Les tableaux à une dimension

Exceptions , recherche et tri

Les exceptions

- Une exception est une erreur qui se produit durant l'exécution d'un programme lorsqu'une situation anormale est rencontrée. Une exception non traitée provoque l'arrêt abrupt du programme, ce qui n'est pas recommandé.
- Exemple: les exceptions ci-dessous se produisent lors d'une mauvaise manipulation des tableaux.

Exception	Situation qui cause l'exception
ArrayIndexOutOfBoundsException	Tentative d'accéder à un indice du tableau hors des bornes.
NegativeArraySizeException	Tentative de créer un tableau avec une taille négative.
NullPointerException	Tentative d'utiliser une référence d'objet ayant la valeur null.

Les Exceptions: Exemple NegativeArraySizeException

- L'exécution du programme ci-dessous provoque une exception de type NegativeArraySizeException. Comme cette erreur n'est pas traitée, elle remonte à travers la pile d'appels jusqu'à la méthode main et provoque l'interruption du programme

```
4      public static int[] generer(int taille) {
5          int[] tabNombres = new int[taille];
6          for (int i = 0; i < tabNombres.length; i++) {
7              tabNombres[i] = (int) (Math.random() * 100);
8          }
9          return tabNombres;
10     }
11
12     public static void main(String[] args) {
13         int[] nombres = generer(-2);
14         afficher(nombres);
15     }
```

- Le message de l'exception affiché dans la console indique, en commençant par le haut, les informations suivantes:
  - le nom de l'exception: NegativeArraySizeException;
  - la ligne qui a généré l'exception: la ligne 5 de la méthode generer de la classe Aleatoire;
  - la méthode qui a appelé cette méthode: la ligne 13 de la méthode main.

```
<terminated> Aleatoires [Java Application] C:\Program Files\Java\jdk1.8.0_152\bin\javaw.exe (5 févr. 2019 22:16:12)
Exception in thread "main" java.lang.NegativeArraySizeException
        at Aleatoires.generer(Aleatoires.java:5)
        at Aleatoires.main(Aleatoires.java:13)
```

Interception d'une exception

- Java possède un mécanisme de gestion des exceptions qui permet d'intercepter les messages d'erreur avant qu'ils ne remontent jusqu'à la méthode main et provoquer l'interruption du programme.
- Ce mécanisme consiste à identifier les instructions qui sont susceptibles de provoquer une exception et à les entourer dans un bloc **try {..}.** On peut alors intercepter l'exception quand elle aura lieu dans le un bloc **catch{}** où on proposera une solution de contournement.
- Voici le modèle de **try..catch**

```
début du programme
try {
     instruction susceptible de déclencher une exception
} catch (nom de l'exception e) {
     instruction de contournement ou message d'erreur
 }
suite du programme
```

- L'instruction try..catch peut avoir plusieurs blocs catch et un bloc finaly. Nous détaillerons cette instruction dans les cours prochains.

Interception d'une exception: exemple (1/2)

- Dans le programme précèdent, on peut intercepter l'exception NegativeArraySizeException si jamais elle est provoquée par l'instruction `int[] tabNombres = new int[taille].` Dans ce cas , on peut et créer un tableau vide(de 0 élément).

```
4    public static int[] generer(int taille) {
5        int[] tabNombres;
6        try {
7            tabNombres = new int[taille];
8
9        } catch (NegativeArraySizeException e) {
10           tabNombres = new int[0];
11       }
12       for (int i = 0; i < tabNombres.length; i++) {
13           tabNombres[i] = (int) (Math.random() * 100);
14       }
15       return tabNombres;
16   }
```

- **Fonctionnement**

  L'instruction se trouvant dans le bloc try est exécutée.

  - S'il elle n'a pas provoqué d'exception, le bloc **catch** est **ignoré** et le programme poursuit son exécution à la ligne 12.
  - Si elle a provoqué une exception de type **NegativeArraySizeException** , le bloc catch est exécuté (création d'un tableau de taille 0) et le programme poursuit son exécution à la ligne 12.

