

## 1<sup>er</sup> pas d'ALGORITHMIQUE

**A - DÉFINIR** : Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes (du nom du mathématicien perse Al-Khwârizmî vers 780)

Ainsi, cette notion est très adaptée à l'informatique (l'ordinateur étant le réceptacle de ces instructions)

**B- RÉDACTION** : Il peut être écrit en langage « naturel » ou en pseudo code et **implémenté** (=traduit) dans différents langages : Python, Javascript, calculatrice...

**Quelques exemples d'instructions :**

Langage « naturel »	Pseudo code	Python	Javascript
Affecter à A la valeur 5	A ← 5	A=5	var A=5
Afficher A	Afficher A	print(A)	alert(A) / console.log(A)
Saisir A	Saisir A	A=float(input('A='))	Prompt A
<b>Si Condition Alors faire</b> Instructions1 <b>Sinon faire</b> Instructions2	<b>Si Condition Alors</b> Instructions1 <b>Sinon</b> Instructions2 <b>FinSi</b>	<b>if Condition :</b> Instruction1 <b>else :</b> Instruction2	<b>if</b> (condition1) { instructions 1; } <b>else</b> { instructions2;}
<b>Pour</b> i allant de 3 à 7 par pas de 1 <b>faire</b> Instructions	<b>Pour</b> i allant de 3 à 7 Instructions <b>FinPour</b>	<b>for</b> i in range (3,8) : Instructions	<b>for</b> (var i=3 ; i<8 ; i++) { instructions ; }
<b>Tant que</b> la condition est vraie <b>faire</b> Instructions	<b>Tant que</b> Condition est vraie <b>faire</b> Instructions <b>Fin du Tant que</b>	<b>while</b> condition : Instructions	<b>while</b> (condition) { Instructions ; }

Citez quelques différences et points communs de ces langages :

On retrouve les mots clé anglais **if, else, for** et **while** par contre **si** en python on indente les blocs avec JavaScript on les mets entre accolades. Les conditions logiques sont proposées entre parenthèses en JavaScript alors qu'en Python ça n'est pas obligatoire.

## C- COMPLEXITÉ TEMPORELLE

En informatique, la question de performance est centrale (surtout si ce temps utile pour obtenir un résultat est « long »). De manière générale, le traitement de données dépend du volume de ces données et de la nature du traitement que l'on fait.

Par exemple, une boucle peut posséder un grand nombre de répétitions le programme affiche 100 lignes

```
n=100
for i in range (n):
    print (i)
```

mais l'effet est encore démultiplié avec des boucles imbriquées. le programme affiche 100<sup>2</sup> lignes

```
n=100
for i in range (n):
    for j in range (n):
        print(i,j)
```

Ce nouveau programme affiche 100<sup>3</sup> lignes.

```
n=100
for i in range (n):
    for j in range (n):
        for k in range(n):
            print(i,j,k)
```

On multiplie maintenant n par 10, donner le coefficient de multiplication du nombre de lignes dans chacun de ces 3 programmes. **les coefficients seront respectivement égaux à 10, 100 et 1000.**

On parle alors de complexité TEMPORELLE de l'algorithme : on s'attache rarement à la détermination exacte de ce temps (on conserve une estimation) Remarque : bien que ce cours se limite à cette complexité temporelle, il existe aussi une complexité spatiale qui rend compte de l'espace mémoire pendant l'exécution

Pour mesurer la complexité temporelle, nous avons déjà vu quelques instructions Python de mesure du temps.

```
from time import time
debut = time()
# Code dont on mesure le temps
fin = time()
print("Temps passé : ", fin - debut)
```

Évaluez la durée des 3 programmes ci dessus.  
Comparez avec vos camarades

**Programme a mis 100 fois plus de temps pour s'exécuter par contre le troisième, lui, a mis 200x plus de temps !!! (il y a consensus pour les différentes personnes ayant testé le programme).**

## D - VALIDITÉ : CORRECTION et TERMINAISON d'un algorithme

Être certain qu'un algorithme donne le bon résultat est essentiel !!

1°) La **terminaison** stipule que les calculs décrits par l'algorithme s'arrêteront. Évidemment, savoir déceler cela avant toute implémentation évite le « plantage » de la machine.

On utilise souvent un **variant de boucle** : c'est une fonction entière, positive, qui décroît strictement

Il vérifie que la boucle se termine:  $f(i)=0 \Rightarrow !B$  (non B, ou B est un booléen)

2°) Afin de déterminer la **correction** d'un algorithme, on détermine un **invariant de boucle** : c'est une propriété ou une formule logique qui doit être vraie, à l'initialisation et à chaque itération de la boucle quel que soit le nombre d'itération. La difficulté réside donc à le trouver.

Exemple : dans un tableau de n entiers, trouver le plus grand élément.

```
def maximum(liste):
    n = len(liste)
    i=0
    maxi = liste[0]
    while i < n:
        if liste[i]>maxi:
            maxi = liste[i]
        i=i+1
    return maxi
# recherche le maximum dans la liste
# longueur de la liste
# le premier indice
# on démarre au premier élément de la liste
# on parcourt la liste (i indice d'un élément de la liste)
# la valeur rencontrée est supérieur à maxi
# on affecte une nouvelle valeur à maxi
# on s'occupe de l'indice suivant
# la liste est parcourue : maxi contient le maximum
```

**variant de boucle** :  $f(i) = n - i$

**invariants de boucle** :

- \* liste (elle n'est jamais modifiée)
- \* maxi est le plus grand élément de la partie de la liste dont les indices varient entre 0 et i (à la fin de la ligne `maxi = liste[i]`)
- \*  $0 \leq i \leq n$

Compléter l'algorithme suivant

```
q=0
n=    # a compléter
while n!=0:
    n-=3
    q+=1
print(q)
```

$n = 3$  ou  $6$  ou ... pour qu'il termine.  $n = 1$  ou  $2$  ou ... pour qu'il ne termine pas.

