ∧ n = n_0}
⇒ {Inv ≡ a=a_0 ∧ 0 ≤ i ≤ n-1 ∧ np = Np(a, i, n)
         ∧ mlp = Mlp(a, i, n) ∧ n = n_0}
```

Plateaux (13)

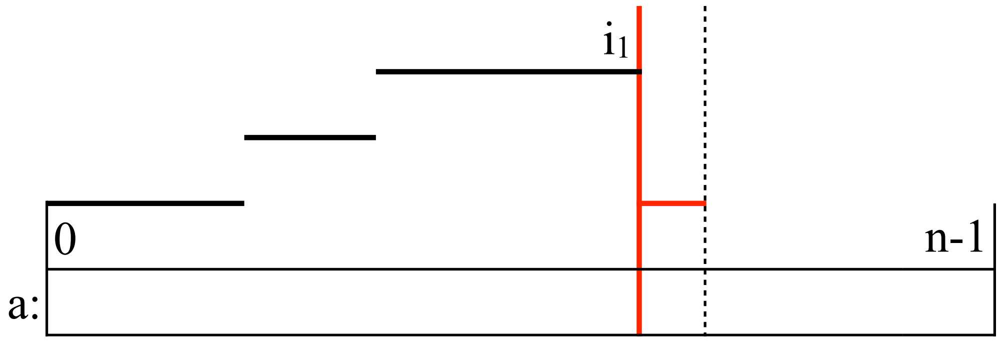
- Critère d'Arrêt (`¬B`)
  - cfr. condition de fin de plateau
  - `i == n-1`
- {Inv ∧ B} `ITER` {Inv}



$i = i_1 \leadsto 0 \leq i_1 < n-1$
$np = Np(a, i_1, n)$
$mlp = Mlp(a, i_1, n)$

Plateaux (14)

- Il faut envisager 2 cas
  1. $a[i_1] \neq a[i_1+1]$

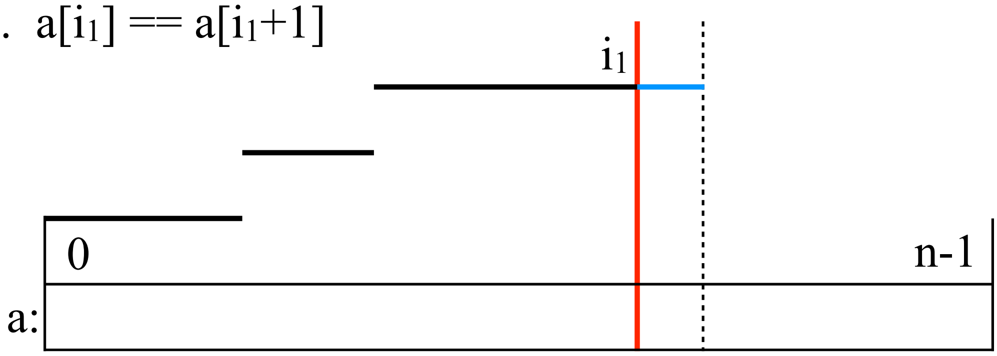


Il faut établir:

$i = i_1 + 1$
un plateau en plus $\Rightarrow np = Np(a, i_1, n) + 1$
pas le plus long plateau $\Rightarrow mlp = Mlp(a, i_1, n)$

Plateaux (15)

2. a[i_1] == a[i_1+1]



Il faut établir:

$i = i_1 + 1$

pas de plateau en plus $\Rightarrow$ np = Np(a, i_1, n)

le plateau en cours est peut-être le plus long

On en sait rien!

Information non contenue dans l'Invariant

$\Rightarrow$ Notre Invariant est incomplet

Plateaux (16)

- Il faut modifier l'Invariant!
- Soit *Ldp(a, i, n)*, avec $0 \leq i \leq n-1$
  - la longueur du dernier plateau de `a[0 ... i]`
- L'Invariant devient
  - $a = a_0 \wedge 0 \leq i \leq n-1 \wedge np = Np(a, i, n) \wedge mlp = Mlp(a, i, n) \wedge ldp = Ldp(a, i, n) \wedge n = n_0$
- Il faut modifier `INIT`

