# Cours 6

M.Adam – JF.Kamp – S.Letellier – F.Pouit

14 août 2016

## Table des matières

# 1 Boucles imbriquées

## 1.1 Définition

Une boucle est dite imbriquée quand elle s'exécute dans une autre boucle.

L'imbrication pour être sur plusieurs niveaux ce qui rend la compréhension et l'écriture du code encore plus complexe.

### 1.1.1 Exemple

Voici une fonction `estInclusTab` qui détermine si un tableau d'entiers est inclus dans un autre.

```
##
# détermine si toutes les valeurs du premier tableau sont dans le second
# @param tableau des valeurs incluses
# @param tableau de valeurs
# @return vrai ssi toutes les valeurs du premier tableau sont dans le second
##
fonction estInclusTab (par_ref tableau_de entier tab1,
                       par_ref tableau_de entier tab2) : booleen
```

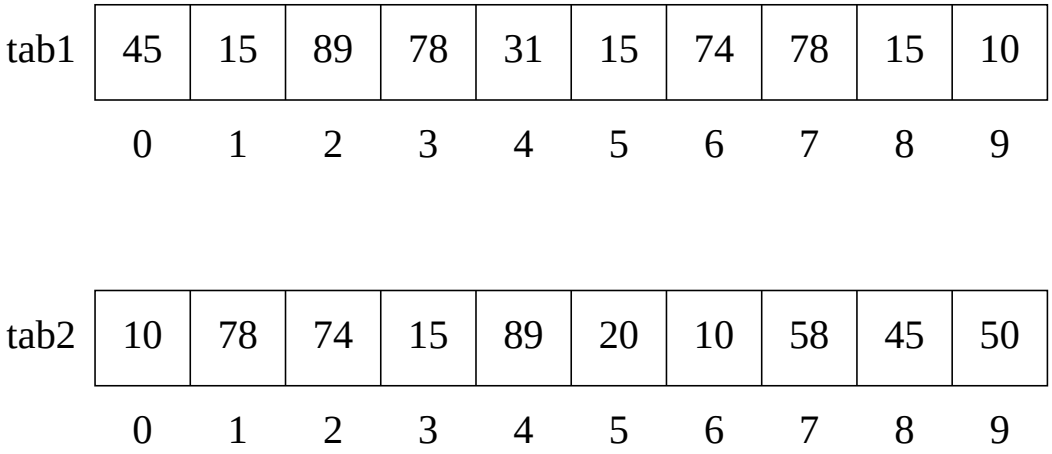
Par commodité, les deux tableaux ont la même taille `LG_TAB`.

**Inclus ou pas ?**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| tab1 | 45 | 15 | 89 | 78 | 50 | 15 | 74 | 78 | 15 | 10 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| tab2 | 10 | 78 | 74 | 15 | 89 | 20 | 10 | 58 | 45 | 50 |

**Inclus ou pas ?**




### 1.1.2 Le code complet

```
##
# détermine si toutes les valeurs du premier tableau sont dans le second
# @param tableau des valeurs incluses
# @param tableau de valeurs
# @return vrai ssi toutes les valeurs du premier tableau sont dans le second
##
fonction estInclusTab (par_ref tableau_de entier tab1,
                       par_ref tableau_de entier tab2) : booleen
var
    entier i;
    entier j;
    booleen present;
debut
    i := 0;
    present := vrai;
    tantque (i < LG_TAB et present)
        j:= 0;
        present := faux;
        tantque (j < LG_TAB et (non present))
            si (tab1[i] == tab2[j]) alors
                present := vrai;
            finsi
            j := j + 1;
       fintantque
       i := i + 1;
   fintantque
   retourne present;
fin
```

Cet algorithme servira d'exemple dans la suite de ce cours !

## 1.2 Méthode

Comme vu précédemment, un des grands principes de la programmation est de décomposer un gros problème en sous-problèmes, plus faciles à résoudre.

L'objectif est d'utiliser pour chaque boucle, intérieure et extérieure, la méthode vue lors du dernier cours.

Deux stratégies :

- commencer par la boucle externe,
- commencer par la boucle interne.

### 1.2.1 Principe générale de l'exemple

Objectif : savoir si le tableau `tab1` de 10, `LG_TAB`, entiers est inclus dans le tableau `tab2` de 10, `LG_TAB`, entiers.

