

Université de Montréal

Département d'Informatique et de Recherche Opérationnelle

Structures de données

Analyse de Complexité et Piles

Auteur

Franz Girardin

Nourdin Azami

Matricule

20078678

20078678

5 février 2024

Table des matières

2	Chapitre 1	
	Analyse de complexité	
1.1	MistFonction1	2
1.2	MistFonction2	4
1.3	MistFonction3	5
1.4	Approche récursive	5
1.4.1	Analyse de MistFonction3	5
1.4.2	Analyse de MistFonction3	6

6	Chapitre 2	
	Pile simple	

8	Chapitre 3	
	Pile Double	

10	Chapitre 4	
	Pile spéciale	

Analyse de complexité

Exercice 1

Compte tenu des fonctions mystérieuses suivantes, pour chacune d'elles, déterminer quelle est la complexité dans le temps et l'espace de son exécution (Big O) et expliquer ce que vous pensez que la fonction fait. Les réponses simples ne seront pas acceptées, il est nécessaire de justifier votre réponse. Par exemple : Si dans l'exercice la récursion est utilisée, vous pouvez soutenir votre justification en présentant l'arbre de récursion. Toutes vos réponses doivent être incluses dans votre rapport.

```

1 public class MistFonction1{
2
3     public static int mistFonction1(int m, int n) {
4         if (m == 1) && (n == 1) return 1;
5         if (m == 0) || (n == 0) return 0
6         return mistFonction1(m -1, n) + mistFonction1(m, n -1)
7     }
8 }

```

La fonction présente deux **cas de bases**, dont l'un retourne la valeur 1 lorsque *les deux* arguments ont pour valeur 1, alors que l'autre retourne 0 sous la condition que *l'un des deux* arguments a pour valeur 0.

La seconde portion de la fonction engendre des appels récursifs qui prennent fin lorsque les cas de bases sont rencontrés. Nous constatons alors que chaque appel de fonction *hors base* engendre **deux appels**, jusqu'à ce que les conditions d'arrêts soient effectives.

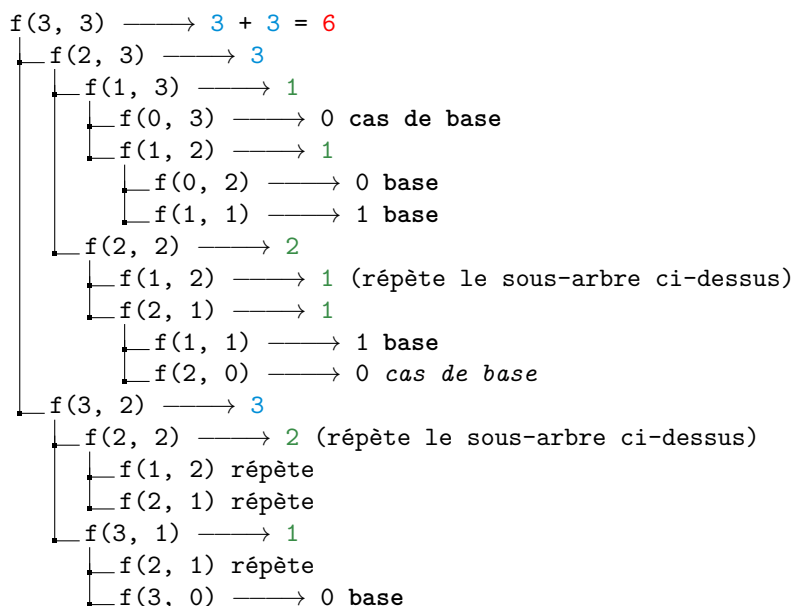


Figure 1.1 – Exemple de $f(3, 3)$

Ainsi, nous pouvons représenter les appels récursifs par un arbre tel qu'à chaque niveau de l'arbre, **le nombre d'appels est doublé** par rapport au précédent. En considérant j comme étant la valeur de m lorsque m atteint 0, et k , la valeur de n lorsque n atteint 1, on obtient l'arbre généralisé à tous les cas possibles qui est présenté à la figure 1.2.