```
{Pré ≡ a initialisé}
int i = 0;
np = 1;
mlp = 1;
ldp = 1;
{Inv ≡ a=a0 ∧ 0 ≤ i ≤ N-1 ∧ np = Np(a, i, n)
      ∧ mlp = Mlp(a, i, n) ∧ ldp = Ldp(a, i, n) ∧ n = n0}
```

Plateaux (17)

- Le raisonnement pour {Inv ∧ B} `ITER` {Inv} était bon
  - il faut juste l'adapter
- Deux cas (en plus de ce qui a été dit):
  1. $a[i_1] \neq a[i_1+1]$
     - ✓ réinitialiser ldp
       - ldp = 1;
  2. $a[i_1] == a[i_1+1]$
     - ✓ mettre à jour ldp
       - ldp = Ldp(a, i_1) + 1
     - ✓ vérifier la longueur du plus long plateau
       - mlp = max(Mlp(a, i_1), ldp)

Plateaux (18)

```
{Inv ≡ a=a0 ∧ 0 ≤ i ≤ N-1 ∧ np = Np(a, i,nN) ∧ n = n0
       ∧ mlp = Mlp(a, i, n)  ∧ ldp = Ldp(a, i, n)}
while(i!=n-1){
  if(a[i]!=a[i+1]){
    np++;
    ldp = 1;
  }else{
    ldp++;
    if(mlp < ldp)
      mlp = ldp;
  }
  i++;
}//fin while
```

Plateaux (19)

- A la sortie de la boucle, il n'y a plus rien à faire
  - vérifier que $\{Inv \wedge \neg B\} \Rightarrow \{Post\}$
    - ✓ $0 \leq i \leq n\text{-}1 \wedge np = Np(a, i, n) \wedge mlp = Mlp(a, i, n) \wedge ldp = Ldp(a, i, n) \wedge i = n\text{-}1 \wedge a = a_0 \wedge n = n_0$
      $\Rightarrow np = Np(a, n\text{-}1, n) \wedge mlp = Mlp(a, n\text{-}1, n) \wedge ldp = Ldp(a, n\text{-}1, n) \wedge a = a_0 \wedge n = n_0$
      $\Rightarrow np = Np(a, n) \wedge mlp = Mlp(a, n) \wedge a = a_0 \wedge n = n_0$
- Fonction de Terminaison
  - *n-1-i*

Plateaux (20)

- Code complet

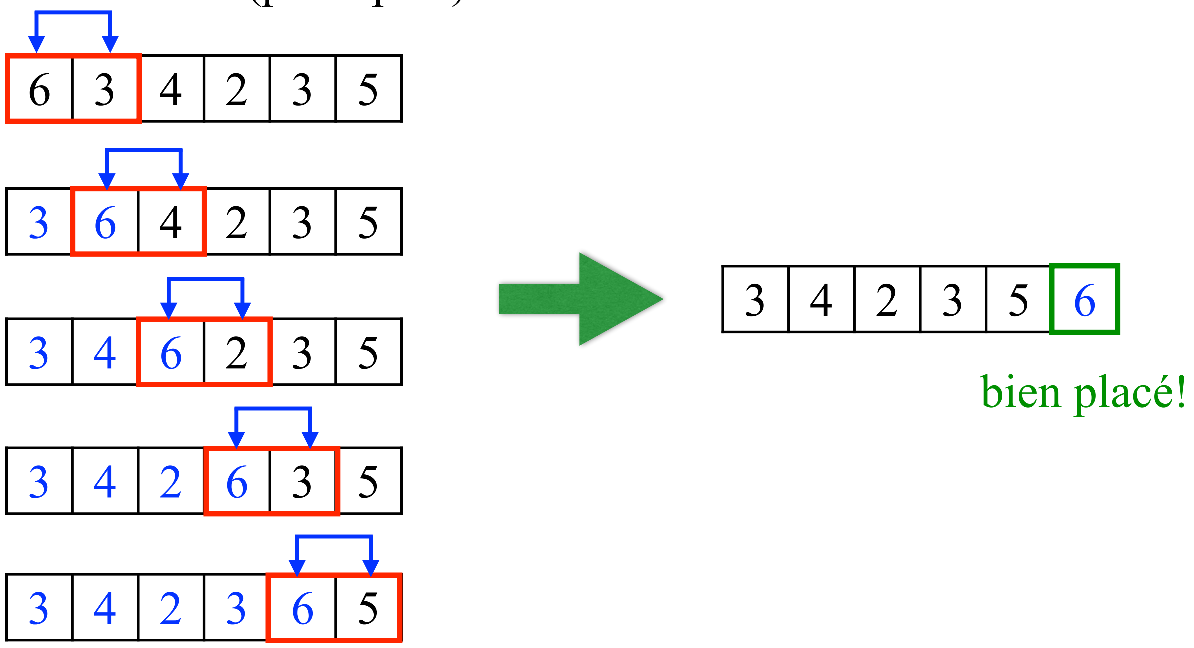
```
{Pré ≡ a initialisé}
int i = 0;
np = 1; mlp = 1; ldp = 1;
{Inv ≡ 0 ≤ i ≤ n-1 ∧ np = Np(a, i, n) ∧ mlp = Mlp(a, i, n)
       ∧ ldp = Ldp(a, i, n) ∧ a = a0 ∧ n = n0}
while(i!=n-1){
  if(a[i]!=a[i+1]){
    np++;
    ldp = 1;
  }else{
    ldp++;
    if(mlp < ldp)
      mlp = ldp;
  }
  i++;
}//fin while
{Post ≡ a = a0 ∧ np = Np(a, n) ∧ mlp = Mlp(a, n)}
```

Tri par Bulles

- Le tri par bulles
  - *Bubble Sort*
- Idée?
  - trier les éléments du tableau 2 par 2
- Objectif?
  - faire "remonter" le maximum en fin de tableau, comme une bulle, de manière itérative

