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.

1.Recherche d'une PREMIERE occurrence

On peut répondre avant d'avoir tout parcouru. On aurait donc tendance à privilégier un TANT QUE.

Implémentation A : while avec 1 sortie

```
def rechercher(t:list, x:'elt') -> int:
    i = 0
    while i < len(t) and t[i] != x:
        i = i + 1
    if i < len(t):
        reponse = i
    else:
        reponse = -1
    return reponse
```

Avantage : propre. Désavantage : un peu lourd ?

Implémentation B : for, deux sortie

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

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

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

2.Recherche de la DERNIERE occurrence

Implémentation C : un while

```
def rechercher(t:list, x:'elt') -> int:
    i = len(t)-1
    while i >= 0 and t[i] != x:
        i = i - 1
    if i >= 0:
        reponse = i
    else:
        reponse = -1
    return reponse
```

Avantage : propre. Désavantage : un peu lourd ?

Implémentation B : for, deux sortie

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

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

3. PREDICAT DE PRESENCE

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

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**.

Même coût que rechercher() : LINEAIRE en **O(n).**

II – Recherche du maximum et du minimum

Algorithme du maximum A

ENTREE : 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

ENTREE : 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 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 l’indice de la valeur maximale

```
1    def indice_du_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 la valeur minimale

```
1    def 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 indice_du_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 ces implémentations sont à coût LINEAIRE en **Θ(n)** car on doit toujours lire les n cases.

Explications de ce coût :

Ligne 2 à coût constant.

Lignes 3-4-5 à coût linéaire puisqu’on réalise en gros n fois les lignes 4-5 qui sont à coût constant : n * 1 = n.

Ligne 6 : coût constant.

Avec les notations asymptotiques, on pourra juste écrire **Θ(1 +n*1 + 1) = Θ(n)**

III – Somme et valeur moyenne

Calculer une somme

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

Explications de ce coût :

Ligne 2 à coût constant.

Lignes 3-4 à coût linéaire puisqu’on réalise n fois la lignes 3 qui est à coût constant : n * 1 = n.

Ligne 5 : coût constant.

Avec les notations asymptotiques, on pourra juste écrire **Θ(1 +n*1 + 1) = Θ(n)**

Valeur moyenne si t NON VIDE

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

`somme()` est à coût linéaire en **Θ(n).**

Explications de ce coût :

L’appel à la fonction **somme()** à coût linéaire.

L’appel à la fonction **len()** est à coût constant.

La division est à coût constant (par rapport au nombre de cases du tableau).

Avec les notations asymptotiques, on pourra juste écrire **Θ(n + 1 + 1) = Θ(n)**