Introduction à la Programmation 1

Séance 10 de cours/TD

Université Paris-Diderot

Objectifs:

- Exécuter sur papier des boucles imbriquées
- Exécuter sur papier des fonctions qui appellent des fonctions
- Reconnaître les erreur de développement
- Débogage "scientifique" d'un programme

1 Exécution de programmes avec boucles imbriquées

Les boucles imbriquées [Cours]

- Lorsqu'une boucle se trouve dans le corps d'une autre boucle, on dit que les boucles sont *imbriquées*.
- Les règles d'exécution vues à la séance 8 s'appliquent normalement aux boucles imbriquées. En conséquence, pour chaque itération de la boucle extérieure, la boucle intérieure fait un tour complet. Intuitivement, la boucle extérieure est lente, la boucle intérieure est rapide.

Exemple d'exécution [Cours]

Pour le programme ci-dessous :

```java
public class ExoBouclesImbriquees{
    public static void main(String[] args){
        String st = "";
        for(int i = 0; i < 2; i++){
            for(int j = 0; j < i; j++){
                st = st + "ab";
            }
            st= st + "c";
        }
    }
}
```

code/ExoBouclesImbriquees.java

On aura la séquence d'exécutions suivante :

PC	st		
3	""		
PC	st	i	
4	""	0	
PC	st	i	j
5	""	0	0
PC	st	i	
8	"c"	0	
PC	st	i	
9	"c"	0	
PC	st	i	
4	"c"	1	
PC	st	i	j
5	"c"	1	0
PC	st	i	j
6	"cab"	1	0
PC	st	i	j
7	"cab"	1	0
PC	st	i	j
5	"cab"	1	1
PC	st	i	
8	"$cabc$"	1	
PC	st	i	
9	"$cabc$"	1	
PC	st	i	
4	"$cabc$"	2	
PC	st		
10	"$cabc$"		

Exercice 1 (Exécution de boucles imbriquées, $\star$)
Décrire l'exécution du programme suivant :

```java
public class ExoBouclesImbriquees2{
    public static void main(String[] args){
        int x = 0;
        int[][] tab = new int [3][];
        for(int i = 0; i < 3; i++){
            tab[i]=new int[i+1];
            for(int j = 0; j <= i; j++){
                tab[i][j] = x;
                x = x + 1;
            }
        }
    }
}
```

code/ExoBouclesImbriquees2.java

□

2 Appel d'une fonction depuis une fonction

Appel d'une fonction depuis une fonction [Cours]

- Rien ne nous empêche d'écrire des fonctions qui appellent elles-mêmes des fonctions.
- Le mécanisme de la pile vu à la séance 8 explique parfaitement comment des appels de fonctions dans des fonctions sont exécutés. La seule chose à remarquer est que la pile des compteurs des programmes peut croître vers la droite, et qu'on peut avoir dans la table de mémoire plusieurs blocs de variables (séparés par une double barre ||).

Exemple d'exécution [Cours]

Pour le programme ci-dessous :

```java
public class Fonctions1{
    public static int f(int x){
        int y = x+x;
        return(y);
    }

    public static int g(int a){
        int b = a*a;
        b = f(b);
        return(b+1);
    }

    public static void main(String[] args){
        int m = 3;
        m = g(m);
    }
}
```

code/Fonctions1.java

On aura la séquence d'exécutions suivante :

PC	m				
14	3				
PC	m	a			
$15 \triangleright 7$	3	3			
PC	m	a	b		
$15 \triangleright 8$	3	3	9		
PC	m	a	b	x	
$15 \triangleright 9 \triangleright 2$	8	3	9	9	
PC	m	a	b	x	y
$15 \triangleright 9 \triangleright 3$	8	3	9	9	18
PC	m	a	b		
$15 \triangleright 9 \triangleright 4$	8	3	18		
PC	m	a	b		
$15 \triangleright 10$	19				

Exercice 2 (Appels de fonctions, ⋆⋆)