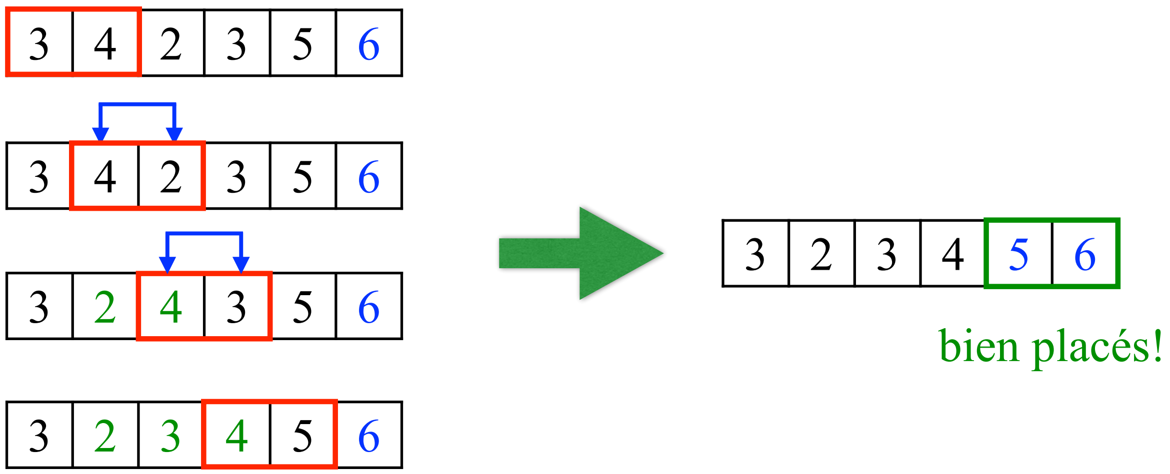
Tri par Bulles (2)

- Illustration du principe
  - itération (principale) 1



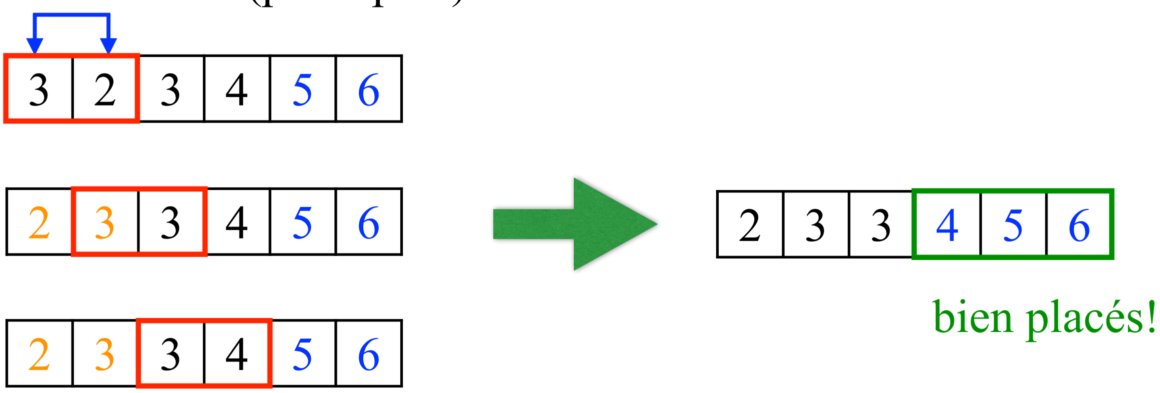
Tri par Bulles (3)

- Illustration du principe
  - itération (principale) 2



Tri par Bulles (4)

- Illustration du principe
  - itération (principale) 2



Tri par Bulles (5)

- Formalisation du problème
- Un tableau `a` à `n` valeurs entières est <u>trié par ordre croissant</u>
  - $trie(a,n) \equiv \forall\ i,\ 0 \leq i < n-1,\ a[i] \leq a[i+1]$
- Le sous-tableau `a[i, ..., j]` d'un tableau `a` à `n` valeurs entières est <u>trié par ordre croissant</u>
  - $trie\_stab(a,i,j,n) \equiv 0 \leq i \leq j \leq n-1 \wedge \forall\ k, i \leq k < j, a[k] \leq a[k+1]$
- Par définition
  - *trie(a, n)* est identique à *trie_stab(a, 0, n-1, n)*

Tri par Bulles (6)

- Le tri implique que le contenu du tableau n'est pas modifié
  - ce sont les mêmes valeurs
  - mais dans un ordre différent
- Comment exprimer cela?

Tri par Bulles (7)

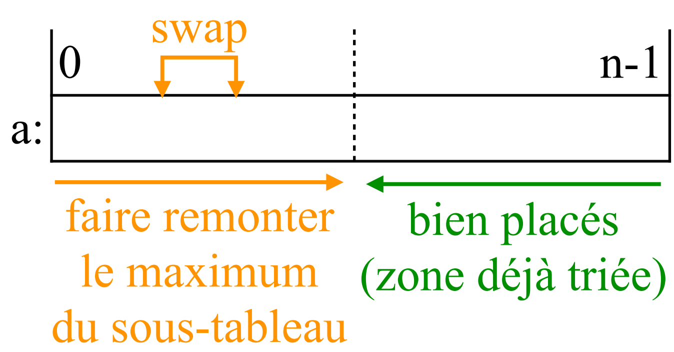
- Notion de **permutation**
- Soient deux suites *L* et *L'*
- *L* est une *permutation* de *L'* (notation *L perm L'*) ssi $\exists$ $i_1, i_2, ..., i_n$ tels que
  - $\{i_1, i_2, ..., i_n\} = \{1, 2, ..., n\}$
  - $L = (l_1, l_2, ..., l_n)$
  - $L' = (l_{i1}, l_{i2}, ..., l_{in})$

Tri par Bulles (8)

- Définition du problème
  - Input
    - `a`, tableau à valeurs entières
    - `n`, la taille du tableau `a`
  - Output
    - le tableau `a` est trié par ordre croissant, *trie(a, n)*
  - Caractérisation des Inputs
    - `a` est un tableau d'entiers
      - `int *a;`
    - `n` est une valeur entière
      - `int n;`

Tri par Bulles (9)

- Analyse du problème
  - SP$_1$: maintenir la zone déjà triée
  - SP$_2$: faire remonter le maximum du sous-tableau
  - SP$_3$: swap entre 2 valeurs du tableau
- Enchaînement
  - (SP$_3$ $\subset$ SP$_2$) $\subset$ SP$_1$



Tri par Bulles (10)

- Spécification du problème général