- **Note**: bien que ces exceptions puissent être traitées par des blocs try catch, il est fortement déconseillé de le faire. Il est préférable d'analyser le code et de le modifier pour éviter qu'elles n'apparaissent.

Interception d'une exception: exemple (2/2)

- On peut mettre dans le bloc try d'autres instructions que l'instruction à risque. Exemple:

```
4    public static int[] generer(int taille) {
5        int[] tabNombres;
6        try {
7            tabNombres = new int[taille];
8            for (int i = 0; i < tabNombres.length; i++) {
9                tabNombres[i] = (int) (Math.random() * 100);
10           }
11       } catch (NegativeArraySizeException e) {
12           tabNombres = new int[0];
13       }
14       return tabNombres;
15   }
```

- Si l'instruction de la ligne 7 provoque une exception, le programme ignore l'instruction for qui suit dans le bloc try et saute directement au bloc catch, à la ligne 11, puisque l'exception est de type NegativeArraySizeException.

Exercice1

- Considérez la classe Exercice1.java
  - La méthode **mettreAZero** reçoit en paramètre un tableau d'entiers et un entier représentant un indice du tableau. Elle met à zéro la case indice de ce tableau. Si indice est en dehors des limites du tableau, elle généré une exception de type IndexOutOfBoundsException. Lancez la méthode main pour le confirmer.

```
private static int[] mettreAZero(int[] tabNombres, int indice)
```

- **Travail à faire:**
  - Modifiez le code de cette méthode pour gérer l'exception de type IndexOutOfBoundsException. Ajoutez les bloc **try{}** et le bloc **catch (IndexOutOfBoundsException e){}**
  - Dans le cas où l'exception est générée, la méthode doit retourner le même tableau que celui reçu en paramètre et affiche un message dans la console comme celui donné ci-dessous.

```
La case 6 n'existe pas aucun changement au tableau
```

La recherche dans un tableau

La recherche d'éléments dans un tableau

- Si le tableau est trié, on peut appliquer l'algorithme de recherche binaire (en anglais, binary search) appelé également recherche dichotomique.
- Si le tableau n'est pas trié, on peut utiliser la recherche séquentielle.

Recherche séquentielle (1/2)

- **Principe**

  comparer les éléments du tableau un à un avec l'élément recherché. Arrêt quand l'élément est trouvé ou si c'est la fin de tableau.

- **Exemple**

  Soit **tableau** un tableau d'éléments de type entier et **element** la valeur à rechercher.

  L'algorithme ci-dessous retourne le premier indice de element dans le tableau s'il existe, sinon il retourne **-1**. L'algorithme ne recherche pas toutes les occurrences de element. Voir la diapositive suivante.

Recherche séquentielle (2/2)

```
Algorithme rechercheSequentielle(tableau[] type entier, element type entier) type entier
    entrée: tableau[] un  tableau d'entiers
                element un entier à rechercher dans le tableau
    sortie:  le premier indice de element dans le tableau s'il existe ou -1 sinon
    variables: i , indiceElement de type entier
début
   indiceElement ← -1;
    i ← 0
    tant que ( element ≠tableau[ i] ET  indice <= tableau.longueur -1) faire
        i ← i+ 1
    fin tant que
    si (tableau [i] = element) alors
        indiceElement ← i;
   fin Si
   retourner indiceElement;
fin RechercheSequentielle
```

Recherche binaire (1/4)

- Cette méthode ne s'applique que si le **tableau est déjà trié.**
- L'algorithme s'apparente à la technique « Diviser pour régner ».
- Le principe