Déterminer la condition de terminaison sur le variant de boucle  $n$  :

Donner un invariant de boucle pour la fonction suivante qui calcule  $x$  à la puissance  $n$  :

$i$  est entre  $0$  et  $n-1$   
au niveau de la ligne for on aura toujours  $r = x^i$

```
def puissance(x,n):
    r=1
    for i in range(n):
        r=r*x
    return r
```

## E- ALGORITHMES DE RECHERCHE

### ALGORITHMES DE TRI

Trier un tableau de nombres, c'est ranger ces nombres dans l'ordre croissant.

Par exemple, le tableau  $[6\ 3\ 7\ 2\ 3\ 5]$  devient une fois trié  $[2\ 3\ 3\ 5\ 6\ 7]$ .

Bien sûr, on a souvent besoin de trier d'autres choses que des nombres (comme des mots par ordre alphabétique, des fichiers par longueur, des messages par date, les résultats d'une recherche par pertinence, des articles par référence, des personnes par date de naissance...), mais les algorithmes utilisés sont les mêmes.

Contrairement à ce qui se passe en général dans la vie courante, en informatique, on trie de très grandes quantités de données (quelques centaines de milliers d'éléments) à tout bout de champ. C'est pourquoi les informaticiens ont inventé de très nombreuses méthodes de tri, plus ou moins rapides et efficaces. Les ordinateurs passent énormément de temps à faire des tris : on considère aujourd'hui que les algorithmes de tri sont ceux qui sont les plus utilisés par les ordinateurs du monde entier !

## Fiche d'identité de l'algorithme de RECHERCHE SÉQUENTIELLE

**Principe** : on cherche  $v$  dans une liste : on parcourt les éléments de la liste un par un jusqu' avoir trouvé «  $v$  » ou être arrivé au bout de la liste → renvoie VRAI si  $v$  est trouvé, FAUX sinon

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

### Pseudo Code

Pour  $i$  dans list

Si  $i$  et  $v$  sont identiques alors :

On renvoi Vrai

On renvoi faux

.....  
.....

### Python

```
def chercher1(v,list):  
    for i in list:  
        if i==v:  
            return True  
    return False  
  
def chercher2(v,list):  
    if v in list:  
        return True  
    return False
```

#recherche la présence de l'entier  $v$  dans la liste "list"  
# on parcourt la liste  
#  $v$  est trouvé dans la liste  
# on indique qu'on a trouvé l'élément  
# la liste est parcourue :  $v$  ne lui appartient pas

#recherche la présence de l'entier  $v$  dans la liste "list"  
# on parcourt la liste  
#  $v$  est trouvé dans la liste et on indique qu'on a trouvé l'élément  
# la liste est parcourue :  $v$  ne lui appartient pas

### TERMINAISON de l'algorithme

Comme 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 !), nous allons concentrer nos efforts pour prouver que cet algorithme s'arrête.

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 avec  $v$  au  $i^{\text{ème}}$  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)

\* Si  $n_1 > 0$  et vaut la longueur de la liste de départ : on entre dans la boucle, on compare  $v$  au 1<sup>er</sup> 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  $k^{\text{ième}}$  itération, trois possibilités sont présentes :

- soit  $v$  est égal au  $k^{\text{ième}}$  terme : on sort de la boucle en renvoyant vrai (et l'algorithme se termine)
- soit  $v$  n'est pas égal au  $k^{\text{ième}}$  terme et  $n_{k+1} = n_k - 1 = n_{k-1} - 2 = n_{k-2} - 3 = \dots = n_1 - k + 1$
- 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 0 donc d'après le cours de spécialité maths de terminale elle converge (on décrémente à chaque tour ce nombre d'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**

## Travail maison (à faire une fois la page précédente couverte en classe)

Soit un l'algorithme suivant :

$v \leftarrow \text{élément cherché}$

*#valeur du nombre entier cherché*

$deb \leftarrow \text{indice du premier élément de la liste}$

$fin \leftarrow \text{indice du dernier élément de la liste}$

$r \leftarrow \text{Faux}$                     *#Réponse initialisée à Faux*

$i \leftarrow deb$

*Tant que  $i \leq fin$  et que  $r = \text{Faux}$  faire*

*Si  $list[i] = v$  alors*

$r \leftarrow \text{Vrai}$

*FinSi*

$i \leftarrow i + 1$

*FinTant que*

*Renvoyer la valeur de  $r$*

Analyser cet algorithme puis le faire tourner « à la main » sur la liste  $list = [19, 2, 81, 70, 7, 97, 85, 26, 45, 86]$  comme dans l'exemple suivant en recherchant 2 d'abord puis 77 ensuite.

