Cours 6

M.Adam – N.Delomez – JF.Kamp – L.Naert

9 août 2022

Table des matières

1 Boucles imbriquées **3**
1.1 Définition . . . 3
1.1.1 Exemple . . . 3
1.1.2 Le code complet . . . 4
1.2 Méthode . . . 5
1.2.1 Principe générale de l'exemple . . . 5
1.3 Boucle externe d'abord . . . 5
1.3.1 Boucle externe . . . 5
1.3.2 Boucle interne . . . 6
1.3.3 Le code complet des deux boucles . . . 8
1.4 Boucle interne d'abord . . . 8
1.4.1 Boucle interne . . . 8
1.4.2 Boucle externe . . . 9
1.5 Méthode de Test . . . 10
1.5.1 Avec AlgoTouch . . . 11
1.6 Remarques . . . 11

2 La récursivité avec le tri par fusion **11**
2.1 Comparaison de deux tris . . . 11
2.2 Principe général du tri fusion . . . 12
2.3 Méthode `copierTab()` . . . 12
2.4 Méthode `fusionnerTab()` . . . 13
2.4.1 Principe . . . 13
2.4.2 Corps de boucle . . . 13

2.4.3 Conditions de sortie 14
2.5 Condition de continuation 14
2.5.1 Initialisation 14
2.5.2 Terminaison 14
2.5.3 Code complet 14
2.6 Méthode `triParFusionInterne()` 15
2.6.1 Principe 15
2.6.2 Corps de la méthode 15
2.6.3 Condition d'arrêt 15
2.6.4 Action associée au cas d'arrêt 16
2.6.5 Initialisation 16
2.6.6 Terminaison 16
2.6.7 Code complet 16
2.7 Méthode `triParFusion()` 16
2.8 Complexités des deux versions 17

3 Et pour finir **17**
3.1 Nous avons vu 17

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 éthode `estInclus()` qui détermine si un tableau d'entiers est inclus dans un autre.

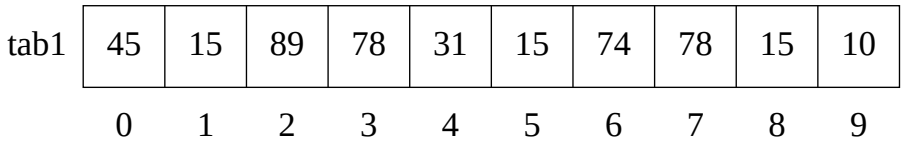
```
/**
 * détermine si toutes les valeurs du premier tableau sont dans le second
 * @param tab1 tableau des valeurs incluses
 * @param tab2 tableau de valeurs
 * @return vrai ssi toutes les valeurs du premier tableau sont dans le second
 */
boolean estInclus (int[] tab1, int[] tab2)
```

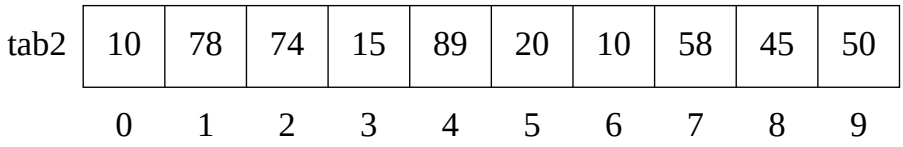
Inclus ou pas ?

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
	0	1	2	3	4	5	6	7	8	9

Inclus ou pas ?





1.1.2 Le code complet

```
/**
 * détermine si toutes les valeurs du premier tableau sont dans le second
 * @param tab1 tableau des valeurs incluses
 * @param tab2 tableau de valeurs
 * @return vrai ssi toutes les valeurs du premier tableau sont dans le second
 */
boolean estInclus (int[] tab1, int[] tab2) {
    int i;
    int j;
    boolean present;
    i = 0;
    present = true;
    while (i < tab1.length && present) {
        j = 0;
        present = false;
        while (j < tab2.length && !present) {
            if (tab1[i] == tab2[j]) {
                present = true;
            }
            j = j + 1;
        }
        i = i + 1;
    }
    return present;
}
```

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.

Une stratégie consiste à remplacer la boucle interne par une méthode. C'est ce que nous avons fait jusqu'à maintenant. Allons plus loin !

L'objectif est d'utiliser pour chaque boucle, intérieure et extérieure, la méthode vue précédemment.

Deux stratégies :

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

1.2.1 Principe générale de l'exemple

Objectif : savoir si le tableau `tab1` d'entiers est inclus dans le tableau `tab2` d'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 :

```
int i;
boolean present;
```

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

L'idée est de programmer plus tard la boucle interne qui constitue une recherche d'une valeur dans un tableau.