1. *Écrivez une fonction* `static int max(int x, int y)` *qui retourne le plus grand de ses deux paramètres.*
2. *Écrivez une fonction* `static int max3(int a, int b, int c)` *qui se sert deux fois de la fonction* `max` *et retourne le plus grand de ses paramètres.*
3. *Donnez l'exécution du programme suivant en remplaçant les "pointillés" par le code de vos fonctions :*

```java
public class ExoMax{
    public static int max(int x, int y) {
        /*...*/
    }

    public static int max3(int a, int b, int c) {
        /*...*/
    }

    public static void main(String[] args){
        int m = 0;
        m = max3(2, 3, 4);
    }
}
```

code/ExoMax.java

4. *Que se passe-t-il si on renomme les paramètres de la fonction* max3 *en* static int max3(int x, int y, int z) *?*

□

Exercice 3 (Sommes de tableaux, ⋆)

1. *Écrivez une fonction* static int sumTab(int[] t) *qui retourne la somme des éléments du tableau passé en paramètre.*
2. *Écrivez une fonction* static int SumMat(int[][] m) *qui se sert de la fonction* somme_tab *et retourne la somme des éléments du tableau de tableaux passé en paramètre.*
3. *Donnez les huit premières étapes de l'exécution du programme suivant en remplaçant les "pointillés" par le code de vos fonctions :*

```java
public class ExoSumTab{
    public static int sumTab(int[] t) {
        /*...*/
    }

    public static int sumMat(int[][] m) {
        /*...*/
    }

    public static void main(String[] args){
        int[][] m = {{1, 2, 3}, {4, 5, 6}};
        int sum = sumMat(m);
    }
}
```

code/ExoSumTab.java

□

La notion de portée [Cours]

- Une variable, ou un paramètre d'une fonction, est toujours locale à la fonction dans laquelle elle est définie.
- La *portée* d'une variable est la partie du texte de programme dans laquelle la variable existe, elle commence à la déclaration de la variable, et se termine avec la fin de la fonction.
- Quand l'exécution d'un programme appelle une fonction, on quitte temporairement la portée de la fonction où il y a l'appel, et on entre dans la portée de la fonction appelée. Quand l'appel de la fonction est terminé, on revient dans la portée de la première fonction.

– Par conséquent, les colonnes de la table de mémoire qui peuvent changer pendant l'exécution d'un programme sont toujours celles qui se trouvent tout à droite, après la dernière double barre ||.
– Il est permis que deux fonctions définissent des variables locales avec le même nom. Dans ce cas il s'agit bien de deux variables différentes.

Exemple [Cours]

Pour le programme ci-dessous :

```java
public class Portees{
    public static void main(String[] args){
        int x = 42;
        int y = 73;
        x = g(y);
    }
    public static int g(int x){
        int y = 11;
        y = f(x);
        return(y);
    }

    public static int f(int y){
        int x = 0;
        return(y-7);
    }
}
```

code/Portees.java

On aura la séquence d'exécutions suivante :

PC	x					
3	42					
PC	x	y				
4	42	73				
PC	x	y	x			
$5 \triangleright 7$	42	73	73			
PC	x	y	x	y		
$5 \triangleright 8$	42	73	73	11		
PC	x	y	x	y	y	
$5 \triangleright 9 \triangleright 13$	42	73	73	11	73	
PC	x	y	x	y	y	x
$5 \triangleright 9 \triangleright 14$	42	73	73	11	73	0
PC	x	y	x	y		
$5 \triangleright 9 \triangleright 15$	42	73	73	66		
PC	x	y				
$5 \triangleright 10$	66	73				

3 Erreurs et débogage

Typologies d'erreurs de programmation [Cours]

Dans la pratique de la programmation vous avez déjà rencontré plusieurs types d'erreurs de programmation. Classifions les :

erreurs de syntaxe (à la compilation) le code du programme ne respecte pas les règles de syntaxe du langage de programmation ; le programme n'est pas compris par le compilateur (ou l'interpréteur, le cas échéant). Vous n'arrivez pas à obtenir un programme exécutable.

erreurs de typage (à la compilation ; plus en général : erreur de non-respect de la sémantique du langage de programmation) la syntaxe du programme est correcte, mais l'usage d'une ou plusieurs expressions n'est pas compatible avec leur contexte. Par exemple, l'évaluation de ces expressions donne des valeurs dont le type n'est pas compatible avec le type attendu. Vous n'arrivez pas à obtenir un programme exécutable non plus.