Exemple avec 81	Avec 77
$v = 81$	$v = 77$
$r = \text{Faux}$	$r = \text{Faux}$
$deb = 0$ et $fin = 9$	$deb = 0$ et $fin = 9$
1 – $list[0] = 19 \neq v$ alors $i = 0 + 1 = 1$	1 – $list[0] = 19 \neq v$ alors $i = 0 + 1 = 1$
2- $list[1] = 2 \neq v$ alors $i = 1 + 1 = 2$	2 – $list[1] = 2 \neq v$ alors $i = 1 + 1 = 2$
3- $list[2] = 81 = v$ alors $r = \text{Vrai}$ et $i = 2 + 1 = 3$	3 – $list[2] = 81 \neq v$ alors $i = 2 + 1 = 3$
4- on renvoie Vrai	4 – $list[3] = 70 \neq v$ alors $i = 3 + 1 = 4$
A vous	5 – $list[4] = 7 \neq v$ alors $i = 4 + 1 = 5$
$v = 2$	6 – $list[5] = 97 \neq v$ alors $i = 5 + 1 = 6$
$r = \text{Faux}$	7 – $list[6] = 85 \neq v$ alors $i = 6 + 1 = 7$
$deb = 0$ et $fin = 9$	8 – $list[7] = 26 \neq v$ alors $i = 7 + 1 = 8$
1 – $list[0] = 19 \neq v$ alors $i = 0 + 1 = 1$	9 – $list[8] = 45 \neq v$ alors $i = 8 + 1 = 9$
2- $list[1] = 2 = v$ alors $i = 1 + 1 = 2$	10 – $list[9] = 86 \neq v$ alors $i = 9 + 1 = 10$
4- on renvoie Vrai	11 – on renvoie Faux

## Complexité en temps

Combien de tours de boucles ont été nécessaires pour chercher 81 ? 2 ? et 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 **Le pire cas c'est celui où la valeur  $v$  n'est pas dans la liste.**

Dans ce pire des cas, combien d'étapes seront nécessaires pour chercher  $v$  dans une liste de longueur  $n$  ? **La boucle tournera  $n$  fois. (+ étape d'affichage)**

## Fiche d'identité de l'algorithme de TRI par INSERTION

**Principe** : on parcourt les éléments de la liste de la gauche vers la droite. Pour chaque élément on le déplace dans la partie de liste jusqu'à ce qu'elle soit dans l'ordre. → tri en place

**Exemple** : le tri du jeu de cartes, on range une carte piochée dans son jeu déjà classé

### Pseudo Code

```
n ← nombre d'éléments de t
pour i allant de 1 à n-1
    j ← i
    tant que j>0 et t[j]<t[j-1]
        permuter t[j] et t[j-1]
        j ← j -1
    fin tant que
fin pour
```

### Python :

```
n=len(t)
for i in range(1,n) :
    j=i
    while j>0 and t[j]<t[j-1] :
        t[j],t[j-1]= t[j-1],t[j]
        j=j-1
```

### Algorithme valide

On choisit comme invariant de boucle H : « la liste  $t[0 : i+1]$  est triée par ordre croissant à l'issue de l'itération  $i$  »

**Initialisation** (est ce vrai avant d'entrer dans la boucle ?)

si  $i=0$ , au rang 0 on a  $t[0]$  ou encore  $t[0:0+1]$  : constituée d'un seul terme, elle est donc triée : H est vraie

**Conservation** = hérédité (reste vraie après une itération, si elle était vraie avant)

A la fin de l'itération  $i$ ,  $t[0 : i+1]=[t[0],t[1],...,t[i]]$  est supposée triée dans l'ordre croissant si H est vraie : on effectue un nouveau tour de boucle donc à l'itération  $i+1$ . La boucle effectue le tri

```
    tant que  $i+1 > 0$  et  $t[i+1]<t[i]$  //teste si  $t[i+1]$  est mal placé
        permuter  $t[i+1]$  et  $t[i]$ 
```

Alors la liste  $t[0 : i+2]=[t[0],t[1],...,t[i], t[i+1]]$  devient triée dans l'ordre croissant. H reste vraie

**Terminaison** (donne le résultat attendu en fin de boucle)

En sortie de boucle,  $i$  a pris sa dernière valeur  $i= n-1$  et la boucle permet d'effectuer le tri au rang  $n-1$

```
    tant que  $n-1 > 0$  et  $t[n-1]<t[n-2]$  //teste si  $t[n-1]$  est mal placé
        permuter  $t[n-1]$  et  $t[n-2]$ 
```

Malgré l'absence de « renvoi »(return en python) le tri est effectué sur la liste  $t$ , modifiée à chaque tour de boucle. La liste  $t$  est triée dans l'ordre croissant et la fonction tri a rempli son objectif

### Complexité en temps :

algorithme lent mais relativement efficace sur de courtes listes.

L'algorithme tirera en effet parti de tout ordre partiel présent dans le tableau. Jointe à la simplicité de l'algorithme, cette propriété le désigne tout naturellement pour "finir le travail" de méthodes plus ambitieuses comme le tri rapide, variante de « diviser pour régner » utilisée par la dichotomie)

Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectuera seulement  $n - 1$  comparaisons. Sa complexité dans le meilleur des cas est donc en  $O(n)$  : c'est un coût linéaire

Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer  $(n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{n(n-1)}{2}$  (formule spé maths 1<sup>ère</sup>) comparaisons et échanges,

On a donc une complexité dans le pire des cas du tri par insertion en  $O(n^2)$  : c'est un coût quadratique

Si tous les éléments de la série à trier sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de  $\frac{n^2-n}{4}$  comparaisons et échanges. La complexité en moyenne du tri par insertion est donc également en  $O(n^2)$

### Exercice 1 – ANALYSER et CONCEVOIR

On souhaite trier la liste de listes Pers selon le numéro situé en 2ème position dans les sous-listes que contient Pers :

```
Pers = [['Portillon',4],['Sam',3],['Julie',1],['Tom',2],['Charlie',5]]
```

Proposer une fonction mettant en œuvre un algorithme de tri par insertion permettant de trier la liste Pers

def triInser2(liste):

```
l=liste[:]
n=len(l)
for i in range (1,n):

    while i>0 and l[i][1]<l[i-1][1]:
        l[i-1],l[i]=l[i],l[i-1]
        i=i-1
    return(l)
```

### Exercice 2 – CONCEVOIR et COMPARER

Afin de comparer les coûts en temps des méthodes de tri, proposer une fonction qui génère aléatoirement un tableau de taille n choisi par l'utilisateur d'entiers positifs.