Conditions de sortie de la boucle externe

- `i >= tab1.length` pour la sortie de tableau `tab1`
- `!present` car la valeur `tab[i]` n'est pas dans `tab2`.
- au total : `i >= tab1.length || !present`

Condition de continuation de la boucle externe

- `!(i >= tab1.length || !present)`
- qui s'écrit aussi : `i < tab1.length && present`

Initialisation de la boucle externe

```
i = 0;           // commencer au début du tableau
present = true;  // par défaut tab1 dans tab2
```

Terminaison de la boucle externe

```
return present;
```

Code de la boucle externe

```
i = 0;           // commencer au début du tableau
present = true;  // par défaut tab1 dans tab2
while (i < tab1.length && present) {
    //
    // tab1[i] dans tab2 ?
    // present vaut
    // - vrai si tab1[i] est dans tab2,
    // - faux sinon
    //
    i = i + 1;
}
return present;
```

A ce niveau, il faut encore programmer la boucle interne dont le rôle est précisé en commentaire.

1.3.2 Boucle interne

La boucle interne a été définie par :

```
//
// tab1[i] dans tab2 ?
// present 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 :

```
int j;
```

Corps de boucle interne :

```
if (tab1[i] == tab2[j]) {
    present = true;
}
j = j + 1;
```

Conditions de sortie

- `j <= tab2.length` : tout le tableau a été parcouru,
- `present` : la valeur `tab1[i]` a été trouvée,
- au total : `j >= tab2.length || present`

Condition de continuation de la boucle interne

- La négation de la condition de sortie : `!(j >= tab2.length || present)`
- qui s'écrit aussi : `j < tab2.length && !present`

Initialisation de la boucle interne

```
j = 0;           // commencer au début du tableau
present = false; // 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 = false; // la valeur tab[i] n'est pas trouvée
while (j < tab2.length && !present) {
    if (tab1[i] == tab2[j]) {
        present = true;
    }
    j = j + 1;
}
```

1.3.3 Le code complet des deux boucles

```
int i;
int j;
boolean present;
i = 0;
present = true;
while (i < tab1.length && present) {
    j = 0;
    present = false;
    while (j < tab2.length && !present) {
        if (tab1[i] == tab2[j]) {
            present = true;
        }
        j = j + 1;
    }
    i = i + 1;
}
return present;
```

1.4 Boucle interne d'abord

Principe général : L'idée est penser d'abord au rôle de la boucle interne. Sur l'exemple, il parait assez évident qu'il faudra chercher la présence d'une valeur de `tab1`, `tab1[i]` dans `tab2`.

- Chercher si un élément `tab1[i]` 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 :

```
int i;
int j;
boolean present;
```

Corps de boucle interne :

```
if (tab1[i] == tab2[j]) {
    present = true;
}
j = j + 1;
```

Conditions de sortie

- `j >= tab2.length` : tout le tableau a été parcouru,
- `present` : la valeur `tab1[i]` a été trouvée,
- au total : `j >= tab2.length || present`

Condition de continuation de la boucle interne

- La négation de la condition de sortie : `!(j >= tab2.length || present)`
- qui s'écrit aussi : `j < tab2.length && !present`

Initialisation de la boucle interne

```
j = 0;           // commencer au début du tableau
present = false; // 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 = false; // la valeur tab[i] n'est pas trouvée
while (j < tab2.length && !present) {
    if (tab1[i] == tab2[j]) {
        present = true;
    }
    j = j + 1;
}
```

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 = false; // la valeur tab[i] n'est pas trouvée
while (j < tab2.length && !present) {
    if (tab1[i] == tab2[j]) {
        present = true;
    }
    j = j + 1;
}
i = i + 1;
```

Conditions de sortie de la boucle externe

- `i >= tab1.length` pour la sortie de tableau `tab1`
- `!present` car la valeur `tab[i]` n'est pas dans `tab2`.
- au total : `i >= tab1.length || !present`

Condition de continuation de la boucle externe

- `!(i >= tab1.length || !present)`
- qui s'écrit aussi : `i < tab1.length && present`

Initialisation de la boucle externe

```
i = 0;           // commencer au début du tableau
present = true;  // par défaut tab1 dans tab2
```

Terminaison de la boucle externe

```
return present;
```

Code de la boucle externe

```
i = 0;           // commencer au début du tableau
present = true;  // par défaut tab1 dans tab2
while (i < tab1.length && present) {
    j = 0;           // commencer au début du tableau
    present = false; // la valeur tab[i] n'est pas trouvée
    while (j < tab2.length && !present) {
        if (tab1[i] == tab2[j]) {
            present = true;
        }
        j = j + 1;
    }
    i = i + 1;
}
return present;
```