Dans la catégorie plus générale de non-respect de la sémantique du langage de programmation on inclut aussi d'autres erreurs, comme par exemple l'appel d'une fonction jamais définie, l'appel d'une fonction avec un nombre incorrect de paramètres, etc.

non-respect de la spécification le programme est syntaxiquement correct et ne contient aucune erreur de type. Vous arrivez à obtenir un programme exécutable. Dans certain cas, l'exécution du programme donne des résultats (ou un comportement) diffèrent de ce que sa spécification demande. Il s'agit d'erreurs *à l'exécution*.

"échec" un cas particulier du non-respect de la spécification est l'échec à l'exécution, non prévu par la spécification, menant à l'arrêt anticipé de l'exécution d'un programme, par exemple à cause d'une exception. Remarquez qu'une spécification peut *prévoir* l'arrêt d'un programme, même avec une exception. Dans ce cas il ne s'agit pas d'un non-respect de la spécification (mais il peut s'agir d'une spécification mal conçue).

Le compilateur Java aide le développeur à ne pas commettre certains types d'erreurs, notamment :
– un programme avec erreurs de syntaxe *ne compile pas* ;
– un programme avec erreurs de typage *ne compile pas*.
Si un programme compile, vous avez donc la certitude que les erreurs de syntaxe et typage ont toutes été résolues.

Exercice 4 (Bestiaire d'erreurs, ⋆)
Trouver, classifier, et corriger (si possible) toutes les erreurs dans les fragments de code suivants.

```
1 public static void main(String[]] args) {
2         for(int 3i = 0; i++) {
3     printInt(i);
4 }
```

code/SyntCompErrEx.java

```
1  public static void hello(int x) {
2      int n = "an integer";
3      printString( "Hello, " );
4      printInt( x );
5      printString( " world!\n" );
6      return x;
7  }
8
9  public static void main(String[] args) {
10     hello(42);
11     hello("42", 43);
12     bonjour("42);
13 }
```

code/TypeCompErrEx.java

```java
/* spécification (informelle): renverser le tableau d'entiers
 * {2, 5, -12, 8} et afficher le résultat sur la sortie standard
 */
public static void main(String[] args){
    int[] tab = {2, 5, -12, 8};
    int len = intArrayLength(tab);
    int[] revtab = new int[len];
    for (int i = 0; i < len; i++) {
        revtab[i] = tab[len - i];
    }
    for (int i = 0; i < len; i++){
        printInt(tab[i]);
    }
}
```

code/ExcepRunTErrEx1.java

```java
/* proportion des entiers sur la somme d'un tableau
 * entrée: un tableau d'entiers t
 * sortie: un tableau d'entiers dont chaque element en position i vaut
 * t_i/sum ou t_i et l'entier correspondant dans l'array t et sum la
 * la somme des éléments de t
 */
public static int[] proportion(int[] t){
    int sum = 0;
    int len = intArrayLength(t);
    int[] proportion = new int[len];

    for(int i = 0; i < intArrayLength(t); i++) {
        sum = sum + tab[i];
    }

    for(int i = 0; i< intArrayLength(t); i++) {
        proportion[i] = i / sum;
    }

    return proportion;
}
```

code/ExcepRunTErrEx2.java

□

Débogage [Cours]

- Le *debugging* (ou *débogage*) est l'approche qu'on suit pour corriger les erreurs de programmation. Notamment, pour corriger les erreurs de non-respect de la spécification.
- Plus précisément, le débogage s'attaque aux *bugs* (ou *bogues*) : défauts d'un programme qui causent un comportement différent du comportement attendu. Pour pouvoir corriger un bug, il faut d'abord le trouver. Le trouver implique l'identification des lignes de code où l'intuition du développeur sur le comportement du programme se sépare du modèle formelle d'exécution
- Bien que pour déboguer vous pouvez toujours vous baser sur des tests empiriques, de la relecture du code, des intuitions, etc, le débogage proprement dit est une application rigoureuse de la *méthode scientifique* à la programmation.
- Le débogage "scientifique" prévoit donc :
  1. *observation* d'une erreur
  2. formulation d'une *hypothèse* qui explique l'erreur

3. *prévision*, compatible avec l'hypothèse, d'un comportement non encore observé du programme
4. *expérimentation* pour tester l'hypothèse
   - si l'expérience confirme la prévision (hypothèse confirmée), *raffinement* de l'hypothèse
   - si non (hypothèse réfutée), formulation d'une hypothèse alternative
5. répétition des points 3–4 jusqu'à l'obtention d'une hypothèse valide qui ne peut pas être raffinée ultérieurement

– pour *observer une erreur* il ne suffit pas de l'avoir "vu passer" quelques fois, il faut être capable de la *reproduire* systématiquement. Vous devez identifier une série d'entrées pour votre programme qui, *d'une façon déterministe*, mènent au comportement incorrecte du programme. Cela s'applique aussi à toutes les étapes d'expérimentation prévues par l'"algorithme" de débogage ci-dessus. La reproductibilité est l'essence de la méthode scientifique !

