ALGORITHMIQUES DE RECHERCHE

Abu Abdallah Muhammad Ibn Musa AL-KHWÂRIZMÎ (780-850) était un mathématicien, géographe, astrologue et astronome perse. Il est considéré comme le "père de l'algèbre". Ses écrits en langue arabe ont permis la diffusion, jusqu'en Europe, des chiffres arabes et de l'algèbre, mot (al jabr) qui a pour origine le titre d'un des ses ouvrages. Il a, en entre autres, présenté les équations du second dégré et les méthodes de résolution des ces équations. Ses écrits seront traduits en latin vers le XIIe siècle. Le mot algorithme a pour origine algoritmus qui est la latinisation du nom d'Al-Khwârizmî.

1 – PROBLEMATIQUE

Est-ce qu'un objet x appartient à un tableau t ?

Les algorithmes de recherches d'éléments dans un tableau sont très utilisés dans les traitements de données. Il est nécessaire que ces algorithmes soient efficaces notamment pour le traitement de données importantes.

Un algorithme de recherche permet de **déterminer si une donnée est présente dans un tableau** et, éventuellement d'en **donner sa position**.

Nous nous intéresserons à deux types d'algorithme de recherche :

- algorithme de **recherche séquentielle** ;
- algorithme de **recherche dichotomique**.

2 – ALGORITHMIQUE DE RECHERCHE SEQUENTIELLE

2.1 – Présentation de l'algorithme

La première méthode consiste à parcourir le tableau à partir du premier élément. Ce type d'algorithme peut permettre de :

- déterminer si une donnée est présente dans le tableau ;
- donner l'emplacement de la première occurence ;
- donner l'emplacement de toutes les occurences.

L'algorithme suivant permet de déterminer si l'objet ***val*** est présent dans le tableau ***tab*** :

```
VARIABLES
tab : tableau d'entiers
val : nombre entier

DEBUT
    Pour tous les élements du tableau tab
        Si élement = val
            Retourner VRAI
    Retourner FAUX
FIN
```

La programmation de l'algorithme en Python est la suivante :

In [ ]:

```python
def recherche_seq1(tab, val) :
    for i in tab :
        if i == val :
            return True
    return False
```

In [ ]:

```python
t = [1, 10, 3, 5, 7, 30, 15, 8, 9, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
```

Question 1 : Tester la fonction ***recherche_seq1***. Expliquer son fonctionnement.

In [ ]:

```
recherche_seq1( , )
```

In [ ]:

```
recherche_seq1( , )
```

Reponse :

> L'algorithme suivant permet de donner l'emplacement de la première occurence de l'objet ***val*** dans le tableau ***tab*** :

```
VARIABLES
tab : tableau d'entiers
val : nombre entier
i : nombre entier

DEBUT
    i <-- 0
    Tant que i < taille(tab) et tab[i] != val
        i <-- i + 1
    Si i < taille(tab)
        Retourner i
    Sinon
        Retourner -1
FIN
```

> La programmation de l'algorithme en Python est la suivante :

In [ ]:

```
def recherche_seq2(tab, val) :
    i = 0
    while i<len(tab) and tab[i] != val :
        i = i + 1
    if i < len(tab) :
        return i
    else :
        return -1
```

In [ ]:

```
t = [1, 10, 3, 5, 7, 30, 15, 8, 9, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
```

Question 2 : Tester la fonction ***recherche_seq2***. Expliquer son fonctionnement.

In [ ]:

```
recherche_seq2( , )
```

In [ ]:

```
recherche_seq2( , )
```

Reponse :

> La fonction ***recherche_seq2bis*** est une variante de la fonction ***recherche_seq2bis*** qui s'écrit en utilisant une boucle ***for***.

Question 3 : Etablir l'algorithme de la fonction ***recherche_seq2bis***.

```
VARIABLES



DEBUT



FIN
```

Question 4 : Editer le programme de la fonction ***recherche_seq2bis***. Tester son fonctionnement.