Principe : pour chaque élément de `tab1`, vérifier s'il est présent dans `tab2`. S'il ne l'est pas c'est qu'il n'y a pas inclusion.

## 1.3 Boucle externe d'abord

### 1.3.1 Boucle externe

**Corps de la boucle externe**
Sont utilisées les variables suivantes :

- `i` pour parcourir `tab1`,
- `present` pour savoir si `tab[i]` est dans `tab2`.

Déclarations :

```
    entier i;
    booleen present;
```

```
##
# tab1[i] dans tab2 ?
# present vaut
# - vrai si tab1[i] est dans tab2,
# - faux sinon
##
i := i+1;
```

**Conditions de sortie de la boucle externe**

- `i >= LG_TAB` pour la sortie de tableau `tab1`
- `non present` car la valeur `tab[i]` n'est pas dans `tab2`.
- au total : `i >= LG_TAB ou (non present)`

**Condition de continuation de la boucle externe**

- `non (i >= LG_TAB ou (non present))`
- qui s'écrit aussi : `i < LG_TAB et present`

**Initialisation de la boucle externe**

```
i := 0;          #commencer au début du tableau
present := vrai; #par défaut tab1 dans tab2
```

**Terminaison de la boucle externe**

```
retourne present;
```

**Code de la boucle externe**

```
i := 0;          #commencer au début du tableau
present := vrai; #par défaut tab1 dans tab2
tantque (i < LG_TAB et present)
    ##
    # tab1[i] dans tab2 ?
    # present vaut vrai si tab1[i] est dans tab2, faux sinon
    ##
    i := i + 1;
fintantque
retourne present;
```

### 1.3.2 Boucle interne

La boucle interne a été définie par :

```
##
# tab1[i] dans tab2 ?
# présent vaut
# - vrai si tab1[i] est dans tab2,
# - faux sinon
##
```

**Principe de la boucle interne**
Parcourir le tableau `tab2` pour chercher si `tab1[i]` est présent.

Sortir de la boucle dès que `tab1[i]` est présent dans `tab2`.

**Corps de la boucle interne**

- `j` sert à parcourir `tab2`.

Déclarations :

```
    entier j;
```

Corps de boucle interne :

```
si (tab1[i] == tab2[j]) alors
    present := vrai;
finsi
j := j + 1;
```

**Conditions de sortie**

- `j <= LG_TAB` : tout le tableau a été parcouru,
- `present` : la valeur `tab1[i]` a été trouvée,
- au total : `j >= LG_TAB ou present`

**Condition de continuation de la boucle interne**

- La négation de la condition de sortie : `non(j >= LG_TAB ou present)`
- qui s'écrit aussi : `j < LG_TAB et (non present)`

**Initialisation de la boucle interne**

```
j := 0;          #commencer au début du tableau
present := faux; #la valeur tab[i] n'est pas trouvée
```

**Terminaison de la boucle interne**
La terminaison est vide !

**Code complet de la boucle interne**

```
j := 0;          # commencer au début du tableau
present := faux; # la valeur tab[i] n'est pas trouvée
tanque (j < LG_TAB et (non present))
    si (tab1[i] == tab2[j]) alors
        present := vrai;
    finsi
    j := j + 1;
fintantque
```

### 1.3.3 Le code complet des deux boucles

```
   i := 0;
   present := vrai;
   tantque (i < LG_TAB et present)
```

```
      j := 0;
      present := faux;
      tantque (j < LG_TAB et (non present))
          si (tab1[i] == tab2[j]) alors
              present := vrai;
          finsi
          j := j + 1;
      fintantque
      i := i + 1;
  fintantque
  retourne present;
```

## 1.4 Boucle interne d'abord

**Principe général :** Chercher si chaque élément `tab1[i]` de `tab1` est dans `tab2`.

Sortir de la boucle dès que `tab1[i]` est présent dans `tab2`.

