# Parcours séquentiel d'un tableau

**S6 - Algorithmique (1)**

## 1. Recherche d'une occurrence

Considérons un tableau. On souhaite disposer d'un algorithme permettant de rechercher une occurrence d'une valeur donnée dans ce tableau. Plus précisément, nous allons définir une fonction qui recherche une valeur donnée dans un tableau et qui retourne le tableau des indices des occurrences de cette valeur dans le tableau. Dans le cas où la valeur n'est pas présente dans le tableau, la fonction retournera un tableau vide.

La méthode est très simple. On parcourt le tableau en testant à chaque fois si la valeur courante est égale à la valeur recherchée. Si c'est le cas, on ajoute l'indice de la valeur courante dans le tableau des indices des occurrences.

```
def occurrences(tab, val):
    """Retourne un tableau contenant les indices des occurrences de val dans tab"""
    indices = []
    for i in range(len(tab)):
        if tab[i] == val:
            indices.append(i)
    return indices
```

Exemple d'utilisation :

```
tab = ["DO", "RE", "MI", "FA", "SOL", "LA", "SI", "DO"]
print(occurrences(tab, "DO"))
print(occurrences(tab, "MI"))
print(occurrences(tab, "UT"))
```

```
[0, 7]
[2]
[]
```

**Complexité de l'algorithme** : Comptons le nombre d'itérations et de tests. Notons $n$ la taille du tableau, l'algorithme parcourt toutes les valeurs du tableau. Il y a donc au total $n$ itérations. De plus, nous avons $n$ tests. Il y a donc $2n$ opérations au total. On peut donc dire que l'algorithme est de complexité $\mathcal{O}(n)$.

## 2. Recherche d'un extremum

Dans cette partie, nous considérons un tableau dont les éléments sont des nombres. Nous allons définir une fonction qui recherche le plus grand élément du tableau. Pour cela, on commence par choisir comme maximum temporaire le premier élément du tableau. On parcourt ensuite le tableau en testant à chaque fois si la valeur courante est plus grande que le maximum temporaire. Si c'est le cas, on met à jour le maximum temporaire avec la valeur courante.

```
def max(tab):
    """Retourne le plus grand élément du tableau"""
    m = tab[0]
    for i in range(1, len(tab)):
        if tab[i] > m:
            m = tab[i]
    return m
```

Exemple d'utilisation :

```
tab = [201, 203, 35, 448, 55, 16, 2023, 14, 999, 100]
print(max(tab))
```

```
2023
```

**Complexité de l'algorithme** : La boucle `for` parcourt toutes les valeurs du tableau, sauf la première. Il y a donc au total $n-1$ itérations. Nous avons aussi $n-1$ comparaisons. Au total, le nombre d'opérations est donc de $2n-2$. On peut donc dire que l'algorithme est de complexité $\mathcal{O}(n)$.

## 3. Calcul d'une moyenne

Dans cette partie, nous considérons un tableau dont les éléments sont des nombres. Nous allons définir une fonction qui calcule la moyenne des éléments du tableau.

```
def moyenne(tab):
    """Retourne la moyenne des éléments du tableau"""
    s = 0
    for i in range(len(tab)):
        s += tab[i]
    return s / len(tab)
```

Exemple d'utilisation :

```
tab = [201, 203, 35, 448, 55, 16, 2023, 14, 999, 100]
print(moyenne(tab))
```

```
409.4
```

**Complexité de l'algorithme** : La boucle `for` parcourt toutes les valeurs du tableau. Il y a donc au total $n$ itérations. Nous avons aussi $n$ additions et une division. Au total, le nombre d'opérations est donc de $2n+1$. On peut donc dire que l'algorithme est encore de complexité $\mathcal{O}(n)$.