In [ ]:

```
def recherche_seq2_bis(tab, val) : # à compléter
```

In [ ]:

```
t = [1, 10, 3, 5, 7, 30, 15, 8, 9, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
```

In [ ]:

```
recherche_seq2_bis( , )
```

In [ ]:

```
recherche_seq2_bis( , )
```

L'algorithme suivant permet de donner l'emplacement de toutes les occurences de l'objet ***val*** dans le tableau ***tab*** :

```
VARIABLES
tab : tableau d'entiers
pos : tableau d'entiers
val : nombre entier
i : nombre entier
DEBUT
    Pour i de 0 à taille(tab)
        Si tab[i] == val
            Ajouter i à pos
    Retourner pos
FIN
```

In [ ]:

```python
def recherche_seq3(tab, val) :
    pos = []
    for i in range(len(tab)) :
        if tab[i] == val :
            pos.append(i)
    return pos
```

In [ ]:

```python
t = [1, 10, 3, 5, 7, 30, 15, 8, 9, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
```

Question 5 : Tester la fonction ***recherche_seq3***. Expliquer son fonctionnement.

In [ ]:

```python
recherche_seq3( , )
```

In [ ]:

```python
recherche_seq3( , )
```

Reponse :

2.2 – Complexité de l'algorithme

Le **complexité** (ou **coût**) permet de quantifier l'efficacité d'un algorithme. Il existe deux types de coût :

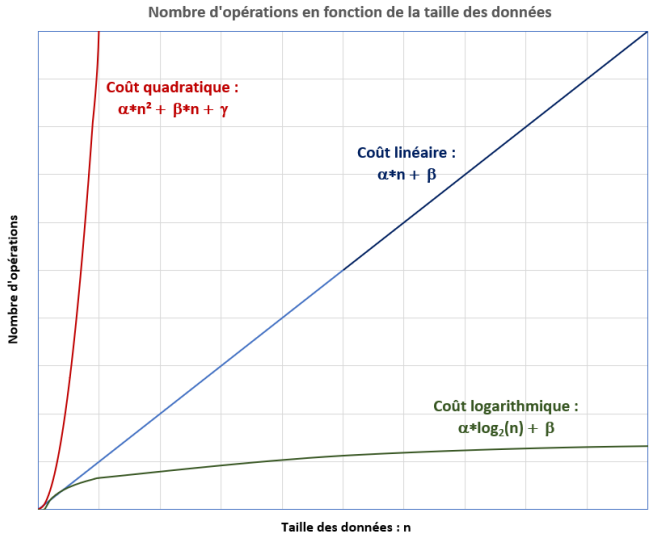
- la **complexité temporelle** (coût en temps) ;
- la **complexité spatial** (coût en mémoire).

Dans la suite du cours nous nous intéresserons uniquement au **coût en temps**. Ce coût correspond à un **ordre de grandeur** du nombre d'**opérations** effectuées par l'algorithme en fonction du nombre de données.

Ce coût peut être :

- **linéaire** : si le nombre d'opérations à effectuer peut s'écrire $\alpha * n + \beta$ avec n représentant la taille des données, l'algorithme a un **coût ou une complexité linéaire**.
- **quadratique** : si le nombre d'opérations à effectuer peut s'écrire $\alpha * n^2 + \beta * n + \gamma$ avec n représentant la taille des données, l'algorithme a un **coût ou une complexité quadratique**.
- **logarithmique** : si le nombre d'opérations à effectuer peut s'écrire $\alpha * log_2(n) + \beta$ avec n représentant la taille des données, l'algorithme a un **coût ou une complexité logarithmique**.

Ces différents coûts sont représentés sur la figure ci-dessous :



Nous pouvons constater qu'un algorithme qui a un coût logarithmique est plus efficace qu'un algorithme à coût linéaire qui est lui même plus efficace qu'un algorithme à coût quadratique.

Reprenons l'exemple de la fonction ***recherche_seq2***.