```
/*
 * @pre: a initialisé ∧ n ≥ 0
 * @post: n = n0  ∧ a perm a0 ∧ trie(a, n)
 */
void bubble_sort(int *a, int n);
```

Tri par Bulles (11)

- Construction SP3 (swap)
- Spécification

```
/*
 * @pre: a initialisé ∧ n > 0 ∧ 0 ≤ i, j ≤ n-1
 * @post: n = n0 ∧ i = i0 ∧ j = j0 ∧ a perm a0
 *        ∧ a[i] = a0[j] ∧ a[j] = a0[i]
 */
void swap(int *a, int n, int i, int j);
```

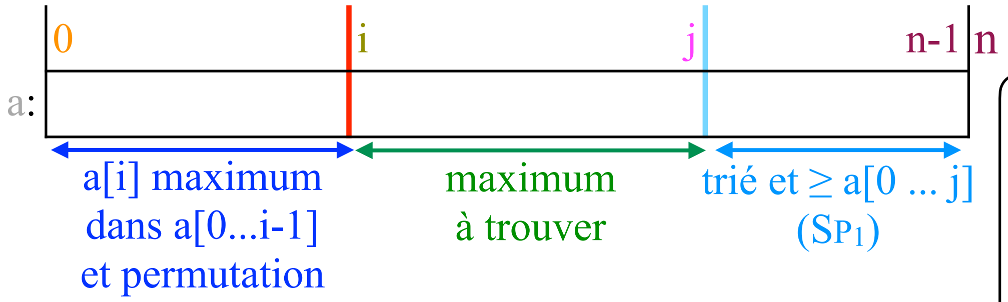

Tri par Bulles (12)

- Code SP3