Compléter à l'aide la fonction déjà utilisée lors de l'introduction du cours d'algorithmique

Vu que le temps varie si on a un tableau initialement plus ou moins rangé, il sera intéressant de faire une moyenne sur plusieurs essais (en mettant la création des listes en dehors de la zone du programme servant à mesurer le temps écoulé.

```
from time import time
debut = time()

# Code dont on mesure le temps

fin = time()
print("Temps passé : ",fin - debut)
```

```
from random import shuffle
from time import time
```

```
def crea(n):
    liste=[i for i in range(n)]
    shuffle(liste)
    return liste
```

```
def triInsertion(liste):
    l=liste[:]
    n=len(l)
    for i in range (1,n):
```

```
        while i>0 and l[i]<l[i-1]:
            l[i-1],l[i]=l[i],l[i-1]
            i=i-1
        return(l)
```

```
taille=10
liste1=crea(taille)
debut = time()
triInsertion(liste1)
fin=time()
print("temps passé : ", fin - debut)
```

Remplissez le tableau ce dessous et comparer avec vos camarades (selon la taille et la machine utilisée)

Taille n du tableau	Coefficients multiplicateurs	Temps Tri insertion	Coefficients multiplicateurs
10		0	
100	10	0.00099945068359375	
1000	10	0.06096315383911133	60
5000	5	1.911820411682129	31
10000	2	6.302114963531494	3,31

Si la taille est multipliée par k, le temps de tri devrait théoriquement être multiplié par  $k^2$

La réalité ne colle pas trop avec les résultats obtenus car chaque liste générée aléatoirement ne va pas prendre autant de temps à gérer, le  $k^2$  est valide pour le pire des cas, pas pour le meilleur (celui où elle est déjà rangée, le coefficient devrait être de l'ordre de k)

# Exercices

## Exercice 1 – CONCEVOIR ET TRADUIRE

Écrire un algorithme qui compte le nombre d'éléments dans une liste, ce sans faire appel à une fonction dédiée (comme *len()* en Python)

Indiquer la complexité temporelle si la liste contient  $n$  entiers.

Implémenter cet algorithme en Python

### Algorithme :

```
Fonction : Long(Liste)
    Longueur = 0
    tant que Liste n'est pas vide :
        Longueur = longueur + 1
        On retire à liste son premier
        élément.
    Renvoyer Longueur
Fin fonction
```

### Fonction Python

```
def Long(Liste) :
    Longueur = 0
    L = Liste[ :]
    While L != [] :
        Longueur += 1
        L.pop()
    return Longueur
```

complexité temporelle :  $O(n)$

## Exercice 2 – CONCEVOIR ET TRADUIRE

1°) Écrire un algorithme qui permette de trouver le minimum d'une liste d'entiers.

### Algorithme

```
Fonction mini(Liste)
Liste est une Liste non vide
Minimum ← Liste[0]
Pour i allant de 1 à n – 1 :
    Si Liste[i] < Minimum
        Alors Minimum ← Liste[i]
Renvoyer Minimum
Fin de fonction
```

### Programme Python

```
def mini(Liste) :
    Minimum = Liste[0]
    for i in range(1, len(Liste)) :
        if Liste[i] < Minimum :
            Minimum = Liste[i]
    return Minimum
```

2°) Améliorer cet algorithme en *minimum\_indice(liste)* pour qu'il renvoie aussi le 1er indice du minimum rencontré

### Algorithme

```
Fonction minimum_indice(Liste)
Liste est une Liste non vide
Rang = 0
Minimum ← Liste[0]
Pour i allant de 1 à n – 1 :
    Si Liste[i] < Minimum
        Alors Minimum ← Liste[i]
        Rang = i
Renvoyer Minimum et Rang
Fin de fonction
```

### Programme Python

```
def minimum_indice(Liste) :
    Minimum = Liste[0]
    Rang = 0
    for i in range(1, len(Liste)) :
        if Liste[i] < Minimum :
            Minimum = Liste[i]
            Rang = i
    return Minimum, Rang
```

**A FAIRE ABSOLUMENT** Créer un algorithme *minimum\_indice(liste, j)* qui sera utile pour le tri séquentiel : il prend en paramètres la liste et un rang pour renvoyer l'indice du minimum de la liste comprise entre les indices  $j$  et  $n-1$



### Algorithme

```
Fonction minimum_indice(Liste,j)
Liste est une Liste non vide
Rang = j
Minimum ← Liste[j]
Pour i allant de j+1 à n - 1 :
    Si Liste[i]<Minimum
    Alors Minimum ← Liste[i]
    Rang = i
Renvoyer Minimum et Rang
Fin de fonction
```

### Programme Python

```
def minimum_indice(Liste,j) :
    Minimum = Liste[j]
    Rang = j
    for i in range(j+1,len(Liste)) :
        if Liste[i]<Minimum :
            Minimum = Liste[i]
            Rang = i
    return Minimum,Rang
```

