Structures de Données
IFT2015
Devoir 1

Franz Girardin

27 janvier 2024

Table des matières

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 MistFonction1

```java
public class MistFonction1{

  public static int mistFonction1(int m, int n) {
      if (m == 1) && (n == 1) return 1;
      if (m == 0) || (n == 0) return 0
      return mistFonction1(m -1, n) + mistFonction1(m, n -1)
  }
}
```

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 valur 0.

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

```
f(3, 3) → 3 + 3 = 6
├── f(2, 3) → 3
│   ├── f(1, 3) → 1
│   │   ├── f(0, 3) → 0 cas de base
│   │   └── f(1, 2) → 1
│   │       ├── f(0, 2) → 0 cas de base
│   │       └── f(1, 1) → 1 cas de base
│   └── f(2, 2) → 2
│       ├── f(1, 2) → 1 (répète le sous-arbre ci-dessus)
│       └── f(2, 1) → 1
│           ├── f(1, 1) → 1 cas de base
│           └── f(2, 0) → 0 cas de base
└── f(3, 2) → 3
    ├── f(2, 2) → 2 (répète le sous-arbre ci-dessus)
    │   ├── f(1, 2) (répète le sous-arbre ci-dessus)
    │   └── f(2, 1) (répète le sous-arbre ci-dessus)
    └── f(3, 1) → 1
        ├── f(2, 1) (répète le sous-arbre ci-dessus)
        └── f(3, 0) → 0 cas de base
```

FIGURE 1.1 – Exemple de $f(3,3)$

Ainsi, nous pouvons représenter les appels récursifs par une 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.

```
f(m, n)
├── f(m-1, n)
│   ├── f(m-2, n)
│   │   └── ⋮
│   │       └── f(j+1, n)
│   │           ├── f(j, n) → 0 cas de base
│   │           └── f(j+1, n-1)
│   │               └── ⋮
│   │                   └── f(j+1, k+1)
│   │                   │   ├── f(j, k+1) → 0 cas de base
│   │                   │   └── f(j+1, k) → cas de base
│   │                   └── f(j+1, k) → cas de base
│   └── f(m-1, n-1)
│       └── ⋮
└── f(m, n-1)
    ├── f(m-1, n-1)
    │   └── ⋮
    │       └── f(j+1, n-1)
    │           ├── f(j, n-1) → 0 cas de base
    │           └── f(j+1, n-2)
    │               └── ⋮
    │                   └── f(j+1, k+1)
    │                   │   ├── f(j, k+1) → 0 cas de base
    │                   │   └── f(j+1, k) → cas de base
    │                   └── f(j+1, k) → cas de base
    └── f(m, n-2)
        └── ⋮
```

FIGURE 1.2 – Stucture arboressante 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'ailleur montré que $f(3,2) = f(2,3)$

```java
public class MistFunction2 {
    public static List<List<String>> mistFunction2(String target, List<String> pieces) {
        List<List<String>>[] table = new ArrayList[target.length() + 1];
        for (int i = 0; i <= target.length(); i++) {
            table[i] = new ArrayList<>();
        }
        table[0].add(new ArrayList<>());

        for (int i = 0; i < target.length(); i++) {
            for (String piece : pieces) {
                if (i + piece.length() <= target.length() &&
                    target.startsWith(piece, i)) {
                    List<List<String>> newCombinations = new ArrayList<>();
                    for (List<String> subarray : table[i]) {
                        List<String> newSubarray = new ArrayList<>(subarray);
                        newSubarray.add(piece);
                        newCombinations.add(newSubarray);
                    }
                    table[i + piece.length()].addAll(newCombinations);
                }
            }
        }

        return table[target.length()];
    }
}


```

Complexité dans l'espace
Chaque appel résursif 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 hateur 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 dêtre 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 drotie (n)

1.2 MistFunction2

Raison d'être Soit une chaine 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 charactère. Cette liste correspond à *toutes les combinaisons possibles* de caractères permmetant 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 fonctions possède deux boucles imbriquées de complexitée proportionnelle à `target.lenght()`. En effet,on suppose que `pieces` est une partitions de `target` tel que :

`target.lenght() = pieces.lenght`

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 `piece` 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 l a 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.