```
void swap(int *a, int n, int i, int j){
  {Pré ≡ a initialisé ∧ n > 0 ∧ 0 ≤ i, j ≤ n-1}
  int tmp = a[i];
  {a = a0 ∧ n = n0 ∧ i = i0 ∧ j = j0 ∧ tmp = a[i]}
  a[i] = a[j];
  {n = n0 ∧ i = i0 ∧ j = j0 ∧ a[i] = a0[j]
   ∧ tmp = a0[i]}
  a[j] = tmp;
  {n = n0 ∧ i = i0 ∧ j = j0 ∧ a perm a0 ∧ a[i] = a0[j]
   ∧ tmp = a0[i] ∧ a[j] = tmp}
  {Post ≡ n = n0 ∧ i = i0 ∧ j = j0 ∧ a perm a0
         ∧ a[i] = a0[j] ∧ a[j] = a0[i]}
}//fin swap()
```

Tri par Bulles (13)

- Construction SP_2 (faire remonter le maximum)
- Invariant Graphique (GLI)



Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Invariant Formel (FLI)

$n = n_0$

$\wedge$ trie_stab(a, j+1, n-1, n) $\wedge\ \forall k, j+1 \leq k \leq n-1,$

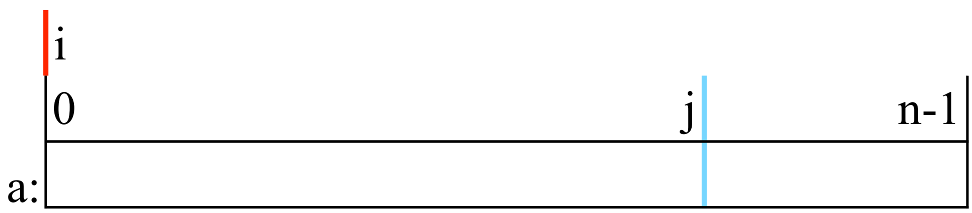
$\forall p, 0 \leq p \leq j, a[k] \geq a[p]$

$\wedge\ 0 \leq i \leq j \leq n-1$

$\wedge\ a[i] = \max_{0 \leq k \leq i-1} a[k] \wedge a \text{ perm } a_0$

Tri par Bulles (14)

- {Pré} `INIT` {Inv}
  - situation à établir



- {Pré} `INIT` {Inv}
  - code

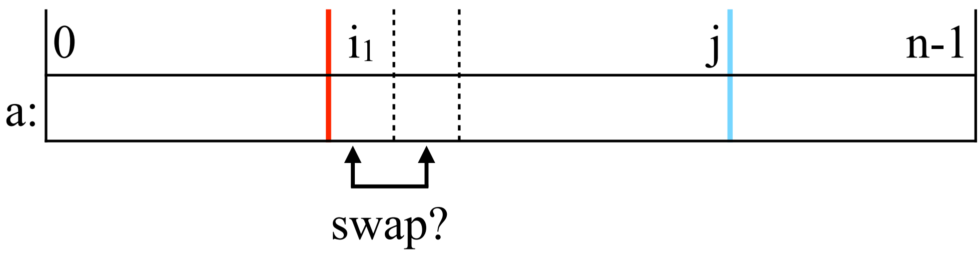
```
{n = n0 ∧ 0 ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n) ∧
∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p] ∧ a perm a0}
int i = 0;
{Inv}
```

Tri par Bulles (15)

- Critère d'Arrêt ($\neg$ B)
  - `i==j`
- {Inv $\wedge$ $\neg$ B} `END` {Post}
  - rien à faire, hormis continuer SP_1
  - on est dans la situation suivante:
    - ✓ $n = n_0 \wedge 0 \leq j \leq n-1 \wedge$ trie_stab(a, $j+1$, n-1, n) $\wedge$
      $\forall k, j+1 \leq k \leq n-1, \forall p, 0 \leq p \leq j, a[k] \geq a[p] \wedge$
      $a[j] = \max_{0 \leq k \leq j-1} (a[k]) \wedge$ a perm a_0
    - ✓ $n = n_0 \wedge 0 \leq j \leq n-1 \wedge$ trie_stab(a, j, n-1, n) $\wedge$
      $\forall k, j \leq k \leq n-1, \forall p, 0 \leq p \leq j-1, a[k] \geq a[p] \wedge$ a perm a_0

Tri par Bulles (16)

- {Inv $\wedge$ B} `ITER` {Inv}



Il faut
  faire éventuellement un swap entre `a[i₁]` et `a[i₁+1]`
  faire progresser `i` (`i = i₁+1`)

Tri par Bulles (17)

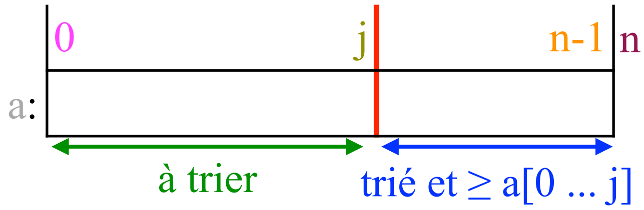
```
{Inv ≡ n = n0 ∧ 0 ≤ i ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
       ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
       ∧ a[i] = max0 ≤ k ≤ i-1 a[k] ∧ a perm a0}
while(i<j){
  {Inv ∧ B ≡ n = n0 ∧ 0 ≤ i < j ≤ n-1 ∧ trie_stab(a, j+1, n-1, 