Exercice 5 (Déboguer, ⋆⋆)
Tester, déboguer "scientifiquement", et corriger le programme suivant :

```java
class ExoRevArrayBuggy {

    public static String[] reverseArray(String[] t) {
        int len = stringArrayLength(t);
        for (int i = 0; i < len; i++) {
            t[len - i - 1] = t[i];
        }
        return t;
    }

    public static void main(String[] args) {
        String[] revArgs = reverseArray(args);
        printStringArray(revArgs);
    }

    public static void printStringArray(String[] args) {
        for (int i = 0; i < stringArrayLength(args); i++) {
            printString(args[i] + " ");
        }
        printString("\n");
    }

    public static void printString(String s) {
        System.out.print(s);
    }

    public static int stringArrayLength(String[] t) {
        return t.length;
    }

}
```

code/ExoRevArrayBuggy.java

□

Tri par insertion [Cours]

- Un algorithme de tri est un algorithme qui permet de réordonner une collection d'éléments (par exemple un tableau d'entiers) selon un ordre déterminé (p.ex. en ordre croissant). Le tri par insertion est un des algorithmes de tri les plus simples.
- Dans le tri par insertion on maintient une collection d'éléments déjà triés. Cette collection est initialement vide ; à la fin de l'exécution elle contiendra la totalité des éléments, dans le bon ordre.
- À chaque itération, le tri par insertion prend un des éléments encore à trier et l'insère à la bonne position parmi les éléments déjà triés. Le nombre d'éléments déjà triés augmente donc de 1 à chaque itération.

Exercice 6 (Déboguer, ⋆⋆⋆)

Tester, déboguer "scientifiquement", et corriger le programme suivant :

```
1  public class ExoInsertionSortBuggy {
2
3      /* Insère un élément dans un tableau partiellement trié
4       * Entrée : t un tableau d'entiers partiellement trié, t est
5       * trié par ordre croissant jusqu'à l'indice (last - 1).
6       * À la fin de la procédure, t est trié jusqu'à l'indice last.
7       */
8      public static void insert(int x, int[] t, int last) {
9          int i = intArrayLength(t) - 1;
10         while (i > 0 && t[i] >= x) {
11             t[i] = t[i-1];
12             i = i - 1;
13         }
14         t[i] = x;
15     }
16
17     /* Trie le tableau d'entiers donné en entrée
18      * Entrée : t un tableau d'entiers
19      * À la fin de la procédure, le tableau t est trié.
20      */
21     public static void sort(int[] t) {
22         for (int i = 1; i < intArrayLength(t); i++) {
23             insert(t[i], t, i);
24         }
25     }
26
27
28     public static void printIntArray(int[] t) {
29         printString("[ ");
30         for (int i = 1; i < intArrayLength(t); i++) {
31             printInt(t[i]);
32             printString(" ");
33         }
34         printString("]\n");
35     }
36
37     public static void main(String[] args) {
38         int len = stringArrayLength(args);
39         int[] t = new int[len];
40         for (int i = 0; i <= stringArrayLength(args); i++) {
41             t[i] = stringToInt(args[i]);
42         }
43         sort(t);
44         printIntArray(t);
```

```
45      }
46
47      public static int intArrayLength(int[] t) {
48          return t.length;
49      }
50
51      public static int stringArrayLength(String[] t) {
52          return t.length;
53      }
54
55      public static void printInt(int x) {
56          System.out.print(x);
57      }
58
59      public static void printString(String s) {
60          System.out.print(s);
61      }
62
63      public static int stringToInt(String s) {
64          return Integer.parseInt(s);
65      }
66
67  }
```