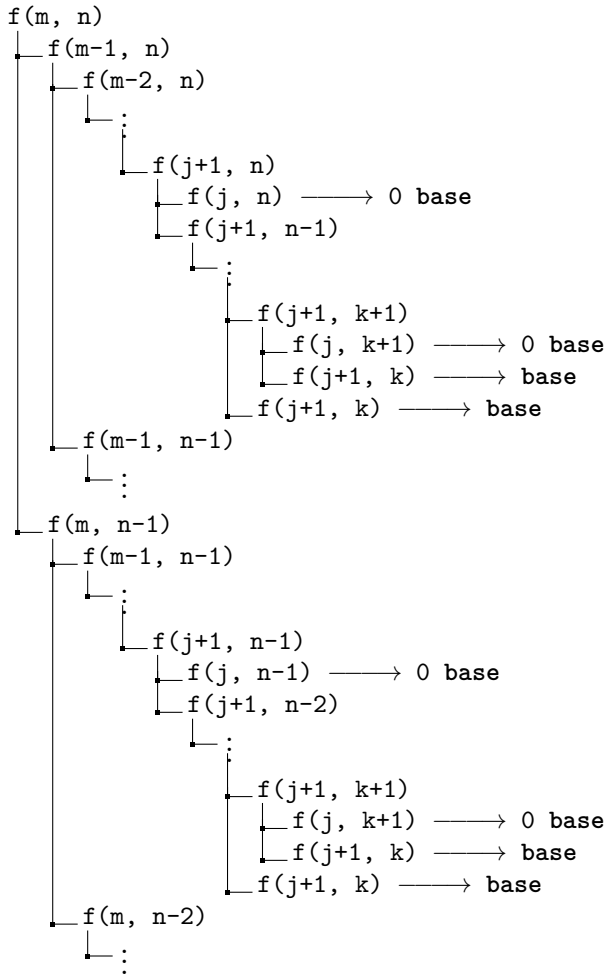


Figure 1.2 – Structure arborescente de la fonction

Complexité dans le temps

La complexité temporelle de cette fonction est *exponentielle*, puisqu'à chaque appel récursif, deux nouveaux appels sont engendrés. Par ailleurs, chaque argument m et n peut augmenter la complexité. Nous avons donc :

$$O(m + n)^2$$

Note :

L'arbre est **symétrique** :

$$\forall n, m \in \mathbb{N}, f(m, n) = f(n, m)$$

Nous avons d'ailleurs montré que $f(3, 2) = f(2, 3)$

Complexité dans l'espace

Chaque appel récursif est additionné dans la **pile d'appels** qui emmagasine en mémoire quelles fonctions parents sont appelées par les fonction enfants. La complexité dans l'espace correspond donc à la hauteur de la pile d'appel qui est elle-même liée à la profondeur maximale de l'arbre.

La profondeur maximale de l'arbre correspond au chemin le plus long de la racine de l'arbre d'appels (lorsque $f(m, n)$ est appelé pour la première fois) jusqu'à une feuille de l'arbre (lorsque une condition de base est atteinte). Dans ce cas, la profondeur maximale est atteinte lorsqu'on suit toujours le chemin $f(m-1, n)$ ou $f(m, n-1)$ jusqu'à ce que m ou n atteigne 0. Si $m \neq n$, l'une des deux direction sera plus longue. Il faut donc considérer $\max(m, n)$.

