4 - Exemples d'algorithmes de recherche dans un tableau ou une liste

I – Introduction

Le problème de la recherche est étant donnée une collection indexés d'objets C et un objet o, déterminer si o appartient à C.
On peut aussi avoir des variantes :

- recherche de doublon
- nombre d'occurrence
- min/max
- position / à la position

Il existe des solutions à ces problèmes de recherche sous forme d'algorithme. Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou d'obtenir un résultat.
Ces algorithmes peuvent être différents selon le type de collection d'objets dans laquelle on lance la recherche :

- Les tableaux : suites indexés d'éléments stocké en mémoire de manière contigüe, l'accès à un élément se fait en temps constant grâce à une opération arithmétique impliquant la taille mémoire de la case du tableau ainsi que la position de la case.
- Les listes : les listes chainés sont des collections d'objets non contiguë en mémoire, chaque élément (maillon) est relié au suivant par pointeur. Au debut on ne connaît que le premier maillon on accède au maillon à la position i en temps O(i). (*voir implémentation python*)

```python
class Maillon:

    def __init__(self,value, suivant=None):
        self.value=value
        self.suivant=suivant


class Linked_List:

    def __init__(self,premier=None):
        self.premier=premier
        self.taille= 0
        if premier is not None:
            self.taille+= 1

    def __len__(self):
        return self.taille

    def __str__(self):
        courant = self.premier
        s = "["
        while courant is not None:
            s+=str(courant.value)+", "
            courant = courant.suivant
        return s+"]"


l = Linked_List(Maillon(1,Maillon(4,Maillon (45, Maillon (2,None)))))
print(l)
```

Il existe deux cas, soit la collection n'est pas triée soit elle l'est.

II – Recherche dans une structure non triée

Dans ce premier cas, les éléments ne sont pas trié ou pas comparable entre eux (type différent)

a- Algorithme de recherche séquentielle

C'est un technique qui consiste à passer en revue les éléments jusqu'à ce qu'on trouve le bon ou que la collection d'objet ait été entièrement parcourue.

Voici un exemple en pseudo-code d'un algorithme de recherche d'appartenance d'un objet à une collection :

```
i <- 0
tantQue ( i < longueur(tab) ) faire
        si (tab[i] = valeur) alors
                retourner VRAI
        finSi
        i <- i+1
finTantQue
retourner FAUX
```

La complexité est la même pour les tableaux et les listes chainées :

- 1er cas : l'élément est au début et donc la complexité est O(1)
- 2ème cas : l'élément est à la fin et donc la complexité est en 0(len(liste))
- 3ème cas : l'élément est à la position 1<i<len(liste) et donc la complexité est O(i)

Développement : autre type de recherche, implémentation pour les listes et les tableaux non triés, explication des complexités

- Recherche de position / à la position
- Recherche de doublon
- Recherche d'occurrence
- Recherche du min/max

b- Algorithme de recherche particulier

L'algorithme est en question est appelé algorithme de recherche adaptative. Il est amené à modifier la liste ou le tableau. Il est assez utile quand les éléments ne sont pas comparables entre eux mais qu'on effectue plusieurs recherche de mêmes éléments.

L'algorithme va effectuer une recherche séquentielle jusqu'à trouver l'élément et ensuite changer cet élément de place dans la liste. Plusieurs possibilités pour changer de place :

- On met l'élément à la première place de la collection
  Complexité tableau : O(i) + O(1)
  Complexité liste : O(i) + O(n-i) + O(n)

- On avance l'élément de 1 dans la collection (inversion des valeurs)
  Complexité tableau : O(i) + O(1)
  Complexité liste : O(i) + O(1)

- On avance l'élément d'un certain pas (ex : moitié de la distance avec le premier élément.)
  Complexité tableau : O(i) + O(n-i+k)
  Complexité liste : O(i) + O(1)

Dû à l'espace mémoire contiguë qu'il occupe, le tableau s'avère être moins efficace que la liste pour ce type d'algorithme, en effet, l'insertion au milieu du tableau implique le décalage des éléments suivants. Plus on insère en début de liste, plus il y a d'élément à décaler.

Développement : Implémentation des 3 méthodes de recherche auto-adaptative

- Sur les tableaux
- Sur les listes

III – Recherche dans une structure triée

Pour qu'une collection soit triée, il faut que les éléments soient comparable 2 à 2. Il doit donc exister une relation d'ordre total sur cet ensemble d'objet qu'on nommera l'ensemble E

$$\forall x, y \in E \quad x \leq y \text{ ou } y \leq x.$$

Propriétés :

- si x ≤ y et y ≤ z, alors x ≤ z (transitivité)
- si x ≤ y et y ≤ x, alors x = y (antisymétrie)
- x ≤ x (réflexivité)
- x ≤ y ou y ≤ x (totalité).

Les trois premières propriétés sont celles faisant de ≤ une relation d'ordre. La quatrième fait de cet ordre un ordre total.