### 1.4.1 Boucle interne

**Principe :** Parcourir le tableau `tab2` pour y chercher la présence de l'élément courant de `tab1`.

**Corps de la boucle interne**
Déclarations :

```
    entier i;
    entier j;
    booleen present;
```

Corps de boucle interne :

```
si (tab1[i] == tab2[j]) alors
    present := vrai;
finsi
j := j + 1;
```

**Conditions de sortie**

- `j >= LG_TAB` : tout le tableau a été parcouru,
- `present` : la valeur `tab1[i]` a été trouvée,
- au total : `j >= LG_TAB ou present`

**Condition de continuation de la boucle interne**

- La négation de la condition de sortie : `non(j >= LG_TAB ou present)`
- qui s'écrit aussi : `j < LG_TAB et (non present)`

#### Initialisation de la boucle interne

```
j := 0;          #commencer au début du tableau
present := faux; #la valeur tab[i] n'est pas trouvée
```

#### Terminaison de la boucle interne

La terminaison est vide !

#### Code complet de la boucle interne

```
j := 0;          # commencer au début du tableau
present := faux; # la valeur tab[i] n'est pas trouvée
tanque (j < LG_TAB et (non present))
    si (tab1[i] == tab2[j]) alors
        present := vrai;
    finsi
    j := j + 1;
fintantque
```

### 1.4.2 Boucle externe

#### Corps de la boucle externe

Sont utilisées les variables suivantes :

- `i` pour parcourir `tab1`,
- `present` pour savoir si `tab[i]` est dans `tab2`.

Toutes les déclarations ont déjà été faites pour la boucle interne.

```
j := 0;          #commencer au début du tableau
present := faux; #la valeur tab[i] n'est pas trouvée
tanque (j < LG_TAB et (non present))
    si (tab1[i] == tab2[j]) alors
        present := vrai;
    finsi
    j := j + 1;
fintantque
i := i + 1;
```

#### Conditions de sortie de la boucle externe

- `i >= LG_TAB` pour la sortie de tableau `tab1`
- `non present` car la valeur `tab[i]` n'est pas dans `tab2`.
- au total : `i >= LG_TAB ou (non present)`

#### Condition de continuation de la boucle externe

- `non (i >= LG_TAB ou (non present))`
- qui s'écrit aussi : `i < LG_TAB et present`

**Initialisation de la boucle externe**

```
i := 0;          #commencer au début du tableau
present := vrai; #par défaut tab1 dans tab2
```

**Terminaison de la boucle externe**

```
retourne present;
```

**Code de la boucle externe**

```
i := 0;              # commencer au début du tableau
present := vrai;     # par défaut tab1 dans tab2
tantque (i < LG_TAB et present)
    j := 0;          #commencer au début du tableau
    present := faux; #la valeur tab[i] n'est pas trouvée
    tanque (j < LG_TAB et (non present))
        si (tab1[i] == tab2[j]) alors
            present := vrai;
        finsi
        j := j + 1;
    fintantque
    i := i + 1;
fintantque
retourne present;
```

## 1.5 Procédure de Test

```
TestAlgo - Interprétation engagée.

*** testTestInclusTab
tab1 : 10 20 30 40 50 60 70 80 90 0
tab2 : 30 40 70 0 90 10 60 80 50 20
estInclusTab (@tab1, @tab2)VRAI

tab1 : 10 20 30 40 50 60 70 80 90 100
tab2 : 30 40 70 0 90 10 60 80 50 20
estInclusTab (@tab1, @tab2)FAUX

TestAlgo - Fin de l'interprétation.
```