n)
             ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
             ∧ a[i] = max0 ≤ k ≤ i-1 a[k] ∧ a perm a0}
  if(a[i] > a[i+1])
    {n = n0 ∧ 0 ≤ i < j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
      ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
      ∧ a[i] = max0 ≤ k ≤ i-1 a[k] ∧ a perm a0 ∧ a[i] > a[i+1]}
     swap(a, i, i+1, n);
    {n = n0 ∧ 0 ≤ i < j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
      ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
      ∧ a[i+1] = max0 ≤ k ≤ i a[k] ∧ a perm a0}
  i++;
  {Inv ≡ n = n0 ∧ 0 ≤ i ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
         ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
         ∧ a[i] = max0 ≤ k ≤ i-1 a[k] ∧ a perm a0}
}//fin while
```


Tri par Bulles (18)

- Fonction de Terminaison
  - $j - i$

Tri par Bulles (19)

- Construction SP1 (maintenir la zone triée)
- Invariant Graphique (GLI)



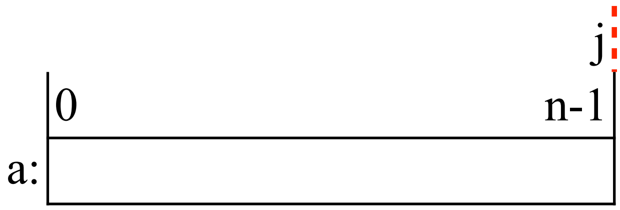
Légende:
Règle 1
Règle 2
Règle 3
Règle 4
Règle 5
Règle 6

- Invariant Formel (FLI)

$n = n_0$
$\wedge$
$-1 \leq j \leq n-1$
$\wedge$
trie_stab(a, j+1, n-1, n)
$\wedge\ \forall\ k,\ j+1 \leq k \leq n-1,\ \forall\ p,\ 0 \leq p \leq j,\ a[k] \geq a[p]$
$\wedge$ a perm a_0

Tri par Bulles (20)

- {Pré} INIT {Inv}
  - situation à établir



- {Pré} INIT {Inv}
  - code

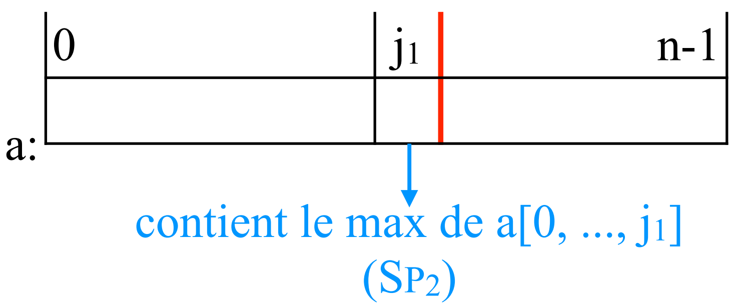
```
{Pré ≡ a initialisé ∧ n > 0}
int j = n-1;
{Inv ≡ n = n0 ∧ -1 ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
       ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
       ∧ a perm a0}
```

Tri par Bulles (21)

- Critère d'Arrêt ($\neg$ B)
  - `j==-1`
- {Inv $\wedge$ $\neg$ B} `END` {Post}
  - rien à faire, le tableau devrait être trié
  - on est dans la situation suivante:
    - ✓ $n = n_0 \wedge -1 \leq j \leq n-1 \wedge$ trie_stab(a, j+1, n-1, n) $\wedge$
      $\forall$ k, $j+1 \leq k \leq n-1$, $\forall$ p, $0 \leq p \leq j$, $a[k] \geq a[p] \wedge$ a perm $a_0 \wedge$
      $j == -1$
    - ✓ $n = n_0 \wedge$ trie_stab(a, 0, n-1, n) $\wedge$ a perm a_0
    - ✓ $n = n_0 \wedge$ trie_stab(a, n) $\wedge$ a perm a_0

Tri par Bulles (22)

- {Inv $\wedge$ B} `ITER` {Inv}



Il faut seulement faire progresser `j` (`j = j1-1`)

Tri par Bulles (23)