Maintenant que nous possédons une collection d'objets triée, regardons si cela change quelque chose pour les algorithmes de recherche séquentielle et si nous pouvons penser à un autre type d'algorithme.

a- Algorithme de recherche séquentielle

Nous allons toujours passer en revue les éléments un par un, mais nous avons maintenant une information supplémentaire, les éléments sont ordonnés. Par exemple, pour des nombres triés dans l'ordre croissant, on va arrêter la recherche dès que notre valeur est plus petite que la valeur de l'élément courant.

```
i <-0
tantQue ( i < longueur(tab) && valeur <= tab[i] ) faire
        si (tab[i] = valeur) alors
                retourner VRAI
        finSi
        i <- i+1
finTantQue
retourner FAUX
```

(Montrer le code python pour liste et tableau)

Développement : autre type de recherche, implémentation pour les listes et les tableaux triés, explication des complexités

- Recherche de position / à la position
- Recherche de doublon
- Recherche d'occurrence
- Recherche du min/max

b- Algorithme de recherche dichotomique

Du grec « dicho » qui veut dire « en deux » et du verbe « tomie » qui signifie « couper », la recherche dichotomique est un algorithme de recherche pour trouver la position d'un élément dans un tableau trié. Le principe est le suivant : comparer l'élément avec la valeur médiane ; si les valeurs sont égales, la tâche est accomplie, sinon on continue dans la partie droite du tableau si la valeur est plus grande que la valeur médiane ou à gauche si la valeur est plus petite.

Voici un pseudo code de l'algorithme en version itérative

```
debut <- 0
fin <- longueur tableau
tantQue fin >=debut
        mediane <- (debut+fin) // 2
        Si L[mediane] = valeur
                renvoyer mediane
        ElseSi L[mediane] < valeur
                debut = mediane+1
        ElseSi L[mediane] > valeur
                fin = mediane -1
renvoyer NULL
```

Cette recherche dichotomique n'a pas d'intérêt sur les listes chainées, en effet celle-ci ne possède pas d'accès direct en temps constant aux éléments.

Preuve de Complexité :

Posons n la longueur la taille du tableau
On sait que n est divisé par deux à chaque itération de la boucle, donc la taille final du tableau est $[n/2^k]$ avec k entier représentant le nombre d'itérations.

La plus petite taille possible de l'instance est 1, on a donc

$$1 \leq \frac{n}{2^k}$$

$$2^k \leq n$$

$$k * \log(2) \leq \log(n)$$

$$k \leq \frac{\log(n)}{\log(2)}$$

$$k \leq \log_2(n)$$

La complexité de l'algorithme de recherche par dichotomie est donc en $O(\log_2(n))$

Développement : Implémentation itérative et récursive en python

IV – Ouverture : la table de symbole

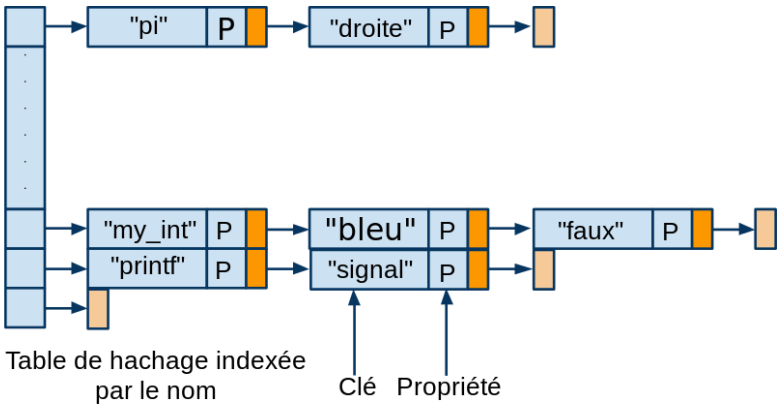
La table de symbole est une collection d'objet basé sur le principe de table de hashage. Elle est surtout utilisée pour répartir des chaines de caractères ou des mots d'un texte dans une table sans les classer alphabétiquement.

Le principe :

On choisit une taille de tableau N, on passe chaque chaine de caractère dans une fonction de hashage. Cette fonction va renvoyer un entier qui sera l'index de la chaine dans le tableau.

Que se passe-t-il si hash(mot)> taille tableau ?
hash(mot)%taille tableau

Que faire si deux valeurs de hashage de deux mots différents sont identiques ?
Au lien de pouvoir stocker une valeur par case, on créer une liste chainée (alvéole)
(« Propriété » pouvant être un entier représentant le nombre d'occurrence du mot dans le texte)



Comment bien choisir sa fonction de hashage ?
Il faut trouver un compromis entre :

- rapidité de la fonction de hachage
- taille à réserver pour l'espace de hachage
- réduction du risque des collisions

Comment rechercher dans ce tableau de symbole ?

```
h =hash(mot)%len
for elem in tableau_symbole[h] :
    if elem.cle == mot :
            return True
            break
return False
```