Université de Montréal

Département d'Informatique et de Recherche Opérationnelle

Structures de données

Analyse de Complexité et Piles

Auteur
Franz Girardin

Matricule
20078678

3 février 2024

Table des matières

Section 1

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.

```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 base
│ │   └─f(1, 1) ───→ 1 base
│ └─f(2, 2) ───→ 2
│   ├─f(1, 2) ───→ 1 (répète le sous-arbre ci-dessus)
│   └─f(2, 1) ───→ 1
│     ├─f(1, 1) ───→ 1 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
  │ └─f(2, 1) répète
  └─f(3, 1) ───→ 1
    ├─f(2, 1) répète
    └─f(3, 0) ───→ 0 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 base
│  │        └─ f(j+1, n-1)
│  │           └─ ⋮
│  │              ├─ f(j+1, k+1)
│  │              │  ├─ f(j, k+1) ───→ 0 base
│  │              │  └─ f(j+1, k) ───→ base
│  │              └─ f(j+1, k) ───→ 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 base
   │        └─ f(j+1, n-2)
   │           └─ ⋮
   │              ├─ f(j+1, k+1)
   │              │  ├─ f(j, k+1) ───→ 0 base
   │              │  └─ f(j+1, k) ───→ base
   │              └─ f(j+1, k) ───→ 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 $f(3,2) = f(2,3)$

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 :

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

```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()];
    }
}
```

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 `MistFonction2`

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 :

$$\texttt{target.lenght() = pieces.lenght}$$

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.

1.3 `MistFonction3`

```java
public class MistFunction3 {
    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 + option <= target) {
                        table[i + option] = true;
                    }
                }
            }
        }
        return table[target];
    }
}
```

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

Complexité dans le temps La complexité temporelle est déterminée par les deux boucles imbriquées itératives. La boucle extérireure 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 un complexité de $O(n)$ où n est la valur de `target`. Par ailleurs, pour chaque itération de la boucle extérireure, nous effectuons m itérations de la boucles intérieure où m est le nombre de possibilité 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.

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

Complexité dans l'espace L'opération la plus déterminante pour la complexité spaciale 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 spaciale évolue donc linéairement avec n où n est la valeur de `target` :

$$O(n)$$

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

$$O(m^n)$$

1.4.2 Analyse de `MistFonction3`

Pour vérifier *récursivement* 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 les tableau original `options`. La complexité serait donc :

$$O(2^m)$$

Pour une `target T`, un tableau d'options `[A, B, C]` (`A`, `B` t `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)
```

Section 2

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

```java
public class ArrayStack<E> implements Stack<E> {
  private static final int NombreElementsMaximal = 100;

    private E[] Pile; // utilisation d’un simple tableau pour l’implementation

    /* Le haut de la pile a comme index -1 lorsque le tableau  est vide
       0 lorsque le tableau a un element, 1 lorsque le tableau a 2 element,
       etc. */
    private int dessus = -1;

    // Constructeur
    public ArrayStack() {
        Pile = (E[]) new Object[NombreElementsMaximal];
        // safe cast; compiler may give warning
    }

    // Implemente les methodes de l’infercae Stack.
    @Override
    public void push(E e) throws IllegalStateException {
        if (size() == NombreElementsMaximal) throw
          new IllegalStateException("La pile a atteint le nombre maximal d’Ã©lÃ©ments");

        //incremente l’index de dessus de pile
        this.dessus++;
        Pile[dessus] = e; // increment t before storing new item
    }

    @Override
    public E pop() {
        if (isEmpty()) return null;
        E elementRetourne = Pile[dessus];
        //Efface artificiellement l’element
        Pile[dessus] = null;
        // Decremente pour avoir un index de dessus valide
        dessus--;
        return elementRetourne;
    }

    @Override
    public E top() {
        if (isEmpty()) return null;
        return Pile[dessus];
    }

    @Override
    public int size() {
        return (dessus + 1);
    }

    @Override
    public boolean isEmpty() {
      // Lorsque le dessus a l’index 1, on sait que la pile est vide
        return (dessus == -1);
    }

```

```java
    @Override
    public boolean isFull() {
       // Lorsque la pile est pleine, le dessus a l'index 99
      return (dessus == 99);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("Contenu de la pile: [");
        for (int i = dessus; i >= 0; i--) {
            sb.append(Pile[i]);
            if (i > 0) sb.append(", ");
        }
        sb.append("]");
        return sb.toString();
    }

}

public interface Stack<E> {
    void push(E e);
    E pop();
    E top();
    int size();
    boolean isEmpty();
    String toString();
        }