In [ ]:

```
def recherche_seq2(tab, val) :
    i = 0
    while i<len(tab) and tab[i] != val :
        i = i + 1
    if i < len(tab) :
        return i
    else :
        return -1

t = [1, 10, 3, 5, 7, 30, 15, 8, 9, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
```

Premier cas : la **donnée est présente** dans le tableau. Par exemple, nous voulons vérifier si le nombre 5 est présent dans le tableau. Une première occurence se trouve à la 4ième position ($j = 3$). Nous allons compter le nombre de fois que chacune des lignes de code est exécutée.

```
i = 0                                   --> 1 fois
    while i<len(tab) and tab[i] != val :   --> 4 fois (j + 1)
        i = i + 1                       --> 3 fois (j)
    if i < len(tab) :                   --> 1 fois
        return i                        --> 1 fois
    else :                              --> 0 fois
        return -1                       --> 0 fois
```

Dans ce cas le nombre d'opérations effectuées est de $2 * j + 4 = 10$.

Second cas : la **donnée n'est pas présente** dans le tableau. Par exemple, nous voulons, maintenant vérifier si le nombre 11 est présent dans le tableau. Celui-ci n'est pas présent dans le tableau possédant 20 éléments ($n = 20$). Comptons à nouveau le nombre de fois que chacune des lignes de code est exécutée.

```
i = 0                                   --> 1 fois
    while i<len(tab) and tab[i] != val :   --> 21 fois (n + 1)
        i = i + 1                       --> 20 fois (n)
    if i < len(tab) :                   --> 0 fois
        return i                        --> 0 fois
    else :                              --> 1 fois
        return -1                       --> 1 fois
```

Dans ce cas le nombre d'opérations effectuées est de $2 * n + 4 = 44$.

Le nombre d'opérations élementaires est plus important lorsque l'objet n'est pas dans le tableau. On parle de **coût ou de complexité dans le pire des cas**. Nous nous interesserons uniquement à ce coût temporel dans le pire des cas.

La fonction ***recherche_seq2*** présente a un coût de $2 * n + 4$. Ce coût dépend de la taille n des données de manière **linéaire**. On dit que cette fonction a **une compléxité linéaire**.

Question 6 : Déterminer le coût dans le pire des cas de la fonction ***recherche_seq3***. Vérifier que cette fonction présente un coût linéaire.

```
def recherche_seq3(tab, val) :
pos = []                            -->  fois
    for i in range(len(tab)) :      -->  fois
        if tab[i] == val :          -->  fois
            pos.append(i)           -->  fois
    return pos                      -->  fois
```

Réponse :

2.3 – Exercices

Question 7 : Editer l'algorithme puis le programme de la fonction ***moyenne*** qui permet de calculer la moyenne des éléments d'un tableau d'entiers. Tester son fonctionnement. Déterminer le coût de cet algorithme.

```
VARIABLES



DEBUT



FIN
```

In [ ]:

```
def moyenne(tab) :    # à compléter
```

In [ ]:

```
t = [9, 10, 3, 5, 7, 30, 15, 8, 1, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
moyenne(t)
```

Réponse :

Question 8 : Editer l'algorithme puis le programme de la fonction ***extremums*** qui permet de chercher les valeurs maximale et minimale des éléments d'un tableau d'entiers. Tester son fonctionnement. Déterminer le coût de cet algorithme.

```
VARIABLES



DEBUT



FIN
```

In [ ]:

```
def extremums(tab) :  # à compléter
```

In [ ]:

```
t = [9, 10, 3, 5, 7, 30, 15, 8, 1, 12, 12, 18, 5, 22, 5, 10, 5, 12, 8, 18]
extremums(t)
```

Réponse :

3 – ALGORITHME DE RECHERCHE DICHOTOMIQUE

3.1 – Présentation de l'algorithme

La recherche dichotomique s'effectue sur un **tableau préalablement trié**. Pour rechercher un élément dans un tableu par recherche dichotomique, il faut **découper le tableau en 2** puis effectuer un test pour déterminer dans quelle partie du tableau se trouve l'élément recherché. Ce principe est appelé "**diviser pour régner**".