## 1.6 Remarques

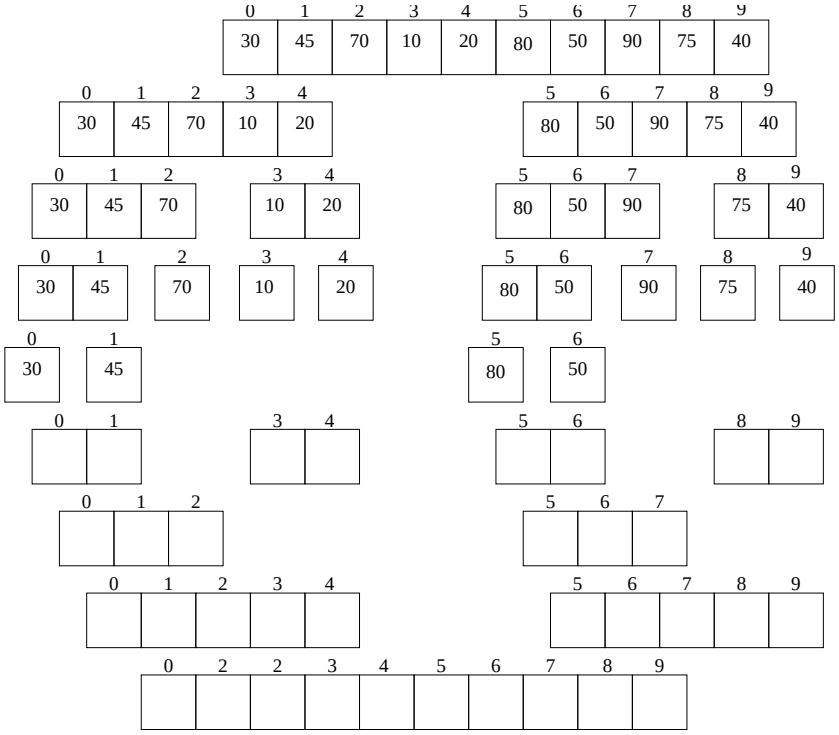
- Cette manière de procéder peut sembler longue et fastidieuse, mais elle est une meilleure garantie de faire un code correct et plus facile à corriger.
- Certains pourront préférer commencer par le code de la boucle interne.
- La boucle interne peut être remplacée par une fonction.

# 2 La récursivité avec le tri par fusion

## 2.1 Comparaison de deux tris

```
##
# compare le tri par fusion et par sélection
##
procedure comparaison()
var
    tableau_de entier tab1[LG_TAB];
    tableau_de entier tab2[LG_TAB];
    entier i;
debut
    i := 0;
    tantque (i < LG_TAB)
        tab1[i] := modulo((1000+i)*(1000+i+i)+i,LG_TAB);
        tab2[i] := tab1[i];
        i := i + 1;
    fintantque
    alaligne();
    afficherln("*** Tri par fusion");
    afficherTab(@tab1,40);
    afficherln("DEBUT");
    triParFusion(@tab1);
    afficherln("FIN");
    afficherTab(@tab1,40);
    alaligne();
    afficherln("*** Tri par sélection");
    afficherTab(@tab2,40);
    afficherln("DEBUT");
    triParSelection(@tab2);
    afficherln("FIN");
    afficherTab(@tab2,40);
fin

TestAlgo - Interprétation engagée.

*** Tri par fusion
0 3003 6010 9021 2036 5055 8078 1105 4136 7171 210 3253 6300 9351 2406 5465 8528 1595 4666 7741 820
DEBUT
FIN
0 0 1 1 3 3 3 3 3 3 3 3 3 3 5 5 6 6 10 10 11 11 15 15 16 16 20 20 21 21 25 25 26 26 28 28 28 28 28 2

*** Tri par sélection
0 3003 6010 9021 2036 5055 8078 1105 4136 7171 210 3253 6300 9351 2406 5465 8528 1595 4666 7741 820
DEBUT
FIN
0 0 1 1 3 3 3 3 3 3 3 3 3 3 5 5 6 6 10 10 11 11 15 15 16 16 20 20 21 21 25 25 26 26 28 28 28 28 28 2

TestAlgo - Fin de l'interprétation.
```

## 2.2 Principe générale




## 2.3 Procédure `copierTab()`

```
##
# copie une partie du tableau dans un autre tableau
# @param tableau source des valeurs à copier
# @param indice de départ de la copie dans la source
# @param indice de fin de la copie dans la source
# @param tableau recevant les valeurs
# @param indice de départ de la copie dans la destination
##
procedure copierTab(par_ref tableau_de entier source,
                    entier d, entier f, par_ref tableau_de entier dest, entier i)
debut
    tantque (d <= f)
        dest[i] := source[d];
        d := d + 1;
        i := i + 1;
    fintantque
fin
```

## 2.4 Procédure `fusionnerTab()`

```
##
# Fusionne deux parties triées d'un tableau en une seule partie triée
# Les deux parties se suivent
# @param tableau d'entiers
# @param indice de départ de la première partie
# @param indice de départ de la deuxième partie
# @param indice du la fin de la deuxième partie
##
procedure fusionnerTab (par_ref tableau_de entier tab,
                        entier deb1, entier deb2, entier ifin)
```

### 2.4.1 Principe

- Parcourir les deux tableaux en parallèle
- Copier la plus petite valeur dans un autre tableau
- Avancer l'indice de cette partie de tableau
- A la fin de la boucle, le reste du tablau est copié
- La copie du tableau est replacée dans l'original.