  Comparer l'élément à rechercher avec la valeur de la case au milieu du tableau ; s'ils sont égaux, on a trouvé. Sinon, si l'élément cherché est plus petit que l'élément du milieu, chercher l'élément dans la moitié gauche du tableau, sinon chercher l'élément dans la moitié droite du tableau. On réapplique ensuite le même principe.

https://fr.wikipedia.org/wiki/Recherche_dichotomique

Recherche binaire (2/4)

- **Algorithme**

```
Algorithme  rechecheBinaire (tableauTrie[] type entier, element type entier) type entier
        entrée : tableauTrie  un tableau d'entiers trié
                 element de type entier (élément à chercher)
        sortie:  le premier  indice de element dans le tableau  s'il existe ou -1 sinon
        variables locales: indiceElement, milieu type entier
début
      indiceElement ← -1;
      milieu ←  tableauTrie.longueur div 2
      si (tableauTrie[milieu] = element)
            indiceElement ← milieu;
      sinon si (element > (tableauTrie[milieu]) {
            indiceElement ←rechercheADroite(tableauTrie, milieu, element)
      sinon { //element < (tableauTrie[milieu]
            indiceElement ←rechercheAGauche(tableauTrie, milieu, element)
      fin si
      retourner  indiceElement
fin RechercheBinaire
```

Recherche binaire: sous programme chercherAGauche (3/4)

Le sous-programme **chercherAGauche** recherche dans le tableau de gauche à partir de l'indice **0** à **pivot-1** la valeur **element**.

- **Le principe**
  - extraire le tableau de gauche;
  - Si ce tableau comporte au moins un élément, appeler le programme **rechercheBinaire** avec le tableau extrait.

```
Algorithme chercherAGauche (tableauTrie[] type entier, milieu type entier , element type entier) type entier
   entrée : tableauTrie,  un tableauTrie d'entiers trié
   sortie: le premier indice de element dans le tableau s'il existe ou -1 sinon
   variables locales:  gauche[] type entier
                       indiceElement type entier
   début
       indiceElement ← -1;
       gauche ← extraire (tableauTrie, 0, pivot-1)
       si (gauche.longueur > 0) alors
              indiceElement ← rechercheBinaire(gauche, elelment)
       fin si
       retourner indiceElement
   fin chercherAGauche
```

Recherche binaire: sous programme chercherADroite (4/4)

Le sous-programme **chercherADroite** recherche dans le tableau de droite à partir de l'indice **pivot+1** jusqu'au dernier indice du tableau, c'est-à-dire **tableauTrie.longueur-1**.

- **Le principe**
  - extraire le tableau de droite;
  - Si ce tableau comporte au moins un élément, appeler le programme **rechercheBinaire** avec le tableau extrait pour trouver l'indice à droite;
  - ajouter **pivot+1** à l'indice à droite pour obtenir l'indice de element dans le tableau.

```
Algorithme  chercherADroite (tableauTrie[] type entier, pivot type entier , element type entier) type entier
   entrée : tableauTrie,  un tableau d'entiers trié
   sortie: le premier indice de element dans le tableauTrie s'il existe ou -1 sinon
   variables locales:  droite[] type entier
                       indiceElement, indiceAdroite type entier
   début
       indiceElement ← -1;
       indiceAdroite ← -1;
       droite← extraire (tableauTrie, pivo+1, tableauTrie.longueur -1)
       si (droite.longueur > 0) alors
              indiceAdroite ← rechercheBinaire(droite, element)
       fin si
       si (indiceAdroite   ≠ -1) alors
              indiceElement ← pivot + 1 + indiceAdroite
       fin si
       retourner indiceElement
   fin chercherADroite
```

Recherche binaire: trace

element =85

tableauTrie

0	1	2	3	4	5	6
45	**60**	**75**	**80**	**85**	**90**	**95**

Appel #1 TableauTrie milieu =7/2=3

0	1	2	**3**	4	5	6
45	**60**	**75**	**80**	**85**	**90**	**95**

tableauTrie[3]< 85

Rechercher à droite: de l'indice 4 (milieu +1) à 6 (TableauTrie.longueur-1)
Ajouter 4 (milieu +1) à l'indice de élément