L'algorithme suivant permet de donner l'emplacement de l'objet ***val*** dans le tableau ***tab*** :

```
VARIABLES
tab : tableau d'entiers
val : nombre entier
milieu : nombre entier
droite : nombre entier
gauche : nombre entier
DEBUT
    gauche  <-- 0
    droite  <-- taille(tab) - 1
    Tant que gauche < droite :
        milieu <-- (gauche + droite) / 2
        Si val = tab[gauche]
            Retourner gauche
        Sinon Si val < tab[milieu]
            droite <-- milieu - 1
        Sinon
            gauche <-- milieu + 1
    Retourner FAUX
FIN
```

Question 9 : Editer la fonction ***recherche_dicho1*** permettant d'implanter l'algorithme précédent. Tester son fonctionnement.

In [ ]:

```
def recherche_dicho1(tab, val) :   # à compléter
```

In [ ]:

```
t = [1, 3, 5, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 22, 23, 28, 29, 30, 33, 35]
```

In [ ]:

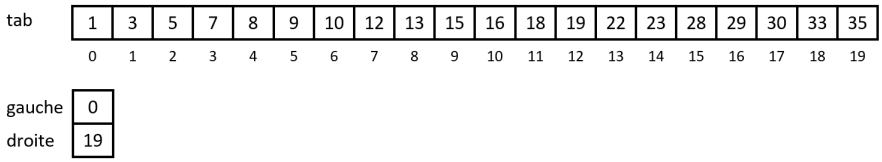
```
recherche_dicho1( ,  )
```

In [ ]:

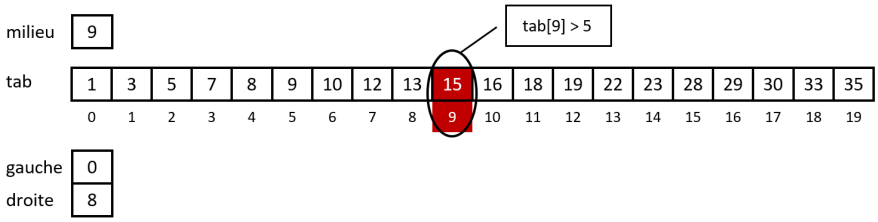
```
recherche_dicho1( , )
```

Le fonctionnement de l'algorithme dans le cas de la recherche de la valeur **5** dans le tableau est le suivant :

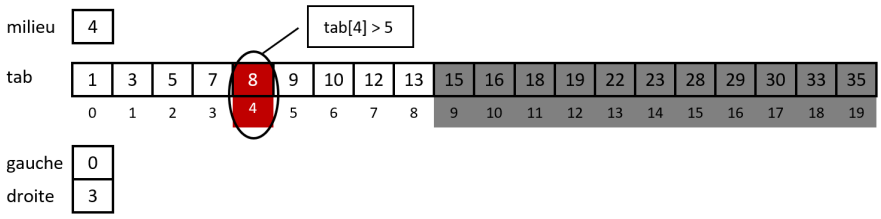
- Initialisation des différentes variables :



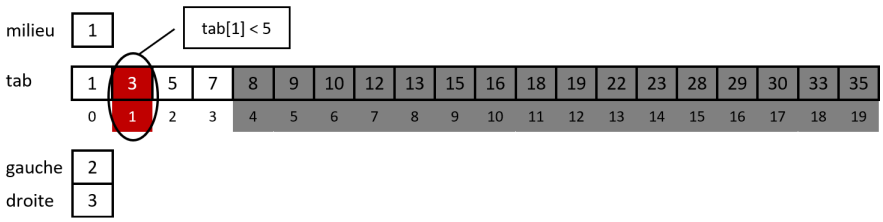
- 1ère itération :



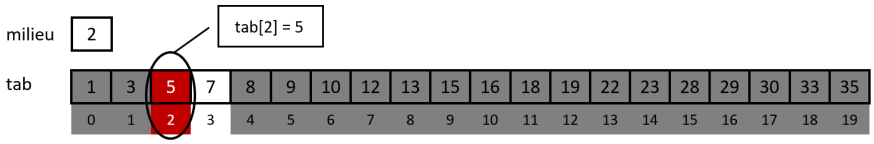
- 2nde itération :



- 3ème itération :



- 4ème itération :



Question 10 : Modifier la fonction ***recherche_dicho1*** afin qu'elle accepte en paramètre un tableau non trié. Tester son fonctionnement.