On peut alors conclure que la complexité en espace de cet algorithme est :

$$O(\max(m, n))$$

```

1 public class MistFunction2 {
2     public static List<List<String>> mistFunction2(String target, List<String> pieces) {
3         List<List<String>>[] table = new ArrayList[target.length() + 1];
4         for (int i = 0; i <= target.length(); i++) {
5             table[i] = new ArrayList<>();
6         }
7         table[0].add(new ArrayList<>());
8
9         for (int i = 0; i < target.length(); i++) {
10             for (String piece : pieces) {
11                 if (i + piece.length() <= target.length() &&
12                     target.startsWith(piece, i)) {
13                     List<List<String>> newCombinations = new ArrayList<>();
14                     for (List<String> subarray : table[i]) {
15                         List<String> newSubarray = new ArrayList<>(subarray);
16                         newSubarray.add(piece);
17                         newCombinations.add(newSubarray);
18                     }
19                     table[i + piece.length()].addAll(newCombinations);
20                 }
21             }
22         }
23
24         return table[target.length()];
25     }
26 }
27
28

```

Raison d'être L'algorithme semble compter le nombre de façons possibles de se déplacer dans une grille $m \times n$ avec comme contrainte de se déplacer uniquement vers le bas (m) ou uniquement vers la droite (n)

1.2 MistFonction2

Raison d'être Soit une chaîne de caractères `target` et une liste `pieces` dont chaque élément est un caractère de la chaîne `target`, la fonction engendre une liste de listes de chaînes de caractère. Cette liste correspond à *toutes les combinaisons possibles* de caractères permettant d'obtenir la chaîne originale.

Exemple 1

Soit `target = "abc"` et `piece = ["a", "b", "c"]`, on obtient alors le résultat suivant :

```

> table = [
  [], ["a"], ["ab"], ["a", "b"],
  ["a", "bc"], ["ab", "c"],
  ["a", "b", "c"]]
> table[target.length()] = ["a", "bc"], ["ab", "c"], ["a", "b", "c"]]
> f(target, pieces) = ["a", "bc"], ["ab", "c"], ["a", "b", "c"]]

```

Complexité dans le temps La fonction possède deux boucles imbriquées de complexité proportionnelle à `target.length()`. En effet, on suppose que `pieces` est une partition de `target` tel que :

$$\text{target.length}() = \text{pieces.length}$$

Dans la seconde boucle, on vérifie si une pièce, disons de longueur k , est le début d'une sous-chaîne. En supposant que k est de longueur 1, cette opération est à temps constant. Autrement, elle dépendrait de la longueur de k .

Finalement, pour chaque combinaison existante à la position i dans `table`, la fonction tente de l'étendre en ajoutant une nouvelle pièce valide. Cette opération de génération de combinaison est dépendante du nombre de combinaisons précédentes C_i ,

qui peuvent croître de manière significative à chaque étape. Ainsi, l'opération de génération de combinaison a une complexité temporelle de $O(n \cdot C)$, où n est la longueur de la cible et C est le nombre total de combinaisons uniques à l'indice i . Si chaque **piece** peut être utilisée une seule fois et correspond à un caractère unique dans **target**, le nombre de combinaisons C_i à chaque indice reste constant, simplifiant la complexité globale de la fonction à $O(n^2)$. Cependant, dans des cas où des caractères se répètent et peuvent être combinés de différentes manières, C peut croître exponentiellement, augmentant ainsi la complexité temporelle.

Complexité dans l'espace La complexité en espace totale est la somme de la mémoire requise pour le tableau **table** et les combinaisons qu'il contient. Dans le pire cas, avec un grand nombre de combinaisons, elle peut devenir exponentielle, notée $O(2^n)$. Toutefois, si chaque **piece** est unique et ne peut être utilisée qu'une seule fois, la complexité est réduite à $O(n^2)$, car chaque indice contiendrait au plus une combinaison de n caractères, et il y a $n+1$ indices.

1.3 MistFonction3

```

1 public class MistFunction3 {
2     public static boolean mistFunction3(int target, int[] options) {
3         boolean[] table = new boolean[target + 1];
4         table[0] = true;
5
6         for (int i = 0; i <= target; i++) {
7             if (table[i]) {
8                 for (int option : options) {
9                     if (i + option <= target) {
10                         table[i + option] = true;
11                     }
12                 }
13             }
14         }
15         return table[target];
16     }
17 }
18

```

Raison d'être La fonction vérifie s'il est possible d'additionner une combinaison d'entiers présents dans le tableau d'options afin d'obtenir le nombre **target**.

