Par Yamir Alejandro Poldo Silva et Luis Eche Guzman

Analyse de complexité

1) MisFonction1

```
public class MisFonction1 {

    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 )
    }
}
```

Que fait la fonction?????

La fonction misFonction1 prend en parametres 2 nombres (int) m et n.

Elle traite 2 cas de base tel que:

- si m et n chacun égal 1, retourne 1
- si m ou n égale 0, retourne 0

Si aucun de ces cas est atteint, la fonction fait un double appel récursif où dans l'un m décrement et dans l'autre n décrement, jusqu'à ce soit m ou n arrive jusqu'à 1 ou 0.

Compte tenu que chaque fois qu'il y a un appel, il y a 2 nouvelles instances de la fonction qui sont crées le nombre d'appels augmente de manière exponentielle dépedemment du nombre de décrement qu'il faut faire pour chaque m et n afin d'atteindre 1 ou 0. Donc, la complexité de la fonction peut-être qualifiée tel que O(2^(m+n)).

2) MisFunction2

```
public class MisFunction2 {
    public static List<List<String>> mistFonction2(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()];
    }
}
```

Que fait la fonction ?????

La fonction mistFonction2 prend en parametres un String "target" et une liste de String "pieces".

Dans un premier temps, la fonction initialise un Array "table" et initiliase chaque élément de table avec une boucle for où la complexité est de O(target.lenght), donc O(n). De plus, la premiere valeur de ce array est initialisé à un element vide.

Ensuite, il y a une série de boucles for imbriquées. La premiere boucle imbriquée itère aussi target.lenght, donc on a une complexité de O(n)

La seconde boucle for itère sur chaque item de la liste de String "pieces", donc on se retrouve avec O(pieces.lenght), autrement 0(m).

La seconde boucle imbriquée itère sur la longeueur de table[i], donc la longeur de target.lenght, donc se retrouve encore a iterer sur O(n).

Ainsi, la complexité de la fonction est O(n*m)

3) MisFonction3

```
public class MisFonction3 {
    public static boolean mistFunction3(int target, int[] options){
        boolean[] table = new boolean[target + 1];
        table[0] = true;

        for (int i = 0; i <= target; i++){
            if (table[i]){
                for (int option : options){
                    if (i + options <= target) {
                        table[i + option] = true;

                    }
                }
```

```
                }
            }
            return table[target];
        }
    }
```

Que fait la fonction?????????

La fonction mistFunction3 prend un nombre target et un array de nombre options en parametres.

Un tableau de boolean est initiés de longeur target + 1 et la premier valeur est initiée à true.

Ensuite, il y a 2 boucles for imbriquées. La premiere boucle itere de 0 jusqu'à target. Si i est true, on itère sur la seconde boucle. La seconde boucle itere sur la longeur du array options.

Compte tenu que toutes les valeurs de table pourrait être true, alors il est possible d'iterer sur toute la seconde boucle dans le pire cas.

Les autres opérations à l'intérieur de la boucle sont O(1) Ainsi, on a O(target * array.lenght) tel que O(n*m).

4)

Pour MistFunction2,

Pour MistFunction3, on peut iterer sur options et faire un appel recursif avec un nombre réduit de options. Compte tenu que le nombre d'appels recursifs va dépendre de la longeur de n et que le nombre d'itération doubles à chaque appels récursifs, on a O(2^n).

Piles

1)

```
public class MovingDay {

    private static int dayCounter = 0;

    public static void moveCity(ArrayStack<CityLevel> source,
ArrayStack<CityLevel> destination, ArrayStack<CityLevel> middleCity) {
        if (source.isEmpty()) {
            throw new IllegalStateException("Source city is empty, no
levels to move.");
        }

        // O(n)
        while (!source.isEmpty()) {
            CityLevel levelToMove = source.pop();
            middleCity.push(levelToMove);
            MovingStatus.printCurrentState(dayCounter++, source,
destination, middleCity);
```

```
        }

        //O(n)
        while (!middleCity.isEmpty()) {

            CityLevel levelInMiddle = middleCity.pop();
            destination.push(levelInMiddle);

            MovingStatus.printCurrentState(dayCounter++, source,
destination, middleCity);

        }
        MovingStatus.isMovePossible(dayCounter <= 10, dayCounter);


    }
}
```

La fonction moveCity de la classe MovingDay prend en paramètres 3 ArrayStack, pour source, destination et middleCity.

Dans un premier temps, on verifie si source est vide.

Ensuite, il y a 2 boucles while qui itèrent sur la longueur de source et middleCity. Ainsi, ces 2 boucles ont une complexité de O(n). Les opérations à l'intérieure des boucles sont de O(1), donc elles n'ont aucun impact sur la complexité de la fonction.

Ainsi, MovingDay a une complexité générale de O(n).

2)

```
public class DuplicateEater {

    public void pairDestroyer(String[] array) {

        ArrayStack<String> pairDestroyerStack = new ArrayStack<>(100);

        for (String item : array) {
            if (!pairDestroyerStack.isEmpty() &&
pairDestroyerStack.top().equals(item)) {
                pairDestroyerStack.pop();
            }
            else{
                pairDestroyerStack.push(item);
            }
        }

        // on peut l'enlever après test
        //System.out.println(pairDestroyerStack.toString());
```

```
            System.out.println(pairDestroyerStack.size());
        }
    }
```

La fonction paurDestroyer de la classe DuplicateEater prend en paramètres un tableau génerique de String.

On initialise un ArrayStack. Ensuite, il y a une boucle for qui itère sur la longueur du tableau en paramètres, donc on a une complexité de O(n). Compte tenu, que ArrayStack<>.pop() et ArrayStack<>.push(e) sont de complexité O(1), alors la complexité générale de la fonction est de O(n).