

Structures de Données  
IFT2015  
**Devoir 1**

Franz Girardin

27 janvier 2024

# Table des matières

2	CHAPITRE 1	
	Analyse de complexité	
1.1	MistFonction1	2
1.2	MistFunction2	3

# 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.

### 1.1 MISTFUNCTION1

```
1 public class MistFunction1{
2
3     public static int mistFunction1(int m, int n) {
4         if (m == 1) && (n == 1) return 1;
5         if (m == 0) || (n == 0) return 0
6         return mistFunction1(m - 1, n) + mistFunction1(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.

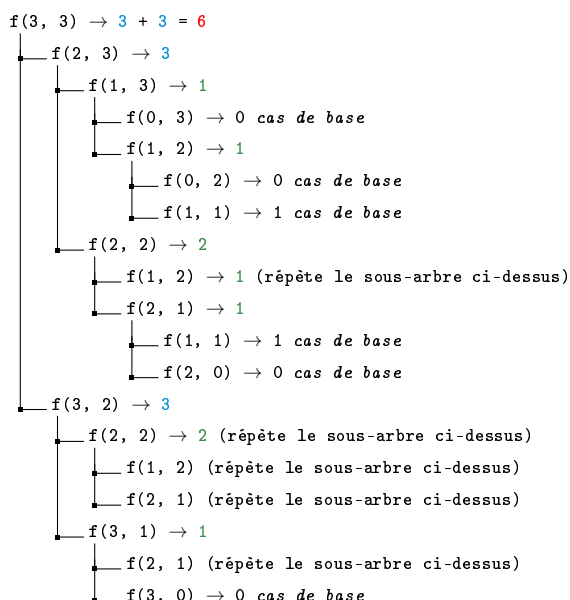


FIGURE 1.1 – Exemple de  $f(3,3)$

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.

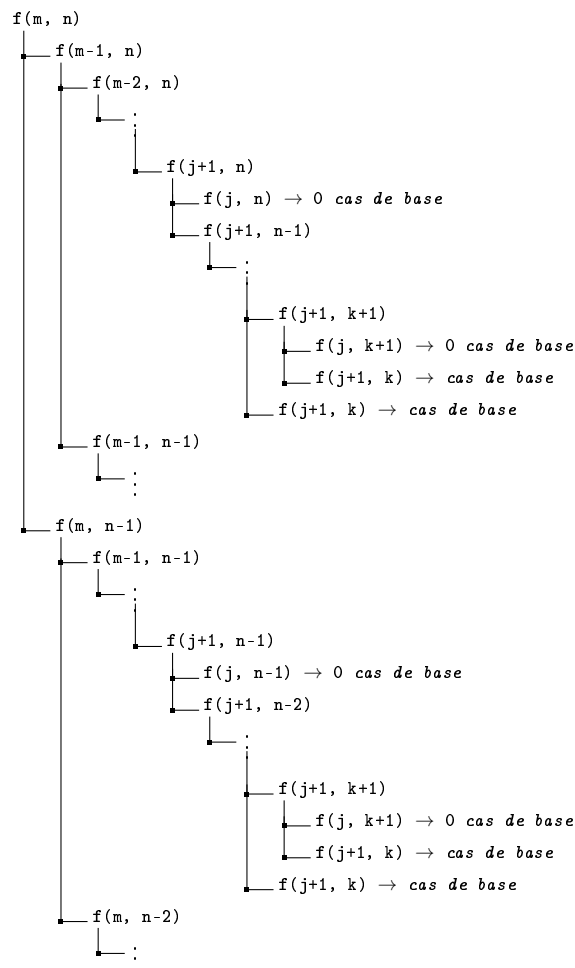


FIGURE 1.2 – Structure arborescente de la fonction

## 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  $f(3,2) = f(2,3)$

```

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

```

## Complexité dans l'espace

Chaque appel récuratif 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 :

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

**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 MISTFUNCTION2

**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 partitions 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 pièce 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.