code/ExoInsertionSortBuggy.java

□

4 Fonctions utilisées

```
1  /*Renvoie la longueur du tableau d'entiers t*/
2  public static int intArrayLength(int[] t) {
3      return t.length;
4  }
5
6  /*Renvoie la longueur du tableau de chaîne de caractères t*/
7  public static int stringArrayLength(String[] t) {
8      return t.length;
9  }
10
11 /* Affiche un entier. */
12 public static void printInt(int x) {
13     System.out.print(x);
14 }
15
16 /* Affiche une chaîne de caractères. */
17 public static void printString(String s) {
18     System.out.print(s);
19 }
20
21
22 /* Renvoie la chaîne de caractère correspondant à un entier */
23 public static int stringToInt(String s) {
24     return Integer.parseInt(s);
25 }
26
```

```java

/*
 * Teste si deux chaînes de caractères s1 et s2 ont le même contenu.
 * Retourne vrai si s1 et s2 sont égales.
 */
public static boolean stringEquals(String s1, String s2) {
    return (s1.equals (s2));
}

/*
 *Attend que l'utilisateur tape une ligne au clavier
 *Retourne la chaîne de caractères correspondante
*/
public static String readLine() {
    return System.console().readLine();
}

/* Retourne la longueur d'une chaîne de caractères. */
public static int stringLength (String t) {
    return t.length();
}

/*
 * Retourne une chaîne constituée du caractère se trouvant à la
 * position $i$ de la chaîne s.
 * Si i est négatif ou en dehors de la chaîne, retourne la chaîne vide.
 */
public static String characterAtPos (String s, int i) {
    String res = "";
    char a;
    if (i >= 0 && i < s.length ()) {
        res = String.valueOf (s.charAt (i));
    }
    return res;
}


/*
* Retourne le nombre de cases d'un tableau de tableau
* d'entiers.
*/
public static int intArrayArrayLength (int [][] t) {
  return  t . length ;
}
```

5 DIY

Exercice 7 (Médiane d'un tableau, ⋆)
On s'intéresse ici à des tableaux de nombres entiers à une seule dimension, possédant un nombre impair d'éléments et dont tous les éléments sont différents. On appelle médiane d'un tel tableau T *l'élément* m *de* T *tel que* T *contienne autant d'éléments strictement inférieurs à m que d'éléments strictement supérieurs à m.*

1. *Écrire une fonction* nbInf *qui, étant donné un tableau d'entiers* T *et un entier* v*, renvoie le nombre d'éléments du tableau* T *strictement inférieurs à* v*.*

2. *Écrire une fonction* mediane *qui, étant donné un tableau* T *satisfaisant les conditions énoncées, renvoie la position de la médiane dans le tableau* T.
3. *Écrire une fonction* verifArray *qui, étant donné un tableau* T *de nombres entiers, renvoie la valeur booléenne* true *si le tableau* T *satisfait effectivement les conditions énoncées et la valeur* false *si ce n'est pas le cas.*
4. *Écrire une fonction* tabInf *qui, étant donné un tableau* T, *retourne un tableau de taille 0 si* T *ne satisfait pas les conditions énoncées et un tableau contenant tous les éléments de* T *inférieurs à la médiane de* T *sinon.*

□

Exercice 8 (Spécification, ⋆⋆⋆)
Écrire un programme `Backdoor` *qui, en boucle infinie, lit une ligne de texte de l'utilisateur à la fois, et l'affiche sur l'écran après l'avoir converti en majuscule les caractères "a", "b", "c", "d", "e", et "f" ; les autres caractères ne sont pas modifiés. Si, par contre, la ligne entrée est le mot secret* "3XzRWo" *(une "backdoor" insérée par le développeur), le programme doit terminer avec une exception après avoir affiché sur l'écran l'ASCII art suivant :*

```
        (__)
        (oo)
  /------\/
 / |    ||
*  /\---/\
   ~~   ~~
```

Vous avez à disposition : la fonction void printString(String s) *pour imprimer sur l'écran,* boolean stringEquals(String s1, String s2) *pour comparer deux chaînes de caractères,* String readLine() *pour lire une ligne de texte de l'utilisateur,* String stringLength(String s) *pour obtenir la longueur d'une chaîne de caractères, et* String characterAtPos(String s, int i) *pour accéder à un caractère dans une chaîne*

À noter : *dans ce cas terminer avec une exception (dans un cas très spécifique) fait partie de la spécification du programme. Il ne s'agit donc pas d'une erreur de programmation. Mais comment causer volontairement une exception ?*