Appel #2 tableauDroite milieu =3/2=1

0	**1**	2
85	**90**	**95**

tableauTrie[1]> 85

Rechercher à gauche: de l'indice 0 à 0 (milieu -1)
L'indice de l'élément ne change pas

Appel #3 tableauGauche milieu =1/2=0

0
85

tableauTrie[0]= 85

Arrêter la recherche: tableauTrie[milieu] =element
L'indice de element = 4 + 0

Trier des tableaux

Trier des tableaux

- Il existe plusieurs algorithmes de tri, chacun ayant des avantages et des inconvénients.
- Vous allez voir plusieurs algorithmes de tri en structure de données, nous en verrons deux dans ce cours. Le tri par sélection et le tri à bulles.
- Ces algorithmes ont l'avantage d'être simples à comprendre , mais ne sont pas les plus performants.

Tri par sélection(ou par extraction)

- On suppose qu'on veuille trier un tableau en ordre croissant de gauche à droit.
- Le principe:

  Chercher le plus petit élément du tableau pour le mettre en premier, puis repartir du second élément et aller chercher le plus petit élément du tableau pour le mettre en second, etc..

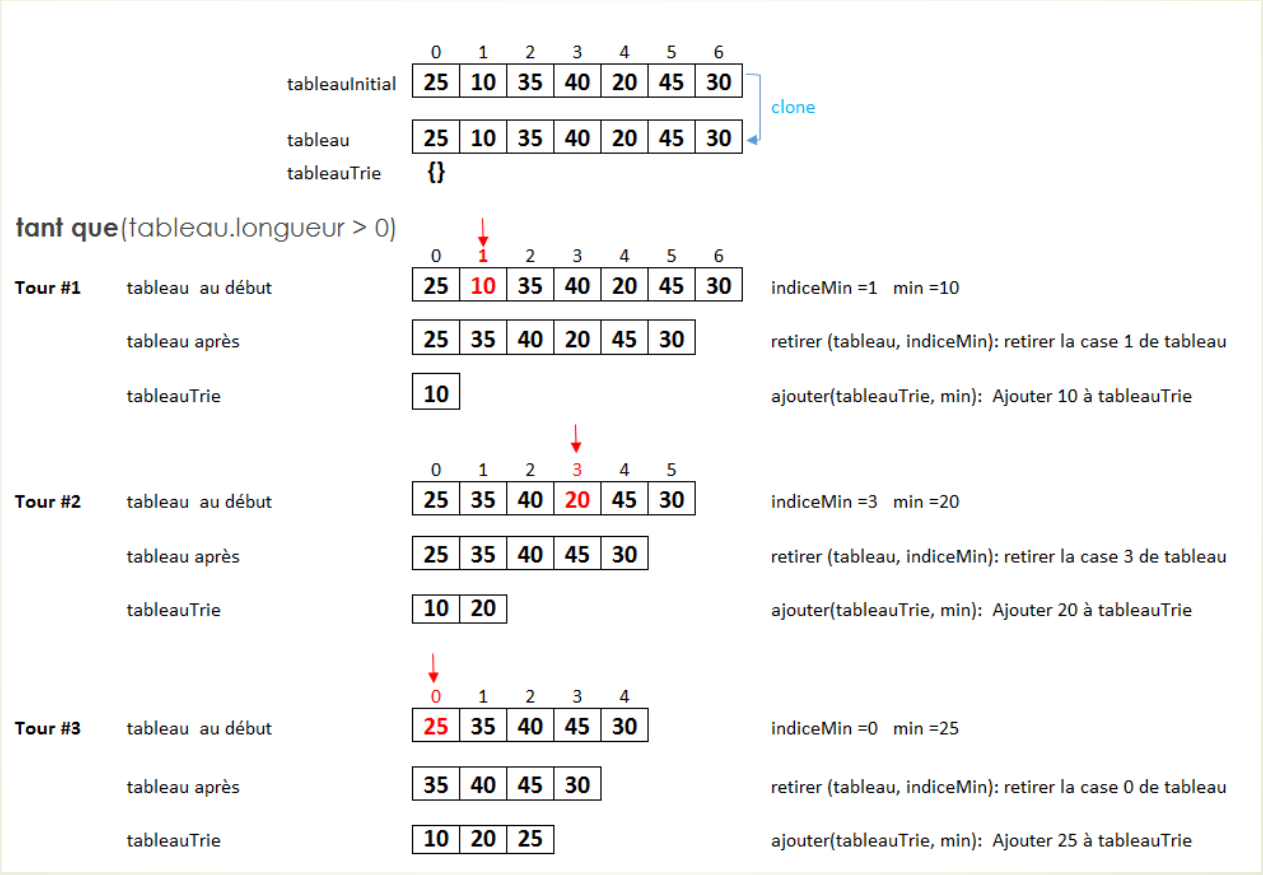
- Animation du tir par extraction

  http://lwh.free.fr/pages/algo/tri/tri_selection.html

Tri par sélection: algorithme

Algorithme triParSelection (tableauInitial [ ] **type entier**) **type [ ]entier**
entrée : tableauInitial: un tableau d'entiers non trié
sortie: un tableau composé des mêmes éléments que le tableau en entré trié par ordre croissant
variables locales: indiceMin, min de **type entier**
tableau[] **type entier**
tableauTrie[] **type entier**
début
tableau← la copie de tableauInitial
initialiser tableauTrie avec 0 entiers
tant que(tableau.longueur > 0)
indiceMin ← indicieMin(tableau)
min ← tableau[indiceMin]
tableau← retirer(tableau, indiceMin);
tableauTrie ← ajouter(tableauTrie, min)
fin tant que