```
{Inv ≡ n = n0 ∧ -1 ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
       ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
       ∧ a perm a0}
while(j>=0){
  {Inv ∧ B ≡ n = n0 ∧ 0 ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
           ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
           ∧ a perm a0}

  //SP2

  {n = n0 ∧ 0 ≤ j ≤ n-1 ∧ trie_stab(a, j, n-1, n)
   ∧ ∀ k, j ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j-1, a[k] ≥ a[p]
   ∧ a perm a0}
  j--;
  {Inv ≡ n = n0 ∧ -1 ≤ j ≤ n-1 ∧ trie_stab(a, j+1, n-1, n)
        ∧ ∀ k, j+1 ≤ k ≤ n-1, ∀ p, 0 ≤ p ≤ j, a[k] ≥ a[p]
        ∧ a perm a0}
}//fin while
```


Tri par Bulles (24)

- Fonction de Terminaison
  - *j-1*

Tri par Bulles (25)

- Code complet (SP_1, SP_2 & SP_3)

```
void bubble_sort(int *a, int n){
  int j=n-1, i;

  while(j>=0){
    i = 0;

    while(i<j){
      if(a[i]>a[i+1])
        swap(a, i, i+1, n);

      i++;
    }//fin while - i

    j--;
  }//fin while - j
}//fin bubble_sort()
```