3°) Créer enfin un algorithme *minimum\_indices(liste)* qui renvoie TOUS les indices de chacune des occurrences du minimum dans la liste. Estimer le coût en temps d'une telle recherche

### Algorithme

```
Fonction minimum_indices(Liste)
Liste est une Liste non vide
Rang = [0]
Minimum ← Liste[0]
Pour i allant de 1 à n - 1 :
    Si Liste[i]<Minimum
    Alors Minimum ← Liste[i]
    Rang = [i]
    Si Liste[i]==Minimum :
    On rajoute i à la fin de la liste Rang
Renvoyer Minimum et Rang
Fin de fonction
```

### Programme Python

```
def minimum_indices(Liste) :
    Minimum = Liste[0]
    Rang = [0]
    for i in range(1,len(Liste)) :
        if Liste[i]<Minimum :
            Minimum = Liste[i]
            Rang = [i]
        elif Liste[i]==Minimum :
            Rang.append(i)
    return Minimum,Rang
```

Complexité temporelle  $O(n)$

### Exercice 3 – TRADUIRE ET DÉVELOPPER

On associe des numéros à des personnages :

Ceci se traduit en Python par la liste :

```
Pers = [['Celine',1],['Lucas',2],['Stephane',3],['Louison',4],['Jean',5]]
```

Écrire une fonction qui permet de rechercher si un personnage de numéro  $v$  (un entier) quelconque est présent dans la liste : la fonction renverra le nom du personnage alors et *False* sinon

Personnage	Numéro
Céline	1
Lucas	2
Stéphane	3
Louison	4
Jean	5

```
def nom(Liste, Nombre) :
    for élément in Liste :
        if élément[1]==Nombre :
            return élément[0]
    return False
```

#### **Exercice 4** – ANALYSER ET CONCEVOIR

On prétend que la fonction suivante teste l'appartenance de la valeur  $v$  au tableau  $t$

Donner des tests pour cette fonction et en particulier des tests montrant plusieurs raisons pour laquelle cette fonction est incorrecte

Si le dernier élément de la liste n'est pas celui que l'on cherche alors la dernière valeur de `trouvee` est `False` quelle que soit les autres valeurs de la liste (et donc même si celle-ci ne contient que des valeurs  $t$ , ce qui serait bien sûr un parfait contre-exemple  $t=[v,v,v,v,v,v-1]$ )

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
    return trouvee
```

Que teste cette fonction en réalité ?

Elle ne teste que le dernier élément.

Aller plus loin....et que pensez vous de

(return a changé de place)

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
    return trouvee
```

Là c'est pire , ça s'arrête au bout d'un élément : le premier  
Donc  $t=[v+1,v,v,v,v]$  ne donne pas un résultat adapté

```
def appartient(v,t)
    for i in range(len(t)) :
        if t[i]==v :
            return True
    return False
```

## Fiche d'identité de l'algorithme de TRI par SÉLECTION (ou tri par extraction)

**Principe** : on va chercher le plus petit élément de la liste pour le mettre en **première position**, puis on repart du **second** élément et on va chercher le plus petit élément de la liste pour le mettre en **deuxième position**, etc...

### Pseudo Code

```
n ← nombre d'éléments de t
pour i allant de 1 à n-2
    i_min ← minimum(t,j)
    si i_min différent de j alors
        permuter t[i] et t[j]
    fin si
fin pour
```

### Python :

```
def minimum(liste,deb):
    """paramètres : liste est la liste complète
        déb est le rang de l'élément à partir
        du quel on cherche le plus petit élément
    sortie : le rang du plus petit élément"""
    mini=min(liste[deb:])
    return liste.index(mini,deb)
```

```
n=len(t)
for i in range(0,n-1):
    i_min= minimum(t,i)
    if i_min!=i :
        t[i],t[i_min]=t[i_min],t[i]
```

### Algorithme valide

On choisit comme invariant de boucle H : « la liste  $t[0 : i+1]$  est triée par ordre croissant à l'issue de l'itération  $i$  »

**Initialisation** (est ce vrai avant d'entrer dans la boucle ?)

si  $i=0$ , au rang 0 on a  $[t[0]]$  ou encore  $t[0:0+1]$  : constituée d'un seul terme, elle est donc triée : H est vraie

**Conservation** = hérédité (reste vraie après une itération, si elle était vraie avant)

A la fin de l'itération  $i$ ,  $t[i : i+1]=[t[0],t[1],...,t[i]]$  est supposée triée dans l'ordre croissant si H est vraie : on effectue un nouveau tour de boucle donc à l'itération  $i+1$ .

```
i_min ← minimum(t,j)
si i_min différent de j alors
    permuter t[i+1] et t[j]
```

Alors la liste  $t[0 : i+2]=[t[0],t[1],...,t[i], t[i+1]]$  devient triée dans l'ordre croissant. H reste vraie

**Terminaison** (donne le résultat attendu en fin de boucle)

En sortie de boucle,  $i$  a pris sa dernière valeur  $i= n-2$  et la boucle permet d'effectuer le tri au rang  $n-2$

```
i_min ← minimum(t,j) // indice du minimum entre t[n-2] et t[n-1]
si i_min différent de n-1 alors
    permuter t[n-2] et t[n-1]
```

Malgré l'absence de « renvoi »(return en python) le tri est effectué sur la liste  $t$ , modifiée à chaque tour de boucle. La liste  $t$  est triée dans l'ordre croissant et la fonction tri a rempli son objectif