Complexité dans le temps La complexité temporelle est déterminée par les deux boucles imbriquées itératives. La boucle extérieure s'exécute une seule fois de 0 à **target**. Plus l'entier **target** est grand, plus grand sera le nombre d'itérations. Pour cette portion, nous obtenons alors une complexité de $O(n)$ où n est la valeur de **target**. Par ailleurs, pour chaque itération de la boucle extérieure, nous effectuons m itérations de la boucle intérieure où m est le nombre de possibilités dans le tableau **options**. Nous avons donc une complexité :

$$O(n \cdot m)$$

Or, il est possible que le nombre d'options fournies soit considérablement inférieure à la valeur de **target**. Dans ce cas, la complexité sera quasi linéaire.

$$O(n \cdot m) \xrightarrow{m \rightarrow 1} O(n)$$

Complexité dans l'espace L'opération la plus déterminante pour la complexité spatiale est la création du tableau **table**. Ce tableau contient une entrée de plus que la taille de l'entier **target**. Ainsi, plus l'entier **target** est grand, plus grand sera le tableau **table** et plus grande sera la consommation de mémoire. La complexité spatiale évolue donc linéairement avec n où n est la valeur de **target** :

$$O(n)$$

1.4 Approche récursive

1.4.1 Analyse de MistFonction3

Pour trouver toutes les combinaisons possibles de **target** en utilisant les pièces **pieces**, chaque appel récursif devra tenter de reconstituer **target**. Pour cela, chaque combinaison sera recalculée plusieurs fois, puisque pour un caractère donnée de **target**,

on épuise les possibilités de `pieces`. Si `target` a une longueur n et `pieces` a une longueur m , la complexité temporelle sera alors :

$$O(m^n)$$

1.4.2 Analyse de `MistFonction3`

Pour vérifier *récurivement* si une combinaison d'éléments dans un tableau d'options `options` permet d'engendrer l'entier `target`, il faudrait évaluer récursivement toutes les combinaisons possibles. Pour chaque élément de `options`, deux possibilités s'offre à nous : inclure l'entrée à l'index courant dans la somme ou ne pas l'inclure. Cela génère 2^m appels récursifs où m est la quantité d'éléments dans le tableau original `options`. La complexité serait donc :

$$O(2^m)$$

Pour un `target` `T`, un tableau d'options `[A, B, C]` (`A, B` et `C` sont des entiers et peuvent prendre n'importe quelle valeur pour cet exemple) et un index i , nous avons les appels récursifs suivants.

```
f(T, [A, B, C], 0)
├─ Inclure A: f(T-A, [A, B, C], 1)
│   └─ Inclure B: f(T-A-B, [A, B, C], 2)
│       ├── Inclure C: f(T-A-B-C, [A, B, C], 3)
│       └─ Exclure C: f(T-A-B, [A, B, C], 3)
│   └─ Exclure B: f(T-A, [A, B, C], 2)
│       ├── Inclure C: f(T-A-C, [A, B, C], 3)
│       └─ Exclure C: f(T-A, [A, B, C], 3)
└─ Exclure A: f(T, [A, B, C], 1)
    ├── Inclure B: f(T-B, [A, B, C], 2)
    │   ├── Inclure C: f(T-B-C, [A, B, C], 3)
    │   └─ Exclure C: f(T-B, [A, B, C], 3)
    └─ Exclure B: f(T, [A, B, C], 2)
        ├── Inclure C: f(T-C, [A, B, C], 3)
        └─ Exclure C: f(T, [A, B, C], 3)
```

Pile simple

Exercice 2 (3 pts)

Écrire une classe « `ArrayStack` » qui fonctionne à l'aide d'un tableau générique (n'oubliez pas que vous devez également implémenter son interface appelée « `Stack` »). Le nombre maximal d'éléments est de 100. Ce sont les méthodes que la pile simple devrait pouvoir exécuter :