retourner tableauTrie
fin triParSelection

Tri par sélection: trace (1/2)



Tri par sélection: trace(2/2)

		0 1 2 3	
Tour #4	tableau au début	35 40 45 30	indiceMin=3 min =30
	tableau après	35 40 45	retirer (tableau, indiceMin): retirer la case 3 de tableau
	tableauTrie	10 20 25 30	ajouter(tableauTrie, min): Ajouter 30 à tableauTrie
		0 1 2	
Tour #5	tableau au début	35 40 45	indiceMin=0 min =35
	tableau après	40 45	retirer (tableau, indiceMin): retirer la case 0 de tableau
	tableauTrie	10 20 25 30 35	ajouter(tableauTrie, min): Ajouter 35 à tableauTrie
		0 1	
Tour #6	tableau au début	40 45	indiceMin=0 min =40
	tableau après	45	retirer (tableau, indiceMin): retirer la case 0 de tableau
	tableauTrie	10 20 25 30 35 40	ajouter(tableauTrie, min): Ajouter 40 à tableauTrie
		0	
Tour #7	tableau au début	45	indiceMin=0 min =45
	tableau après	{}	retirer (tableau, indiceMin): retirer la case 0 de tableau
	tableauTrie	10 20 25 30 35 40 45	ajouter(tableauTrie, min): Ajouter 45 à tableauTrie

Fin de la boucle : tableau. longueur > 0 est faux

Tri à bulles 

- L'objectif du tri à bulles est de faire remonter les plus grandes valeurs en haut du tableau (ou plutôt à droite).
- Illustration à l'aide d'une animation : http://lwh.free.fr/pages/algo/tri/tri_bulle.html
- **Fonctionnement**
  - Parcourir le tableau et comparer deux à deux les éléments consécutifs, tableau[i] et tableau [i+1], et effectuer une permutation s'ils ne sont pas ordonnés.
  - Après le premier parcours complet du tableau, le plus grand élément est placé à la dernière case du tableau dans sa position définitive et n'a plus besoin d'être reconsidéré.
  - Après le deuxième parcours, les deux plus grands éléments sont à leur position définitive, etc.
  - Recommencer tant que le tableau n'est pas trié.