**Complexité en temps** : on s'intéresse au nombre de comparaisons effectuées. La liste est triée par ordre croissant. La boucle pour de la fonction tri effectue  $n-1$  opérations. A chaque itération, la fonction minimum est appelée et effectue  $n - (j + 1)$  itérations. Sa complexité est donc en  $O(n^2)$  de l'ordre de  $n \times n$

### Formalisation

1°) Faire fonctionner à la main le tri par sélection sur un nouveau jeu de 4 cartes, puis sur une liste,  $[9,3,1,6]$  par exemple qui pourra être aussi illustrée si besoin par les 4 cartes de valeur correspondantes → Implémenter alors en pseudo code puis en Python cet algorithme

$[9,3,1,6] \rightarrow [1,9,3,6] \rightarrow [1,3,9,6] \rightarrow [1,3,6,9]$  (italique = sous liste sur laquelle on travaille, gras : minimum de cette sous liste)

2°) Quel est le pire des cas ? Combien y-a-il alors d'itérations dans la boucle pour? Combien y-a-il d'appel à la fonction minimum dans chaque tour de boucle? Combien d'itérations effectue cette dernière fonction ?

La liste est triée par ordre décroissant : on effectue  $n-1$  itérations (de l'ordre de  $n$ ) dans la boucle pour du tri. A chaque itération, la fonction minimum est appelée une fois et effectue  $n-(j+1)$  itérations (de l'ordre de  $n$  opérations) : la complexité est donc quadratique, en  $O(n^2)$ .

3°) Définir et trouver un Invariant de boucle

### Exercices

### Exercice 3 – ANALYSER et TRADUIRE

En supposant que le tri par sélection prend un temps directement proportionnel à  $n^2$  et qu'il prend 6,8 secondes pour trier 16000 valeurs (voir tableau des temps de ce tri), calculer le temps qu'il faudrait pour trier un million de valeurs

On utilise la proportionnalité :  $\frac{6.8 \times 1000000^2}{16000^2} \approx 26650 \approx 7 \text{ h } 23 \text{ min}$  tout de même.....Ce tri n'est pas celui que l'on emploie pour des milliers de données...heureusement !

Comparer ces temps avec les tris de Python

On pourra générer aléatoirement des listes de 16000 éléments (liste\_1) entre 1 et 10 000, puis de 1 million d'éléments (liste\_2) toujours entre 1 et 10 000

à l'aide du petit programme de comparaison d'écoulement du temps comparer les temps de tri de `liste.sort()` ; un tri en place et `sorted(liste)` qui crée, elle, une copie triée dans Python.

```
from time import time
debut = time()
# Code dont on mesure le temps
fin = time()
print("Temps passé : ", fin - debut)
```

Les résultats sont surprenants de rapidité

Temps passé L1 sorted : 0.003000020980834961

Temps passé L1 sort : 0.003000020980834961

Temps passé L2 sorted : 0.3500199317932129

Temps passé L2 sort : 0.32901859283447266

## Une nouvelle recherche....

Après la recherche séquentielle vue en début de chapitre, puis deux tris, nous allons de nouveau chercher un élément dans une liste mais cette fois-ci, TRIÉE ....

1°) Retrouvez le programme écrit en Python qui cache un nombre entier entre 1 et 100 et demande à l'utilisateur de le retrouver. [https://www.dimension-k.com/maths/NSI/INSI-\(03-boucles\)-jeu.py](https://www.dimension-k.com/maths/NSI/INSI-(03-boucles)-jeu.py)

On comptait même le nombre de coups (7)

Le nombre de candidats restants si le nombre n'est pas donné est au plus

$100 \rightarrow 50 \rightarrow 25 \rightarrow 13 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 1$  7 est le nombre minimum tel que  $2^7 > 100$

Observez et faire fonctionner le fichier Jeu dichotomisé depuis le site: décrire son rôle

[https://www.dimension-k.com/maths/NSI/INSI-\(16-algorithmes-tri\)-Jeu-dicho-inverse.py](https://www.dimension-k.com/maths/NSI/INSI-(16-algorithmes-tri)-Jeu-dicho-inverse.py)

Ce programme donne pour chaque nombre de 1 à 100, le nombre de coups pour retrouver ce nombre dans la liste triée de 1 à 100 (pas 1) par dichotomie.

2°) Nous allons nous intéresser toujours à une liste triée mais, désormais la liste n'est plus forcément une liste de nombres consécutifs.....

a) Comment repérer les éléments de cette liste et surtout comment repérer l'élément du milieu de la liste ?

Par leur position, donnée par leur indice.

b) Faire fonctionner le même principe A LA MAIN afin de savoir si la valeur  $v = 81$  est dans la liste triée suivante : list = [2 , 7 , 19 , 26 , 45 , 70 , 81 , 85 , 86 , 97 ]

ind	0	1	2	3	4	5	6	7	8	9
list	2	7	19	26	45	70	81	85	86	97

1 -  $m = \text{int}((0+9)/2) = 4$  , list[4]=45  $\neq$  81 comme list[m] < 81, alors deb = m+1

2 -  $m = \text{int}((5+9)/2) = 7$  , list[7]=85  $\neq$  81 comme list[m] > 81, alors fin = m-1

3 -  $m = \text{int}((5+6)/2) = 5$  , list[5]=70  $\neq$  81 comme list[m] < 81, alors deb = m+1