Fonction	Signature	Détails
Push	<code>public void push(E e)</code>	Ajoute un élément sur la pile
Pop	<code>public E pop()</code>	Retire le dernier élément sur la pile et le renvoie
Top	<code>public E top()</code>	Renvoie le dernier élément sur la pile
Size	<code>public int size()</code>	Renvoie la longueur de la pile
Is Empty	<code>public boolean isEmpty()</code>	Vérifie si la pile est vide
To String	<code>public String toString()</code>	Produit une représentation en chaîne des éléments de la pile classés de haut en bas.

```

1 public class ArrayStack<E> implements Stack<E> {
2     private static final int NombreElementsMaximal = 100;
3
4     private E[] Pile; // utilisation d'un simple tableau pour l'implementation
5
6     /* Le haut de la pile a comme index -1 lorsque le tableau est vide
7      0 lorsque le tableau a un element, 1 lorsque le tableau a 2 element,
8      etc. */
9     private int dessus = -1;
10
11     // Constructeur
12     public ArrayStack() {
13         Pile = (E[]) new Object[NombreElementsMaximal];
14         // safe cast; compiler may give warning
15     }
16
17     // Implemente les methodes de l'interface Stack.
18     @Override
19     public void push(E e) throws IllegalStateException {
20         if (size() == NombreElementsMaximal) throw
21             new IllegalStateException("La pile a atteint le nombre maximal d'éléments");
22
23         //incrémente l'index de dessus de pile
24         this.dessus++;
25         Pile[dessus] = e; // increment t before storing new item
26     }
27
28     @Override
29     public E pop() {
30         if (isEmpty()) return null;
31         E elementRetourne = Pile[dessus];
32         //Efface artificiellement l'element
33         Pile[dessus] = null;
34         // Decrémente pour avoir un index de dessus valide
35         dessus--;
36         return elementRetourne;
37     }
38
39     @Override
40     public E top() {
41         if (isEmpty()) return null;
42         return Pile[dessus];
43     }
44
45     @Override
46     public int size() {
47         return (dessus + 1);
48     }
49
50     @Override
51     public boolean isEmpty() {
52         // Lorsque le dessus a l'index 1, on sait que la pile est vide
53         return (dessus == -1);
54     }
55 }

```



```

56  @Override
57  public boolean isFull() {
58      // Lorsque la pile est pleine, le dessus a l'index 99
59      return (dessus == 99);
60  }
61
62  @Override
63  public String toString() {
64      StringBuilder sb = new StringBuilder("Contenu de la pile: ");
65      for (int i = dessus; i >= 0; i--) {
66          sb.append(Pile[i]);
67          if (i > 0) sb.append(", ");
68      }
69      sb.append("]");
70      return sb.toString();
71  }
72
73  }
74
75  public interface Stack<E> {
76      void push(E e);
77      E pop();
78      E top();
79      int size();
80      boolean isEmpty();
81      String toString();
82      }
83

```

Pile Double

Exercice 3 (4 pts)

Écrire une classe « `ArrayDoubleStack` » qui implémente 2 piles dans un même tableau générique (n'oubliez pas que vous devez également implémenter son interface appelée « `DoubleStack` »). Le nombre maximal d'éléments (longueur pile 1 + longueur pile 2) est de 100. Cette classe doit avoir toutes les fonctions ci-dessus pour chaque pile, avec pour seule différence un booléen « `one` » qui indique si l'on traite les éléments à la 1re ou 2e pile.