In [ ]:

```
def recherche_dicho2(tab, val) :  # à compléter
```

In [ ]:

```
t = [9, 10, 3, 5, 7, 30, 15, 8, 1, 12, 28, 18, 35, 22, 16, 13, 29, 33, 23, 19]
```

In [ ]:

```
recherche_dicho2( , )
```

In [ ]:

```
recherche_dicho1( , )
```

3.2 – Complexité de l'algorithme

La difficulté pour déterminer le coût de l'algorithme précédent consiste à **connaitre le nombre d'itérations effectuées** dans le cas le plus défavorable (l'objet n'est pas présent dans le tableau).

Pour déterminer ce nombre d'itérations, il faut connaître le nombre de fois qu'il faut diviser le tableau en 2 pour obtenir un tableau comportant un seul entier, c'est-à-dire **combien de fois faut-il diviser n (taille du tableau) par 2 pour obtenir 1** ?

Cette question se traduit par l'**équation mathématique** $\frac{n}{2^k} = 1$ avec k correspond au nombre de divisions qu'il faut effectuer. Pour résoudre cette équation, il faut faire appel au **logarithme base 2** noté log_2 qui est définit par $log_2(2^x) = x$.

$$\frac{n}{2^k} = 1 \implies n = 2^k \implies log_2(n) = log_2(2^k) = k \implies k = log_2(n)$$

Le nombre d'itérations est donc lié à la taille du tableau par un fonction logarithmique.

Question 11 : Déterminer le coût de la fonction ***recherche_dicho1***. Justifier que cet algorithme de recherche dichotomique est plus efficace qu'un algorithme de recherche séquentielle dans le cas d'un tableau préalablement trié.

```
def recherche_dicho1(tab, val) :
    gauche = 0                                  -->      fois
    droite = len(tab) - 1                       -->      fois
    while gauche < droite :                     -->      fois
        milieu = (gauche + droite) // 2         -->      fois
        if val == tab[milieu] :                 -->      fois
            return milieu                       -->      fois
        elif val < tab[milieu] :                -->      fois
            droite = milieu - 1                 -->      fois
        else :                                  -->      fois
            gauche = milieu + 1                 -->      fois
     return False                               -->      fois
```

Réponse :

3.3 – Terminaison de l'algorithme

Un algorithme ne doit comporter qu'un nombre fini d'étapes. Afin de prouver la terminaison d'un algorithme itératif utilisant des boucles conditionnelles (while), il faut utiliser la notion de **variant de boucle**. Un **variant de boucle** est une grandeur, variable ou expression, qui **varie à chaque itération**. Cette variation entrainera, à un moment ou à un autre, la **sortie de la boucle** et donc l'arrêt de l'algorithme.

Par exemple considérons le programme suivant qui parcourt un tableau t à la recherche d'un élément val :

In [ ]:

```
def recherche(t, val) :
    i = 0
    n = len(t)
    while i < n :
        if t[i] == val :
            return i
        i = i + 1
```

La variable i est incrémentée à chaque tour jusqu'à atteindre la valeur n = len(t). La variant est l'expression **n - i**, dont la valeur diminue à chaque tour, jusqu' à s'annuler. Nous pouvons donc justifier que ce programme se termine toujours.

Question 12 : Rechercher quelle variable ou expression pourrait être utilisée comme variant pour justifier la terminaison de l'algorithme de recherche dichotomique présenté plus haut.

Réponse :

L'expression **(droite - gauche)** correspond à la **longueur du tableau après chaque itération**. La longueur du tableau est divisée par 2, tant que l'objet n'a pas été trouvé, jusqu'à atteindre une longueur de 1.

Après k itérations : (droite - gauche) $\leq \frac{n}{2^k}$

(droite - gauche) étant strictement décroissante, après n itérations on obtient : $\frac{n}{2^k} = 1 \implies$ (droite - gauche) ≤ 1 donc droite $\leq$ gauche, ce qui permet de **sortir de la boucle** et donc l'**arrêt du programme**.