□

Exercice 9 (Championnat, ⋆⋆)
On considère un tableau à 3 dimensions stockant les scores d'un championnat de handball. Pour n *équipes, le tableau aura* n *lignes et* n *colonnes et dans chacune de ses cases on trouvera un tableau à une dimension de longueur 2 contenant le score d'un match (on ne tient pas compte de ce qui est stocké dans la diagonale). Ainsi, pour le tableau championnat* `ch`*, on trouvera dans* `ch[i][j]` *le score du match de l'équipe* `i+1` *contre l'équipe* `j+1` *et dans* `ch[j][i]` *le score du match de l'équipe* `j+1` *contre l'équipe* `i+1`*. De même, pour un score stocké, le premier entier de* `ch[i][j]` *sera le nombre de but(s) marqué(s) par l'équipe* `i+1` *dans le match l'opposant à l'équipe* `j+1`*. Finalement, on suppose que lorsqu'une équipe gagne un match, elle obtient 3 points, 1 seul point pour un match nul et 0 point dans le cas où elle perd le match.*

1. *Écrire une fonction* `numberPoints` *qui prend en arguments un tableau championnat* `ch` *de côté* n *et le numéro d'une équipe (entre 1 à* n*) et qui renvoie le nombre de point(s) obtenu(s) par cette équipe pendant le championnat.*
2. *Écrire une fonction* `storeScore` *qui prend en arguments un tableau championnat* `ch` *de côté* n*, le numéro* i *d'une équipe, le numéro* j *d'une autre équipe et le score du match de* i *contre* j *et qui met à jour le tableau* `ch`*.*
3. *Écrire une fonction* `champion` *qui prend en argument un tableau championnat* `ch` *et qui renvoie le numéro de l'équipe championne. Une équipe est championne si elle a strictement plus de points que toutes les autres. Si plusieurs équipes ont le même nombre de points, alors une équipe est meilleure si elle a marqué strictement plus de buts. Dans le cas d'égalité parfaite (même nombre maximum de points et même nombre maximum de buts marqués), la fonction renverra 0 pour signaler l'impossibilité de désigner un champion.*

□

Exercice 10 (Recherche dichotomique dans un tableau trié, $\star\star$)
On souhaite définir une fonction pour rechercher efficacement un élément dans un tableau d'entiers. La fonction prendra en paramètres un tableau d'entiers t *et un entier* x *et retournera un indice* i *tel que* t[i] *soit égal à* x*, si un tel indice existe, et* -1 *sinon.*

1. *Définir une fonction* searchA *qui vérifie la spécification ci-dessu et qui retourne un résultat* **dès qu'***un indice qui satisfait la spécification a été trouvé.*
2. *Dans quelles situations la fonction* searchA *effectue-t-elle le plus d'accès au tableau ? Dans ce cas, combien de fois accède-t-on au tableau ?*
3. *On suppose maintenant, et dans la suite de l'exercice, que les tableaux d'entiers que reçoit notre fonction sont toujours triés par ordre croissant (c'est-à-dire que* $t[i] \leq t[i+1]$ *tant qu'on est dans les bornes du tableau).*

   Définir une nouvelle fonction searchB *qui assure que si la valeur recherchée est hors des valeurs extrêmes du tableau, on ne fait pas plus de deux accès au tableau avant de retourner* -1.
4. *On conserve la même hypothèse que ci-dessus (les tableaux passés à la fonction sont triés) et on demande de définir une fonction* searchC *qui tire partie du fait que le tableau est trié à partir de l'observation suivante : l'indice recherché se trouve soit dans la première moitié du tableau, soit dans la seconde moitié du tableau et pour savoir dans quelle situation on se trouve, il suffit de comparer une seule valeur du tableau à la valeur recherchée.*

   searchC *utilisera une fonction auxiliaire* searchAux *à définir qui prend en paramètres non seulement un tableau (trié)* t *et une valeur recherchée* x *mais également deux indices* i *et* j *et qui effectue la recherche de la valeur* x *entre les indices* i *et* j *de* t. searchC(t, x) *appellera* searchAux(t, x, 0, intArrayLength (t) $-$ 1).
5. *($\star\star\star$) Dans quelle situation* searchC *accède-t-elle le plus de fois aux valeurs du tableau ? Dans ce cas, combien de fois la fonction accède-t-elle aux cases du tableau ?*

□