Fiche d'identité de l' algorithme de **RECHERCHE SEQUENTIELLE**

Principe : on cherche v dans une liste , on parcourt les éléments de la liste un par un jusqu'à avoir trouvé v → renvoie VRAI si v est trouvé, faux sinon

Exemple : feuilleter un livre page après page depuis le début pour retrouver une information

Pseudo Code

v ← élément cherché dans la liste « list »

Pour i dans list

si i =v alors

renvoyer Vrai

renvoyer Faux

Python

```
def chercher1(v,list):          #recherche la présence de l'entier v dans la liste "list"
    for i in list:              # on parcourt la liste
        if i==v:                # v est trouvé dans la liste
            return True         # on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas

def chercher2(v,list):          #recherche la présence de l'entier v dans la liste "list"
    if v in list:               # on parcourt la liste
        return True             # v est trouvé dans la liste et on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas
```

TERMINAISON de l'algorithme

Nous allons nous intéresser au fait que cet algorithme s'arrête....en effet, prouver ici que l'algorithme remplit bien son rôle (Correction de l'algorithme) est « trivial » (mais ce ne sera pas le cas des algorithmes de tri!)

Pour prouver qu'un algorithme s'arrête, on choisit une variable et on vérifie que la suite formée par les valeurs de cette variable au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Cette variable s'appelle un **variant de boucle.**

Dans ce cas précis, on peut choisir la suite du nombre d'éléments $(n_i)_{i\in\mathbb{N}*}$ de la liste qui n'ont pas encore été comparés à v au $i^{ème}$ tour.

* Supposons que $n_1 = 0$; la liste est vide, on n'entre pas dans la boucle et faux est retourné (l'algorithme se termine donc)

Si $n_1>0$ et vaut la longueur de la liste de départ : on entre dans la boucle, on compare v au 1^{er} élément ; soit il est égal à v et on sort de la boucle en renvoyant vrai(et l'algorithme se termine) ; soit il n'est pas égal à v et $n_2 = n_1 - 1$

* A la $k^{ième}$ itération, trois possibilités sont présentes :

soit v est égal au $k^{ème}$ terme : on sort de la boucle en renvoyant vrai(et l'algorithme se termine)

soit v n'est pas égal au $k^{ème}$ terme et $n_{k+1} = n_k - 1 = n_{k-1} - 2 = n_{k-2} - 3 = \ldots.. = n_1 - k-1 = n_1 - k+1$

soit $n_k = 0$, tous les éléments de la liste ont été comparés à v sans succès, on sort de la boucle et faux est retourné (l'algorithme se termine donc)

* La suite strictement décroissante $(n_i)_{i\in\mathbb{N}*}$ est minorée par zéro : elle converge donc (on décrémente à chaque tour ce nombre de une unité)

Complexité en temps : Dans le pire des cas (celui où v n'appartient pas à la liste) , on parcourt l'ensemble de la liste de longueur n *:* on effectue n comparaisons, le temps de calcul de la fonction EST PROPORTIONNEL à la longueur de la liste. On parle d'une complexité en **O(n)** (lire « grand o de n ») : c'est un **coût linéaire** Si on considère qu'une opération sur un ordinateur prend en moyenne environ 10^{-9} s, si la liste contient 10^6 éléments, le programme mettra 10^{-3} s ; pour 10^{12} éléments, le programme mettra 10^3 s etc....

On considère la liste **`list=[19,2,81,70,7,97,85,26,45,86]`**

81 est il présent dans cette liste ? ET 2 ? Et 77 ?

Quel est le moyen le plus simple pour savoir si un **entier v** est présent dans une liste notée « list » (éventuellement non triée) d'entiers déterminés ?

Écrire en français la démarche

Le parcours séquentiel d'une liste est écrit : « je compare le premier élément de la liste à v, si ils sont égaux, je renvoie Vrai . Sinon je passe au 2ème élément et je le compare à v, si ils sont égaux, je renvoie Vrai . Sinon Etc... Lorsque j'arrive à la fin de liste, si aucun élément ne vaut v, je renvoie Faux et je m'arrête.

Formalisation

1°) Pseudo code

Rédigeons ensemble un algorithme en pseudo code qui remplisse ce rôle

```
v ← élément cherché dans la liste « list »  # bien noter l'affectation par ←
Pour i dans list
    si  i = v alors                    #la notation = =  est du python pas du pseudo code
        renvoyer Vrai
renvoyer Faux                          # s'apesantir sur ce que doit renvoyer la fonction
```

Programmer le en Python (de 2 manières voir fiche ID séquentiel + Chercher.py)

2°) Analyser l'algorithme suivant :

```
v ← élément cherché                    #valeur du nombre entier cherché
deb ← indice du premier élément de la liste
fin ← indice du dernier élément de la liste
r ← Faux                               #Réponse initialisée à Faux
i ← deb
Tant que i<= fin et que r=Faux   faire
    Si list[i] = v alors
        r ← Vrai
    FinSi
    i ← i+1
FinTant que
Renvoyer la valeur de r
```

puis le faire tourner « à la main » sur la liste suivante

`list=[19,2,81,70,7,97,85,26,45,86]` en prenant exemple sur la recherche de 81 d'abord 2 puis 77 ensuite.

_____v=81 *r=Faux* *deb=0 et fin= 9* *1 – list[0]= 19 ≠ v alors i=0+1 = 1* *2- list[1]= 2 ≠ v alors i=1+1 = 2* *3- list[2]= 81 = v alors r=Vrai et i=2+1 = 3* *4- on renvoie Vrai* *_____v=2* *r=Faux* *deb=0 et fin= 9* *1 – list[0]= 19 ≠ v alors i=0+1 = 1* *2- list[1]= 2 = v alors r=Vrai et i=1+1 = 2* *3- on renvoie Vrai* *_____v=77* *r=Faux* *deb=0 et fin= 9* *1 – list[0]= 19 ≠ v alors i=0+1 = 1* *2- list[1]= 2 ≠ v alors i=1+1 = 2* *3- list[2]= 81 ≠ v alors i=2+1 = 3* *4- list[3]= 70 ≠ v alors i=3+1 = 4* *5- list[4]= 7 ≠ v alors i=4+1 = 5* *6- list[5]= 97 ≠ v alors i=5+1 = 6* *7- list[6]= 85 ≠ v alors i=6+1 = 7* *8- list[7]= 26 ≠ v alors i=7+1 = 8* *9- list[8]= 45 ≠ v alors i=8+1 = 9* *10- list[9]= 89 ≠ v alors i=9+1 = 10* *11- i=10>fin=9 : on renvoie Faux*	On remarquera que ce tri séquentiel est * assez rapide si v est en début de liste. * fastidieux et long si v n'appartient pas à la liste.(la liste proposée est volontairement de longueur 10) On prépare ainsi les notions de coût en temps

Programmer cet algorithme en Python (voir Chercher3.py)

3°) Complexité en temps

Combien de tours de boucles ont été nécessaires pour chercher 81 ? 2 et 77 ?

Il a fallu 3 tours de boucle pour 81 2 pour 2 et 10(=longueur de la liste) pour 77

Afin d'estimer la complexité de cet algorithme, on se place dans le « pire des cas » , c'est à dire le plus grand nombre de tours de boucle possible : décrivez les conditions de cette situation pour un nombre v donné et une liste donnée.

Dans ce pire des cas,combien d'étapes seront nécessaires pour chercher v dans une liste de longueur n?

Le pire des cas correspond à un élément v qui n'est pas dans la liste : on parcourt la boucle autant de fois qu'il y a d'éléments soit n fois pour une liste de longueur n