Tri à bulles: algorithme

Algorithme triBulles (tableauInitial[] **type entier**) **type []entier**

entrée : tableauInitial, un tableau d'entiers non trié

sortie: un tableau composé des mêmes éléments que le tableauInitial mais trié par ordre croissant

variables locales: plafond **type entier**

tableauEnCoursDeTri[] **type entier**

début

tableauEnCoursDeTri ← la copie de tableauInitial

plafond ← tableauEnCoursDeTri.longueur

tant que (**Non** siDejaTrie(tableauEnCoursDeTri)

tableauEnCoursDeTri ← monterBullesSousPlafond(tableauEnCoursDeTri, plafond)

plafond ← plafond -1

fin tant que

retourner tableauEnCoursDeTri

fin triBulles

Note: Les algorithmes **siDejaTrie** et **monterBullesSousPlafond** sont donnés dans les diapositives suivantes.

Tri à bulles: sous programme siDejaTrie

Le sous-programme **siDejaTrie** vérifie si un tableau est trié. L'appel de ce sous-programme permet d'optimiser l'algorithme triBulle dans les meilleurs des cas. En effet, cela permet d'arrêter de monter les bulles dès que le tableau est tiré.

- **Le principe**
  - initialiser un flag **dejaTrie** à **vrai**
  - Parcourir le tableau et comparer deux à deux les éléments consécutifs (i et i+1) . Quand tableau[i] > tableau[i+1] mettre le flag **dejaTrie** à **faux**.
  - Sortir de la boucle quand dejaTrie est faux ou c'est la fin du tableau.

```
Algorithme  siDejaTrie (tableau[] type entier) type booléen
   entrée : tableau,  un tableau d'entiers non trié
   sortie: vrai si tableau est trié , faux sinon
   variables locales: dejaTrie type booléen
                      i type entier
   début
       i ← 0
       dejaTrie ← vrai // on suppose que le tableau est trié en partant
       tant que  (i < tableau.longueur-1 Et  dejaTrie) faire  /Attention aux bornes du tableau avec i+1
              si(tableau[i] > tableau[i+1]) alors
                     dejaTrie ← false //dès qu'une case n'est en ordre, dejaTrie devient  faux pour sortir de la boucle
              fin si
              i← i+1
       fin tant que
       retourner dejaTrie
   fin siDejaTrie
```