1.5 Méthode de Test

```
$
*** testEstInclusTab
estInclus({45, 15, 89, 78, 50, 15, 74, 78, 15, 10},
          {10, 78, 74, 15, 89, 20, 10, 58, 45, 50} = true  : OK
estInclus({45, 15, 89, 78, 31, 15, 74, 78, 15, 10},
          {10, 78, 74, 15, 89, 20, 10, 58, 45, 50} = false : OK

------------------
(program exited with code: 0)
```

1.5.1 Avec AlgoTouch

- Le programme AlgoTouch : `https://tinyurl.com/C06AGT01`
- La vidéo de la construction : `https://tinyurl.com/C06AGTYT01`

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 méthode.

2 La récursivité avec le tri par fusion

2.1 Comparaison de deux tris

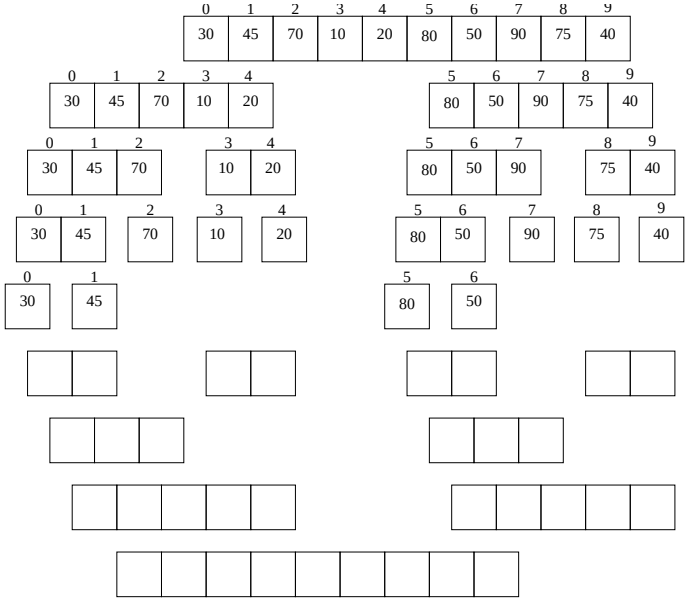
```
/**
 * compare le tri par fusion et par sélection
 */
void comparaison() {
    int[] tab1 = new int[LG_TAB];
    int[] tab2 = new int[LG_TAB];
    int i = 0;
    while (i < LG_TAB) {
        tab1[i] = (int) (Math.random() * LG_TAB);
        tab2[i] = tab1[i];
        i = i + 1;
    }

    System.out.println ();
    System.out.println ("*** Tri par fusion");
    afficherTab(tab1);
    System.out.println ("DEBUT");
    triFusion(tab1);
    System.out.println ("FIN");
    afficherTab(tab1);
    System.out.println ();
    System.out.println ("*** Tri par sélection");
    afficherTab(tab2);
    System.out.println ("DEBUT");
    triSelection(tab2);
    System.out.println ("FIN");
    afficherTab(tab2);
}
```

La valeur de `LG_TAB` est 500 000.

```
$

*** Tri par fusion
{3677, 338242, 287536, 324493, 94807, 24731, 229934, 484731, ...
DEBUT
FIN
{0, 1, 1, 3, 5, 6, 6, 7, 7, 8, 9, 9, 10, 13, 14, 14, 17, 18, ...

*** Tri par sélection
{3677, 338242, 287536, 324493, 94807, 24731, 229934, 484731, ...
DEBUT
FIN
{0, 1, 1, 3, 5, 6, 6, 7, 7, 8, 9, 9, 10, 13, 14, 14, 17, 18, ...


------------------
(program exited with code: 0)
```

A priori, aucune différence entre les deux exécutions !

2.2 Principe général du tri fusion



2.3 Méthode `copierTab()`

```
/**
 * Copie une partie du tableau dans un autre tableau
 * @param source tableau source des valeurs à copier
```

```
 * @param d      indice de départ de la copie dans la source
 * @param dest   tableau recevant les valeurs
 * @param i      indice de départ de la copie dans la destination
 */
void copierTab(int[] source, int d, int[] dest, int i) {
    while (d < source.length) {
        dest[i] = source[d];
        d = d + 1;
        i = i + 1;
    }
}
```

2.4 Méthode `fusionnerTab()`

```
/**
 * Fusionne deux tableaux triés en un tableau trié
 * @param tab1 tableau trié d'entiers
 * @param tab2 tableau trié d'entiers
 * @return un tableau trié d'entiers contenant les valeurs des deux passés en paramètre
 */
int[] fusionnerTab (int[] tab1, int[] tab2)
```