Fonction	Signature	Détails
Push	<code>public boolean push(boolean one, E e)</code>	Ajoute un élément sur la pile et renvoie vrai. Renvoie faux si ce n'est pas possible.
Pop	<code>public E pop(boolean one)</code>	Retire le dernier élément sur la pile et le renvoie.
Top	<code>public E top(boolean one)</code>	Renvoie le dernier élément sur la pile.
Size	<code>public int size(boolean one)</code>	Renvoie la longueur de la pile.
Is Full	<code>public boolean isFull()</code>	Vérifie si la pile est pleine.
Print	<code>public void print()</code>	Imprime le contenu des 2 piles.

```

1 interface DoubleStack<E> {
2     boolean push(boolean one, E element);
3     E pop(boolean one);
4     E top(boolean one);
5     int size(boolean one);
6     boolean isFull();
7     void print();
8 }
9 public class ArrayDoubleStack<E> implements DoubleStack<E> {
10     private E[] doublePile;
11     private int nombreMaximalElements = 100;
12     private int dessus1;
13     private int dessus2;
14
15     public ArrayDoubleStack() {
16         doublePile = (E[]) new Object[nombreMaximalElements];
17         // Le dessus de la pile 1 se trouve du cote "gauche" du tableau
18         dessus1 = -1;
19
20         // Le dessus de la pile 2 se trouve du cote "droit" du tableau
21         dessus2 = nombreMaximalElements;
22     }
23
24     /* La pile est pleine lorsque l'index de pile 1 est -1 (index initial) + 50 + 1= 50 et
25      * l'index de pile 2 est 100 (index initial) - 50 = 50*/
26     public boolean isFull() {
27         return (dessus1 + 1 == dessus2);
28     }
29
30     /* La methode verifie d'abord si la double pile est pleine et ajoute
31      * l'element au dessus du bon cote de la doublePile en fonction
32      * de l'argument fourni (true ou false) */
33     public boolean push(boolean one, E element) {
34         if (isFull()) {
35             return false;
36         }
37         if (one) {
38             /* On increment le dessus de pile 1 et met
39              * l'element au 'nouveau' dessus1 */
40             doublePile[++dessus1] = element;
41         } else {
42             /* Ou alors On decrement le dessus de pile 2 et met
43              * l'element au 'nouveau' dessus2 */
44             doublePile[--dessus2] = element;
45         }
46         return true;
47     }
48
49     public E pop(boolean one) {
50         if (one) {
51             // Verifie que la pile contient au moin un element
52             if (dessus1 >= 0) {
53                 /* Decrement l'index de dessus, efface le contenu de dessus
54                  et retourne la valeur qui etait au dessus1 */
55                 E reponse = doublePile[dessus1];
56                 doublePile[dessus1] = null;
57                 dessus1--;
58                 return reponse;
59             }
60         } else if (!one && dessus2 < nombreMaximalElements) {
61             E reponse = doublePile[dessus2];
62             doublePile[dessus2] = null;
63             dessus2++;
64             return reponse;
65         }
66         else {
67             // Si aucune des pile ne contient d'element
68             throw new IllegalStateException(
69                 "La double pile est vide, vous ne pouvez pas obtenir d'element. ")
70         }
71     }

```

```

72 }
73
74 /* En fonction du boolean fourni, on verifie si la pile
75 * est plein et retourne l'element du dessus1 ou dessus2 */
76 public E top(boolean one) {
77     if (one) {
78         if (dessus1 >= 0) {
79             return doublePile[dessus1];
80         }
81     } else if (!one && dessus2 < nombreMaximalElements) {
82         return doublePile[dessus2];
83     }
84     else {
85         throw new IllegalStateException(
86             "La double pile est vide, vous ne pouvez pas regarder l'element. ")
87     }
88 }
89
90 /* En fonction du boolean fourni, on retourne la valeur
91 * dessus1 + 1 (taille pile 1) ou nbMaximalElements - dessus (taille pile 2) */
92 public int size(boolean one) {
93     if (one) {
94         return dessus1 + 1;
95     } else {
96         return nombreMaximalElements - dessus2;
97     }
98 }
99
100
101 public void print() {
102     System.out.println("Pile 1:");
103     for (int i = 0; i <= dessus1; i++) {
104         System.out.println(doublePile[i]);
105     }
106     System.out.println("Pile 2:");
107     for (int i = nombreMaximalElements - 1; i >= dessus2; i--) {
108         System.out.println(doublePile[i]);
109     }
110 }
111 }
112