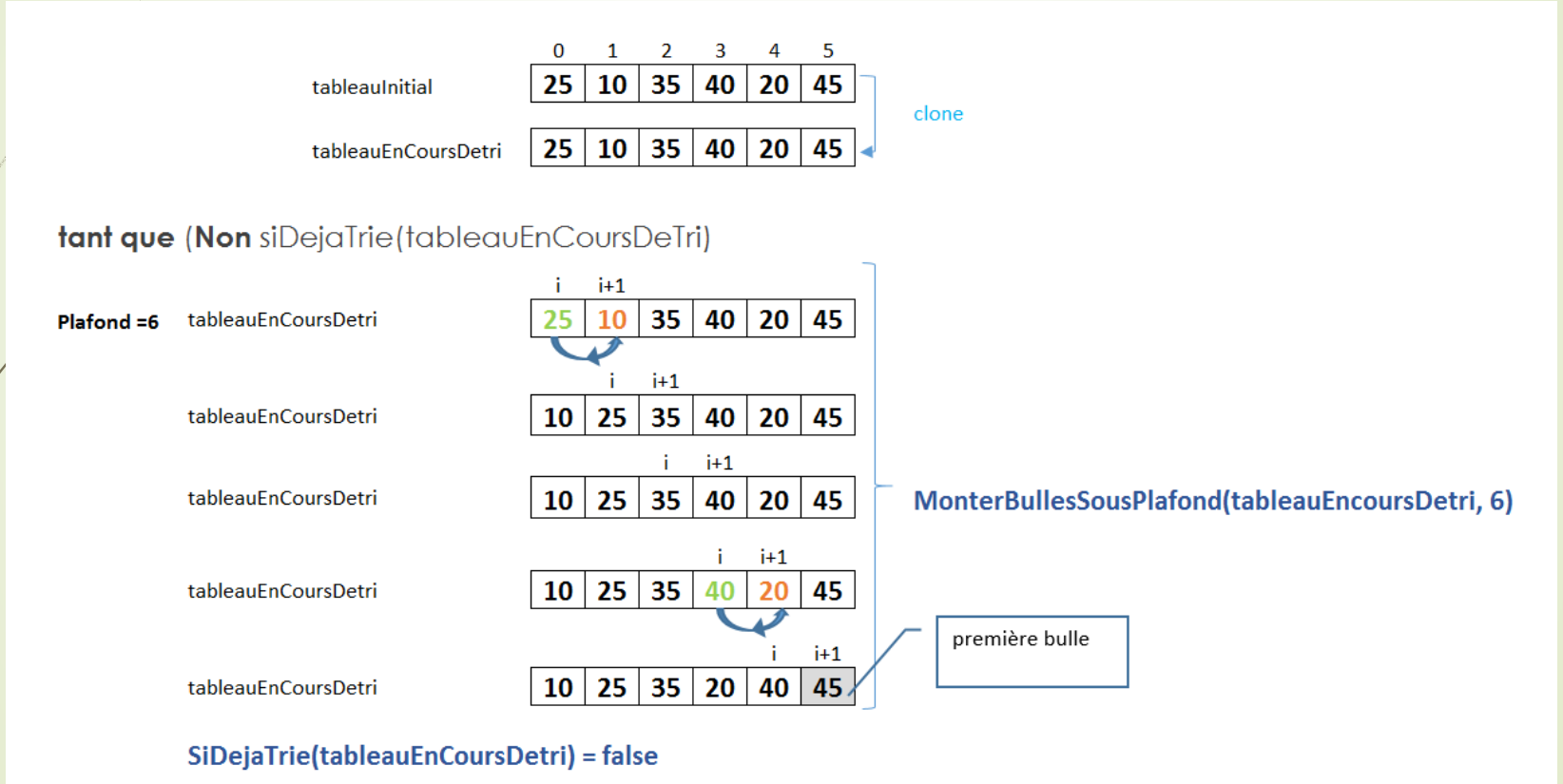
Tri à bulles: sous-programme monterBullesSousPlafond

Le sous-programme **monterBullesSousPlafond** permet de remonter l'élément le plus grand d'un tableau à la dernière case.

- **Le principe**: Parcourir le tableau, du début jusqu'à la fin, et comparer deux à deux les éléments consécutifs. Quand tableau[i] > tableau[i+1], permuter les éléments tableau[i] et tableau[i+1].

```
Algorithme monterBullesSousPlafond (tableau[] type entier, plafond type entier) type []entier
   entrée : tableau,  un tableau d'entiers non trié
   sortie: un tableau contentant les mêmes éléments, mais
                    où l'élément le plus grand est  placé dans la dernière case
   variables locales: tableauBulle[] type entier
                              i, palfond type entier
   début
       tableauBulles ← la copie de  tableau
       pour i de 0 à plafond-1 faire
              si(tableauBulle[i] > tableauBulle[i+1])
                     tableauBulle ←  echanger(tableauBulle, i, i+1)
              fin si
       fin pour
       retourner tableauBulle
   fin monterBullesSousPlafond
```

Tri à bulles : trace(1/2)



Tri à bulles : trace (2/2)