2.4.1 Principe

- Parcourir les deux tableaux en parallèle
- Copier la plus petite valeur dans le nouveau tableau `tab`
- Avancer l'indice de ce tableau
- A la fin de la boucle, le reste du tableau est copié

2.4.2 Corps de boucle

- `i1` indice qui parcourt le premier tableau `tab1`
- `i2` indice qui parcourt le deuxième tableau `tab2`
- `i` indice qui parcourt la copie du tableau `tab`

```
        if (tab1[i1] > tab2[i2]) {
                tab[i] = tab2[i2];
                i2 = i2 + 1;
            } else {
                tab[i] = tab1[i1];
                i1 = i1 + 1;
            }
        i = i + 1;
```

2.4.3 Conditions de sortie

- `i1 >= tab1.length` : le premier tableau est épuisé
- `i2 >= tab2.length` : le deuxième tableau est épuisé
- Au total : `i1 >= tab1.length || i2 >= tab2.length`

2.5 Condition de continuation

- `!i1 >= tab1.length || i2 >= tab2.length`
- qui se réécrit : `i1 < tab1.length && i2 < tab2.length`

2.5.1 Initialisation

```
i  = 0;
i1 = 0;
i2 = 0;
```

2.5.2 Terminaison

```
if (i1 >= tab1.length) {
    copierTab(tab2, i2, tab, i);
} else {
    copierTab(tab1, i1, tab, i);
}
return tab;
```

2.5.3 Code complet

```
/**
 * Fusionne deux tableau trié en un tableau trié
 * @param tab1 tableau trié d'entiers
 * @param tab2 tableau trié d'entiers
 * @return un tableau trié d'entiers contenant les valeurs des deux passés en paramètre
 */
int[] fusionnerTab (int[] tab1, int[] tab2) {
    int[] tab = new int[tab1.length + tab2.length];
    int i  = 0;
    int i1 = 0;
    int i2 = 0;
    while (i1 < tab1.length && i2 < tab2.length) {
        if (tab1[i1] > tab2[i2]) {
            tab[i] = tab2[i2];
            i2 = i2 + 1;
        } else {
```

```
            tab[i] = tab1[i1];
            i1 = i1 + 1;
        }
        i = i + 1;
    }

    if (i1 >= tab1.length) {
        copierTab(tab2, i2, tab, i);
    } else {
        copierTab(tab1, i1, tab, i);
    }
    return tab;
}
```

2.6 Méthode `triParFusionInterne()`

```
/**
 * trie par fusion un tableau d'entiers
 * @param tab tableau d'entiers
 * @param d   indice de debut du tableau
 * @param f   indice de fin du tableau
 */
int[] triFusionInterne (int[] tab, int d, int f)
```

2.6.1 Principe

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

2.6.2 Corps de la méthode

```
    int milieu = (d + f) / 2;
    int[] t1;
    int[] t2;
    t1 = triFusionInterne (tab, d, milieu);
    t2 = triFusionInterne (tab, milieu + 1, f);
    t = fusionnerTab (t1, t2);
```

2.6.3 Condition d'arrêt

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

2.6.4 Action associée au cas d'arrêt

```
    t = new int[1];
    t[0] = tab[d];
```

2.6.5 Initialisation

```
    int[] t = null;
```

2.6.6 Terminaison

```
    return t;
```

2.6.7 Code complet

```
/**
 * trie par fusion un tableau d'entiers
 * @param tab tableau d'entiers
 * @param d   indice de debut du tableau
 * @param f   indice de fin du tableau
 */
int[] triFusionInterne (int[] tab, int d, int f) {
    int[] t = null;
    if (d == f) {
        t = new int[1];
        t[0] = tab[d];
    } else {
        int milieu = (d + f) / 2;
        int[] t1;
        int[] t2;
        t1 = triFusionInterne (tab, d, milieu);
        t2 = triFusionInterne (tab, milieu + 1, f);
        t = fusionnerTab (t1, t2);
    }
    return t;
}
```

2.7 Méthode `triParFusion()`

Maintenant il reste à écrire la méthode `triFusion()`.

```
    void triFusion (int[] tab){
        int[] t;
        t = triFusionInterne(tab, 0, tab.length - 1);
        copierTab(t, 0, tab, 0);
```

```
}
```

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 $\theta(n^2)$,
- le tri par fusion est en $\theta(nlog(n))$.

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 méthode cette même méthode.
- Des algorithmes produisant le même résultat ne sont pas équivalent en terme de temps d'exécution.