```

Pile spéciale

Exercice 4 (3 pts)

Écrire une classe « SpecialArrayStack » qui fonctionne à l'aide d'un tableau générique (N'oubliez pas que vous devez également implémenter son interface appelée « SpecialStack »). Le nombre maximal d'éléments est de 100. La pile spéciale devrait implémenter une méthode getMax() qui devrait renvoyer l'élément maximum stocké dans la pile spéciale en $O(1)$ temps et $O(1)$ espace supplémentaire, en plus des mêmes méthodes qui ont été implémentées dans « Pile simple ».

```

1 /* Seules les classes qui extends l'interface Java Comparable peuvent être stockées dans
2 * la SpecialArrayStack. Notre methode push(E e) a besoin de comparer l'element qu'on est sur
3 * le point d'ajouter (e) a l'element Maximal Maximaux[prochainMaximum].
4 * Puisqu'on ne connaît pas a l'avance le type des objets stockés dans la
5 * SpecialArrayStack, il faut utiliser le type generique E */
6 public class SpecialArrayStack<E extends Comparable<E>> implements Stack<E> {

```

```

7
8 private static final int NombreElementsMaximal = 100;
9 private E[] Pile;
10 private E[] Maximaux; // Tableau pour stocker les éléments maximaux
11 private int dessus = -1;
12 private int prochainMaximum = -1; // Index pour suivre le prochain élément maximal
13
14
15 // Constructeur
16 public SpecialArrayStack() {
17     Pile = (E[]) new Object[NombreElementsMaximal];
18     // Initialisation du tableau des maximaux
19     Maximaux = (E[]) new Object[NombreElementsMaximal];
20 }
21
22 // Implémente les méthodes de l'interface Stack...
23 @Override
24 public void push(E e) throws IllegalStateException {
25     if (isFull()) {
26         throw new
27             IllegalStateException("La pile a atteint le nombre maximal d'éléments");
28     }
29
30     if (isEmpty() || e.compareTo(getMax()) >= 0) {
31         prochainMaximum++;
32         // Ajoute le nouveau maximum au tableau des maximaux
33         Maximaux[prochainMaximum] = e;
34     }
35     dessus++;
36     Pile[dessus] = e; // Ajoute l'élément à la pile
37 }
38
39 @Override
40 public E pop() {
41     if (isEmpty()) {
42         throw new
43             IllegalStateException("La pile est vide, vous ne pouvez pas retirer d'élément.");
44     }
45
46     E elementRetourne = Pile[dessus];
47     Pile[dessus] = null; // Efface artificiellement l'élément
48     dessus--;
49
50     if (elementRetourne.equals(getMax())) {
51         // Retire le maximum actuel du tableau des maximaux
52         Maximaux[prochainMaximum] = null;
53         prochainMaximum--;
54     }
55
56     return elementRetourne;
57 }
58
59 @Override
60 public E top() {
61     if (isEmpty()) return null;
62     return Pile[dessus];
63 }
64
65 @Override
66 public int size() {
67     return (dessus + 1);
68 }
69
70 @Override
71 public boolean isEmpty() {
72     // Lorsque le dessus a l'index 1, on sait que la pile est vide
73     return (dessus == -1);
74 }
75
76 @Override
77 public boolean isFull() {

```

```

78         return (dessus == NombreElementsMaximal - 1);
79     }
80
81
82     @Override
83     public String toString() {
84         StringBuilder sb = new StringBuilder("Contenu de la pile: ");
85         for (int i = dessus; i >= 0; i--) {
86             sb.append(Pile[i]);
87             if (i > 0) sb.append(", ");
88         }
89         sb.append("]");
90         return sb.toString();
91     }
92
93     public E getMax() {
94         if (prochainMaximum == -1) {
95             throw new IllegalStateException("La pile est vide, il n'y a pas d'Élément maximal.");
96         }
97         return Maximaux[prochainMaximum];
98     }
99 }
100
101

```