```

Section 3

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

```java
 interface DoubleStack<E> {
    boolean push(boolean one, E element);
    E pop(boolean one);
    E top(boolean one);
    int size(boolean one);
    boolean isFull();
    void print();
}
public class ArrayDoubleStack<E> implements DoubleStack<E> {
    private E[] doublePile;
    private int nombreMaximalElements = 100;
    private int dessus1;
    private int dessus2;

    public ArrayDoubleStack() {
        doublePile = (E[]) new Object[nombreMaximalElements];
        // Le dessus de la pile 1 se trouve du cote "gauche" du tableau
        dessus1 = -1;

        // Le dessus de la pile 2 se trouve du cote "droit" du tableau
        dessus2 = nombreMaximalElements;
    }

    /* La pile est pleine lorsque l'index de pile 1 est -1 (index initial) + 50 + 1= 50 et
     * l'index de pile 2 est 100 (index initial) - 50 = 50*/
    public boolean isFull() {
        return (dessus1 + 1 == dessus2);
    }

    /* La methode verifie d'abord si la double pile est pleine et ajoute
     * l'element au dessus du bon cote de la doublePile en fonction
     * de l'argument fourni (true ou false) */
    public boolean push(boolean one, E element) {
        if (isFull()) {
            return false;
        }
        if (one) {
            /* On increment le dessus de pile 1 et met
             * l'element au 'nouveau' desuss1 */
              doublePile[++dessus1] = element;
        } else {
            /* Ou alors On decrement le dessus de pile 2 et met
             * l'element au 'nouveau' desuss2 */
            doublePile[--dessus2] = element;
        }
        return true;
    }

    public E pop(boolean one) {
        if (one) {
            // Verifie que la pile contient au moin un element
            if (dessus1 >= 0) {
                /* Decrement l'index de dessus, efface le contenu de dessus
                et retourne la valeur qui etait au dessus1 */
                E reponse = doublePile[dessus1];
                doublePile[dessus1] = null;
                dessus1--;
                return reponse;
            }
        } else if (!one && dessus2 < nombreMaximalElements) {
            E reponse = doublePile[dessus2];
            doublePile[dessus2] = null;
            dessus2++;
            return reponse;

        }
          else {
        // Si aucune des pile ne contient d'element
            throw new IllegalStateException(
                "La double pile est vide, vous ne pouvez pas obtenir d'element. ")
```

```java
        }
    }

    /* En fonction du boolean fourni, on verifie si la pile
     * est plein et retourne l'element du dessus1 ou dessus2 */
    public E top(boolean one) {
        if (one) {
            if (dessus1 >= 0) {
                return doublePile[dessus1];
            }
        } else if (!one && dessus2 < nombreMaximalElements)  {
            return doublePile[dessus2];
        }
          else {
              throw new IllegalStateException(
                  "La double pile est vide, vous ne pouvez pas regarder l'element. ")
        }
    }

    /* En fonction du boolean fourni, on retourne la valeur
     * dessus1 + 1 (taille pile 1) ou nbMaximalElements - dessus (taille pile 2) */
    public int size(boolean one) {
        if (one) {
            return dessus1 + 1;
        } else {
            return nombreMaximalElements - dessus2;
        }
    }


    public void print() {
        System.out.println("Pile 1:");
        for (int i = 0; i <= dessus1; i++) {
            System.out.println(doublePile[i]);
        }
        System.out.println("Pile 2:");
        for (int i = nombreMaximalElements - 1; i >= dessus2; i--) {
            System.out.println(doublePile[i]);
        }
    }
}
```

Section 4

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».

```java
/* Seules les classes qui extends l'interface Java Comparable peuvent Ãªtre stockes dans
 * la SpecialArrayStack. Notre methode push(E e) a besoin de comparer l'element qu'on est sur
 * le point d'ajouter (e) a l'element Maximal  Maximaux[prochainMaximum].
 * Puisqu'on ne connait pas a l'avance le tyupe des objets stockes dans la
 * SpecialArrayStack, il faut utiliser le type generique E */
```

```java
public class SpecialArrayStack<E extends Comparable<E>> implements Stack<E> {

    private static final int NombreElementsMaximal = 100;
    private E[] Pile;
    private E[] Maximaux; // Tableau pour stocker les Ã©lÃ©ments maximaux
    private int dessus = -1;
    private int prochainMaximum = -1; // Index pour suivre le prochain Ã©lÃ©ment maximal


    // Constructeur
    public SpecialArrayStack() {
        Pile = (E[]) new Object[NombreElementsMaximal];
        // Initialisation du tableau des maximaux
        Maximaux = (E[]) new Object[NombreElementsMaximal];
    }

    // ImplÃ©mente les mÃ©thodes de l'interface Stack...
    @Override
    public void push(E e) throws IllegalStateException {
        if (isFull()) {
            throw new
            IllegalStateException("La pile a atteint le nombre maximal d'Ã©lÃ©ments");
        }

        if (isEmpty() || e.compareTo(getMax()) >= 0) {
            prochainMaximum++;
            // Ajoute le nouveau maximum au tableau des maximaux
            Maximaux[prochainMaximum] = e;
        }
        dessus++;
        Pile[dessus] = e; // Ajoute l'Ã©lÃ©ment Ã  la pile
    }

    @Override
     public E pop() {
            if (isEmpty()) {
                throw new
                IllegalStateException("La pile est vide, vous ne pouvez pas retirer d'Ã©lÃ©ment.");
            }

            E elementRetourne = Pile[dessus];
            Pile[dessus] = null; // Efface artificiellement l'Ã©lÃ©ment
            dessus--;

            if (elementRetourne.equals(getMax())) {
            // Retire le maximum actuel du tableau des maximaux
                Maximaux[prochainMaximum] = null;
                prochainMaximum--;
            }

            return elementRetourne;
        }

    @Override
    public E top() {
        if (isEmpty()) return null;
        return Pile[dessus];
    }

    @Override
    public int size() {
        return (dessus + 1);
    }

    @Override
    public boolean isEmpty() {
      // Lorsque le dessus a l'index 1, on sait que la pile est vide
        return (dessus == -1);
    }

    @Override
```

```java
    public boolean isFull() {
        return (dessus == NombreElementsMaximal - 1);
    }


    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("Contenu de la pile: [");
        for (int i = dessus; i >= 0; i--) {
            sb.append(Pile[i]);
            if (i > 0) sb.append(", ");
        }
        sb.append("]");
        return sb.toString();
    }

    public E getMax() {
        if (prochainMaximum == -1) {
            throw new IllegalStateException("La pile est vide, il n'y a pas d'Ã©lÃ©ment maximal.");
        }
        return Maximaux[prochainMaximum];
    }
}
```