Algorithmique 5 – Différents algorithmes de parcours

I – Recherche linéaire dans un tableau

Implémentation : réalisation concrète d'un algorithme en machine à l'aide d'un langage de programmation. Il existe toujours plusieurs implémentations, non distinguables de l'extérieur.

1.Recherche d'une PREMIERE occurrence

Il s'agit d'un problème où on peut sortir avant d'avoir tout parcouru. On aurait donc tendance à privilégier un TANT QUE.

Implémentation A : while avec 1 sortie

```
1    def rechercher(t:list, x:'elt') -> int:
2        i = 0
3        longueur = len(t)
4        while i < longueur and t[i] != x:
5            i = i + 1
6        if i < longueur:
7            reponse = i
8        else:
9            reponse = -1
10       return reponse
```

Avantage : propre. Désavantage : un peu lourd ?

Implémentation B : while avec 2 sorties

```
1    def rechercher(t:list, x:'elt') -> int:
2        i = 0
3        longueur = len(t)
4        while i < longueur and t[i] != x:
5            i = i + 1
6        if i < longueur:
7            return i
8        return -1
```

Avantage : -2 lignes Désavantage : 2 sorties

Implémentation C : un peu bricolée

```
1    def rechercher(t:list, x:'elt') -> int:
2        for i in range(len(t)):
3            if t[i] == x:
4                return i
5        return -1
```

Avantage : court Dés. : for pour coder un TQ.

Toutes les implémentations précédentes sont à coût LINEAIRE et de complexité **O(n)** car on peut arrêter avant la fin.

On peut trouver des **variantes**. Par exemple :

2.Recherche de la DERNIERE occurrence

Cette fois, on doit aller au bout à chaque fois, on peut privilégier un POUR.

Implémentation D : for avec 1 sortie

```
1    def rechercher(t:list, x:'elt') -> int:
2        reponse = -1
3        for i in range(len(t)):
4            if t[i] == x:
5                reponse = i
6        return reponse
```

Implémentation E : un while ?

```
1    def rechercher(t:list, x:'elt') -> int:
2        reponse = -1
3        i = 0
4        while i < len(t):
5            if t[i] == x:
6                reponse = i
7            i = i + 1
8        return reponse
```

Toutes les implémentations précédentes sont à coût LINEAIRE et de complexité **Θ(n)** car on ne pourra jamais trouver de meilleur cas.

3. PREDICAT DE PRESENCE

```
1    def est_present(t:list, x:'elt') -> bool:
2        return rechercher(t, x) >= 0
```

Il s'agit d'une fonction-prédicat car elle renvoie True ou False : elle prédit la présence de x dans t.

C'est ce qui se passe lorsqu'on utilise le mot-clé **in**.

II – Recherche du maximum et du minimum

Algorithme du maximum A

ENTREES : t un tableau (et sa longueur)
Précondition 1 : tableau t NON VIDE
Précondition 2 : on peut utiliser > sur les éléments
SORTIE : L'élément maximum du tableau.

```
max_tempo ← t[0]
i ← 1
TANT QUE i < longueur
    SI t[i] > à max_tempo
        max_tempo ← t[i]
    Fin Si
    i ← 1 + 1
Fin Tant que
Renvoyer max_tempo
```

Algorithme du maximum B

ENTREES : t un tableau (et sa longueur)
Précondition 1 : tableau t NON VIDE
Précondition 2 : on peut utiliser > sur les éléments
SORTIE : L'élément maximum du tableau.

```
max_tempo ← t[0]
POUR i de 1 à longueur-1 (inclus)
    SI t[i] est supérieure à max_tempo
        max_tempo ← t[i]
    Fin Si
Fin Pour
Renvoyer max_tempo
```

Implémentation de la valeur maximale

```
1    def rechercher_maximum(t:'list NV') -> 'elt':
2        max_temp = t[0]
3        for i in range(1, len(t)):
4            if t[i] > max_temp:
5                max_temp = t[i]
6        return max_temp
```

Implémentation de la valeur minimale

```
1    def rechercher_minimum(t:'list NV') -> 'elt':
2        min_temp = t[0]
3        for i in range(1, len(t)):
4            if t[i] < min_temp:
5                min_temp = t[i]
6        return min_temp
```

Implémentation de l'indice de la valeur maximale

```
1    def rechercher_pos_max(t:'list NV') -> int:
2        imax = 0
3        for i in range(1, len(t)):
4            if t[i] > t[imax]:
5                imax = i
6        return imax
```

Implémentation de l'indice de la valeur maximale

```
1    def rechercher_pos_min(t:'list NV') -> int:
2        imin = 0
3        for i in range(1, len(t)):
4            if t[i] < t[imin]:
5                imin = i
6        return imin
```

Toutes les implémentations précédentes sont à coût LINEAIRE et de complexité **Θ(n)** car on ne pourra jamais trouver de meilleur cas.

III – Somme et valeur moyenne

Calculer une somme

```
1    def calculer_somme(t:'list NV') -> int:
2        s = 0                                  _____
3        for i in range(len(t)):                _____
4            s = s + t[i]                       _____
5        return s                               _____
```

Calculer une somme v2

```
1    def calculer_somme(t:'list NV') -> int:
2        s = 0                                  _____
3        for v in t:                            _____
4            s = s + v                          _____
5        return s                               _____
```

Ces fonctions ont un coût LINEAIRE, et de complexité **Θ(n)**

Valeur moyenne si t NON VIDE

```
1    def calculer_moyenne(t:'list NV') -> int:
2        return calculer_somme(t) / len(t)
```

`calculer_somme()` est de complexité **Θ(n).**
`len()` est de complexité **Θ(n).**
Cette fonction est donc en **Θ(n+n)** et donc en **Θ(n).**