4 -  $m = \text{int}((6+6)/2) = 6$  , list[6]=81 = 81 on renvoie 6 (indice de 81 dans list)

c) Recommencez seul avec  $v = 77$

1 -  $m = \text{int}((0+9)/2) = 4$  , list[4]=45  $\neq$  77 comme list[m] < 77, alors deb = m+1

2 -  $m = \text{int}((5+9)/2) = 7$  , list[7]=85  $\neq$  77 comme list[m] > 77, alors fin = m-1

3 -  $m = \text{int}((5+6)/2) = 5$  , list[5]=70  $\neq$  77 comme list[m] < 77, alors deb = m+1

4 -  $m = \text{int}((6+6)/2) = 6$  , list[6]=81  $\neq$  77 comme deb=fin, l'élément v n'est pas dans list : on renvoie FAUX

### Fiche d'identité de l'algorithme de RECHERCHE DICHOTOMIQUE

**Principe** : diviser pour régner

La méthode diviser pour régner consiste en trois étapes :

DIVISER : découper le problème initial en sous problèmes ;

RÉGNER : résoudre les sous problèmes

COMBINER : calculer une solution au problème initial à partir des solutions des sous problèmes

Dans un premier temps, la recherche par dichotomie va en fait trouver LA POSITION d'un élément dans la liste triée : il faut comparer l'élément avec la valeur de la case au milieu du tableau ; si les valeurs sont égales, la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente

**Pseudo Code** : à compléter

**Pseudo Code** : à compléter

$v \leftarrow$  élément cherché

deb  $\leftarrow$  indice du premier élément de la liste

fin  $\leftarrow$  indice du dernier élément de la liste

$m \leftarrow$  arrondi ou troncature de  $(\text{deb} + \text{fin})/2$

Tant que deb  $\leq$  fin faire

si  $v = \text{liste}[m]$  alors

renvoyer m

sinon si  $v < \text{liste}[m]$  alors

fin  $\leftarrow m - 1$

sinon

deb  $\leftarrow m + 1$

fin si

$m \leftarrow$  arrondi ou troncature de  $(\text{deb} + \text{fin})/2$

Fin Tant que

Renvoyer Faux

## Python à implémenter

```
1 def dichotomie(v, list):
2     deb = 0
3     fin = len(list)-1
4     m=int((deb+fin)/2)
5     while deb<=fin:
6         if list[m] ==v:
7             return m
8         elif list[m]>v:
9             fin = m-1
10        else :
11            deb = m+1
12            m=int((deb+fin)/2)
13    return False
```

#recherche la présence de l'entier v dans la liste triée list  
#indice de debut de liste initialisé à 0  
#indice de fin de liste initialisé au dernier indice de la liste  
# m est l'indice de l'élément du "milieu"  
# condition d'arrêt : la liste est parcourue  
# v est trouvé dans la liste

# on va s'occuper de la liste inférieure  
# on s'occupe de la liste supérieure

# on indique que l'élément n'a pas été trouvé

## 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) n'est pas trivial...

On cherche un variant de boucle dans une liste de  $n$  éléments

Il s'agit de démontrer que l'algorithme se termine dans tous les cas et qu'il n'y a pas de boucle infinie.

Dans ce cas précis, on peut choisir la suite  $(fin_i - deb_i)_{i \in \mathbb{N}}$  des indices de fin et début de la liste considérée.

La condition d'arrêt de la boucle TANT QUE utilisée dans l'algorithme est Tant que  $deb \leq fin$  mais on n'entre pas toujours dans la boucle :

Soit  $fin_i$  et  $deb_i$  avec  $i$  l'indice qui compte la  $i$ ème itération de la boucle :

$deb_0 = 0$  et  $fin_0 = N - 1 = len(liste) - 1$ .

On note alors  $m_i = \text{arrondi} \left( \frac{deb_i + fin_i}{2} \right)$

Si nous sommes à la  $k$ ème itération, il y a trois grandes possibilités :

- $v = \text{liste}[m_k]$ , l'algorithme se termine, car la boucle n'est pas parcourue.

- $deb_k < fin_k$  et  $v < \text{list}[m_k]$ , on "entre" dans la boucle :  $deb_{k+1} = deb_k$  et  $fin_{k+1} = m_k - 1$  :  
on a  $fin_{k+1} - deb_{k+1} < fin_k - deb_k$

- $deb_k < fin_k$  et  $v > \text{list}[m_k]$ , on "entre" dans la boucle :  $deb_{k+1} = m_k + 1$  et  $fin_{k+1} = fin_k$  :  
on a  $fin_{k+1} - deb_{k+1} < fin_k - deb_k$

On a donc chaque fois,  $fin_{k+1} - deb_{k+1} < fin_k - deb_k$  : la suite  $(fin_i - deb_i)_{i \in \mathbb{N}}$  est strictement décroissante.

Il existe donc un entier  $p$  de  $\mathbb{N}$  tel que :

- soit  $deb_p > fin_p$ , l'algorithme va se terminer car on ne parcourt pas la boucle et l'algorithme renvoie

« faux »

- soit  $v = \text{list}[m_p]$  avec  $m_p = \frac{(deb_p + fin_p)}{2}$ , l'algorithme va se terminer, car on parcourt la boucle et

l'algorithme renvoie  $m_p$

L'algorithme se termine donc toujours.