### 2.4.2 Corps de boucle

- `i1` indice qui parcourt la première partie du tableau
- `i2` indice qui parcourt la deuxième partie du tableau
- `i` indice qui parcourt la copie du tableau

```
si (tab[i1] > tab[i2]) alors
    tabTri[i] := tab[i2];
    i2 := i2 + 1;
sinon
    tabTri[i] := tab[i1];
    i1 := i1 + 1;
finsi
i := i + 1;
```

### 2.4.3 Conditions de sortie

- `i1 >= deb2` : la première partie du tableau est épuisée
- `i2 > ifin` : la deuxième partie du tableau est épuisée
- Au total : `i1 >= deb2 ou i2 > ifin`

## 2.5 Condition de continuation

- `non(i1 >= deb2 ou i2 > ifin)`!
- qui se réécrit : `i1 < deb2 et i2 <= ifin`

### 2.5.1 Initialisation

```
i  := deb1;
i1 := deb1;
i2 := deb2;
```

### 2.5.2 Terminaison

```
si (i1 >= deb2) alors
    copierTab(@tab, i2, ifin, @tabTri, i);
sinon
    copierTab(@tab, i1, deb2-1,  @tabTri, i );
finsi
copierTab(@tabTri, deb1, ifin, @tab, deb1);
```

## 2.6 Fonction `partieEntiere()`

```
##
# partie entière d'un réel
# @param réel dont la partie entière est à calculer
# @return partie entière du réel
##
fonction partieEntiere (reel r) : entier
var
    entier e;
debut
    e := arrondi(r);
    si (e - r > 0) alors
        e := e - 1;
    finsi
    retourne e;
fin
```

## 2.7 Procédure `triParFusionInterne()`

```
##
# trie par fusion un tableau d'entiers
# @param tableau d'entiers
# @param indice de début du tableau
# @param indice de fin du tableau
##
procedure triParFusionInterne (par_ref tableau_de entier tab, entier d, entier f)
```

### 2.7.1 Principe

- Diviser le "tableau" en deux parties égales
- Trier chaque partie

- Fusionner chaque partie

### 2.7.2 Corps de la procédure

```
milieu := partieEntiere((d+f)/2);
triParFusionInterne(@tab, d, milieu);
triParFusionInterne(@tab, milieu+1, f);
fusionnerTab(@tab, d, milieu+1, f);
```

### 2.7.3 Condition de sortie

- `d == f` : le tableau est réduit à un élément et est donc trié

### 2.7.4 Condition de continuation

- `non(d == f)`
- qui se réécrit : `d <> f`

### 2.7.5 Initialisation et terminaison

Rien

### 2.7.6 Code complet

```
##
# trie par fusion un tableau d'entiers
# @param tableau d'entiers
# @param indice de début du tableau
# @param indice de fin du tableau
##
procedure triParFusionInterne (par_ref tableau_de entier tab, entier d, entier f)
var
    entier milieu;
debut
    si (d <> f) alors
        milieu := partieEntiere((d+f)/2);
        triParFusionInterne(@tab, d, milieu);
        triParFusionInterne(@tab, milieu+1, f);
        fusionnerTab(@tab, d, milieu+1, f);
    finsi
fin
```

## 2.8 Complexités des deux versions

La détail du calcul sera vu lors du prochain module M1103. Le tableau est composé de $n$ entiers.

- le tri par sélection en en $n^2$,
- le trie par fusion est en $n * log(n)$.

# 3 Et pour finir

## 3.1 Nous avons vu

- La construction de boucles imbriquées,
- soit en commençant par la boucle externe,
- soit en commençant par la boucle interne.
- La récursivité qui consiste à réutiliser dans une procédure cette même procédure.
- Des algorithmes produisant le même résultat ne sont pas équivalent en terme de temps d'exécution.