3.4 – Assertions

Le mécanisme d'**assertions**, proposé par Python, permet de s'assurer que certaines conditions dans un programme sont respectées et ainsi gérer de possibles erreurs d'utilisation prévues à l'avance. Pour cela il faut utiliser l'instruction **assert**.

Dans la fonction suivante, nous avons ajouter une assertion qui va créer un erreur et arrêter le programme dans le cas de l'appel de la fonction avec un **tableau vide**, c'est-à-dire si l'assertion **len(tab) != 0 n'est pas vérifiée**.

In [ ]:

```
def recherche_dicho3(tab, val) :
    assert len(tab) != 0
    gauche = 0
    droite = len(tab) - 1
    while gauche < droite :
        milieu = (gauche + droite) // 2
        if val == tab[milieu] :
            return milieu
        elif val < tab[milieu] :
            droite = milieu - 1
        else :
            gauche = milieu + 1
    return False
```

In [ ]:

```
t = []
```

In [ ]:

```
recherche_dicho3(t, 3)
```

Le tableau **tab** est trié. Si **val < t[0]** ou **val > t[n-1]** alors la valeur **val** n'est pas présente dans le tableau, il est donc inutile dans ce cas de faire la recherche de la valeur dans le tableau.

Question 13 : Modifier la fonction précédente en ajoutant une assertion qui arrête le programme si la valeur recherchée n'est pas dans le tableau. Tester le fonctionnement de la fonction.

In [ ]:

```
def recherche_dicho4(tab, val) :   # à compléter
```

In [ ]:

```
t = [1, 3, 5, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 22, 23, 28, 29, 30, 33, 35]
```

In [ ]:

```
recherche_dicho4(t, 5)
```

In [ ]:

```
recherche_dicho4(t, 11)
```

In [ ]:

```
recherche_dicho4(t, 40)
```

In [ ]:

```
recherche_dicho4(t, -1)
```

3.5 – Exercices

Question 14 : Editer un programme qui permette à l'ordinateur de jouer à *devine le nombre* contre l'utilisateur. L'utilisateur choisit un nombre entre 0 et 100 et l'ordinateur doit le trouver, le plus efficacement possible. A chaque proposition faite par l'ordinateur, l'utilisateur doit donner une réponse telle que "Plus grand", "Plus petit" ou "Bravo".

In [ ]:

```
def devine1(val) :   # à compléter
```

In [ ]:

```
devine1(5)
```

In [ ]:

```
devine1(110)
```

Question 15 : Mofifier le programme afin de comptabiliser et d'afficher le nombre de coups.

In [ ]:

```
def devine2(val) :  # à compléter
```

In [ ]:

```
devine2(67)
```

Question 16 : Ecrire en Python la fonction **dichotomie** qui prend en paramètres trois flottants a, b, précision et qui permet de déterminer par dichotomie à la précision près l'unique solution de $f(x) = x^2 - 10 * x + 25 = 0$ dans l'intervalle [a ; b]. Afficher le nombre d'étapes nécessaires pour trouver la solution. L'algorithme de cette fonction est le suivant :

```
VARIABLES
a : nombre flottant
b : nombre flottant
precision : nombre flottant

DEBUT
    m <-- (a + b) / 2
    Tant que (b - a) > precision et valeur absolue de f(m) > precision
        m <-- (a + b) / 2
        Si f(a) * f(b) < 0
            b <-- m
        Sinon
            a <-- m
        m <-- (a + b) / 2
    Retourner m
FIN
```

In [ ]:

```
import math
def dichotomie(a, b, precision) :   # à compléter
```

In [ ]:

```
dichotomie(0, 10, 1E-10)
```

Question 16 : Mofifier la fonction précédente àfin de trouver la solution de $f(x) = cos(x) = 0$ dans l'intervalle [a ; b].

In [ ]:

```
import math
def dichotomie2(a, b, precision) :   # à compléter
```

In [ ]:

```
dichotomie2(0,math.pi, 1E-5)
```