## Complexité en temps :

Le logarithme en base 2

DEFINITION : On le note  $\log_2$  et pour  $x \in \mathbb{R}$ ,  $\log_2(2^x) = x$

Lors d'une recherche dichotomique d'un élément  $v$  dans une liste triée « list » de longueur  $n$ , le nombre  $f$  de fois dans le cas le plus défavorable ( $v \notin \text{list}$ ) où la liste sera coupée en deux vérifie :  $\frac{n}{2^f} = 1 \Leftrightarrow 2^f = n \Leftrightarrow f = \log_2(2^n)$

La complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est  $O(\log_2(n))$  (ce qui est beaucoup mieux qu'un  $O(n^2)$  comme la séquentielle).

Tracez les courbes  $x \rightarrow \log_2(x)$  et  $x \rightarrow x^2$   
et comparer leurs croissances...

$y = x$  (en rouge),  $y = x^2$  (en bleu) et  $y = \log_2(x)$  (en vert)

L'algorithme de recherche dichotomique est donc plus efficace que l'algorithme de recherche séquentielle car pour tout  $x$ ,  $x > \log_2(x)$ . Néanmoins, il est à noter que la recherche dichotomique utilise un tableau TRIÉ, ce qui peut aussi avoir un coût en temps...

## Exercices

### Exercice 1 – ANALYSER

Combien de valeurs sont examinées lors d'un appel à la

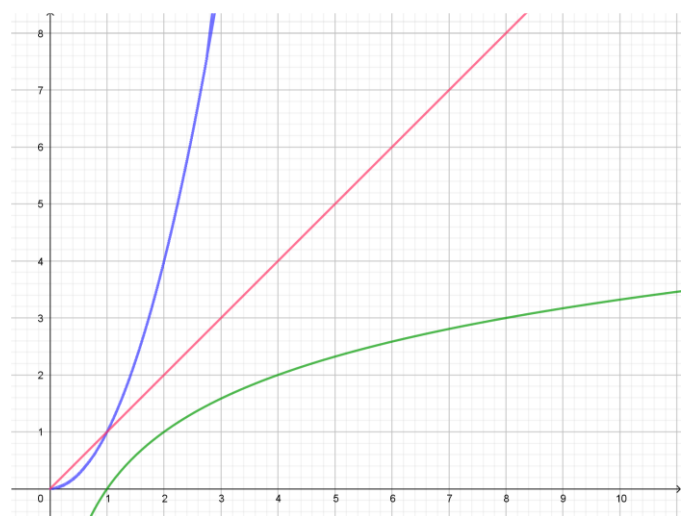
dichotomie sur la liste  $[0, 1, 2, 2, 3, 6, 9, 13, 13, 20]$  pour trouver 7 ? Et avec un appel à une recherche séquentielle ?

DICHOTOMIE 1 -  $m = \text{int}((0+9)/2) = 4$ ,  $\text{list}[4] = 3 \neq 7$  # 1ere valeur

comme  $\text{list}[m] < 7$ , alors  $deb = m + 1$

2 -  $m = \text{int}((5+9)/2) = 7$ ,  $\text{list}[7] = 13 \neq 7$  # 2ème valeur

comme  $\text{list}[m] > 7$ , alors  $fin = m - 1$



3 -  $m = \text{int}((5+6)/2) = 5$ ,  $\text{list}[5] = 6 \neq 7$  #3ème valeur      comme  $\text{list}[m] < 7$ , alors  $\text{deb} = m+1$   
4 -  $m = \text{int}((6+6)/2) = 6$ ,  $\text{list}[6] = 9 \neq 7$  #4ème valeur      comme  $\text{list}[m] > 7$ , alors  $\text{fin} = m-1$   
5 -  $m = \text{int}((6+5)/2) = 5$  et  $\text{deb} > \text{fin} \Rightarrow$  on sort de la boucle et on renvoie r donc FAUX  
On a examiné 4 valeurs.

### SEQUENTIELLE

C'est le pire des cas (7 n'est pas dans la liste) : on doit donc examiner toutes les valeurs une par une : il faut tester 10 valeurs ! C'est plus du double de la dichotomie !!

### Exercice 2 – TRADUIRE ET DÉVELOPPER

Écrire en Python une fonction Repetition (n) qui calcule et renvoie le plus petit entier f tel que  $2^f > n$ .

Rappeler ce que représente f dans une recherche dichotomique dans une liste de taille n.

f représente le plus nombre de fois (donc entier) pour lequel la division par 2 de n donne un résultat inférieur à 1 : si on divise le nombre d'éléments de la liste par 2 de manière répétitive, f représente le nombre de divisions pour lequel est on certain de n'avoir plus qu'un seul choix.



residus

tri selection

Exercice 2 – ANALYSER et TRADUIRE

On souhaite maintenant trier la liste de listes Pers selon le numéro situé en 2ème position dans les sous-listes que contient Pers :

```
Pers = [['Sam', 3], ['Julie', 1], ['Tom', 2], ['Portillon', 4], ['Charlie', 5]]
```

Mais cette fois ci, seul un élément est mal trié. Il s'agit de la liste ['Sam', 3] qui correspond à l'indice 0 dans la liste Pers . Proposer une fonction mettant en œuvre un algorithme de tri conçu par vous même permettant de trier la liste Pers.

On ne remplacera que l'élément mal placé, sans trier toute la liste.

( cette situation peut être illustrée par le rangement d'un livre selon son titre dans une bibliothèque dont les ouvrages sont classés par ordre alphabétique : on ne trie pas toute la bibliothèque mais on insère l'ouvrage à sa place)

*Aides : del Pers[0] supprime l'élément mal placé*

*Pers.insert(i,mal\_placee) insère la liste mal\_placee (à déterminer) à l'indice i (à déterminer)*