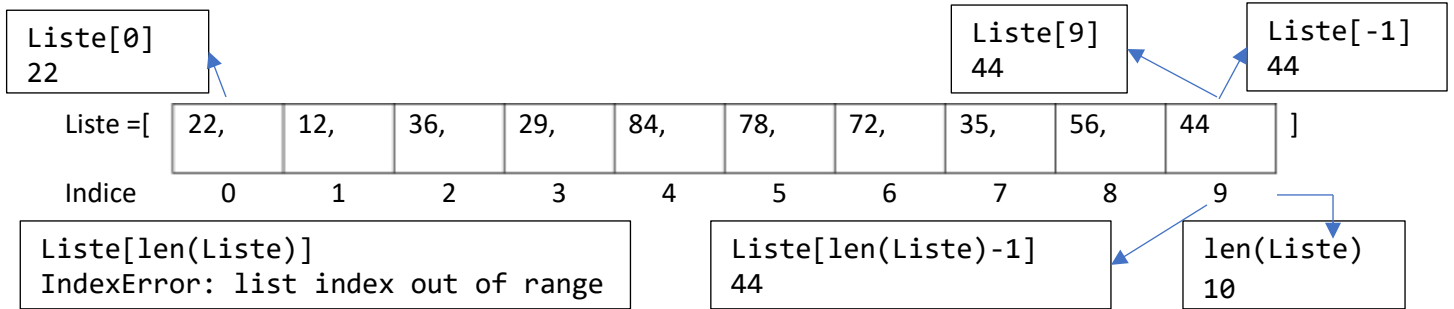


# 1 Prérequis

## 1.1 Parcours d'une liste

Ici tableau est utilisé dans le sens séquence (c'est-à-dire listes, tableaux Numpy, chaînes de caractères, tuples...)



### 1.1.1 Parcours d'une liste par indice

#### 1.1.1.1 Boucle for

```
for i in range(len(Liste)):
    print(Liste[i])
```

#### 1.1.1.2 Boucle while

```
i = 0 # initialisation de i
while i < len(Liste): # vérification
    print(Liste[i])
    i += 1 # incrémentation de i
```

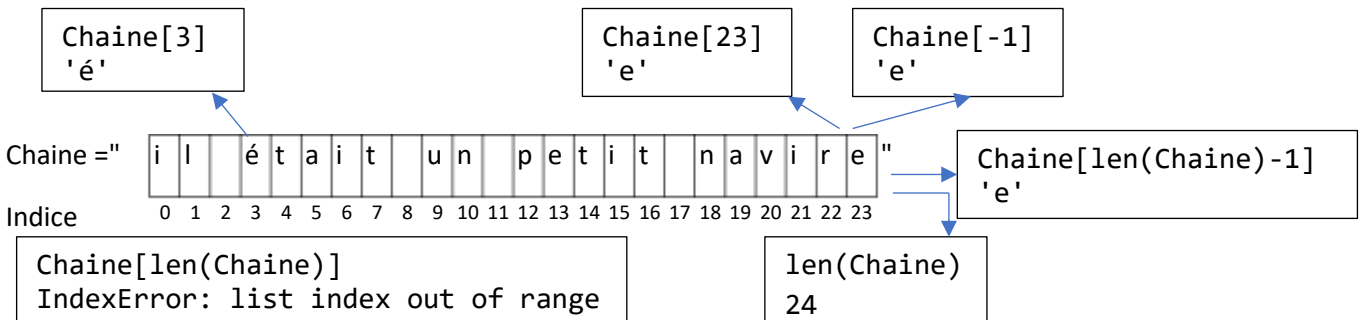
### 1.1.2 Parcours d'une liste par valeur

```
for element in Liste :
    print(element)
```

### 1.1.3 Parcours d'une liste par indice et valeur

```
for indice, element in enumerate(Liste):
    print(indice, element)
```

## 1.2 Parcours d'une chaîne de caractères



### 1.2.1 Parcours d'une chaîne de caractères par indice

#### 1.2.1.1 Boucle for

```
for i in range(len(Chaine)):
    print(Chaine[i])
```

#### 1.2.1.2 Boucle while

```
i = 0 # initialisation de i
while i < len(Chaine): # vérification
    print(Chaine[i])
    i += 1 # incrémentation de i
```

### 1.2.2 Parcours d'une liste par valeur

```
for element in Chaine :
    print(element)
```

### 1.2.3 Parcours d'une liste par indice et valeur

```
for indice, element in enumerate(Chaine):
    print(indice, element)
```

## 2 Exercices

Les exercices de ce TP doivent être enregistrés dans un dossier 4\_unidimensionnel.

### 2.1 Déterminer un extremum dans une liste

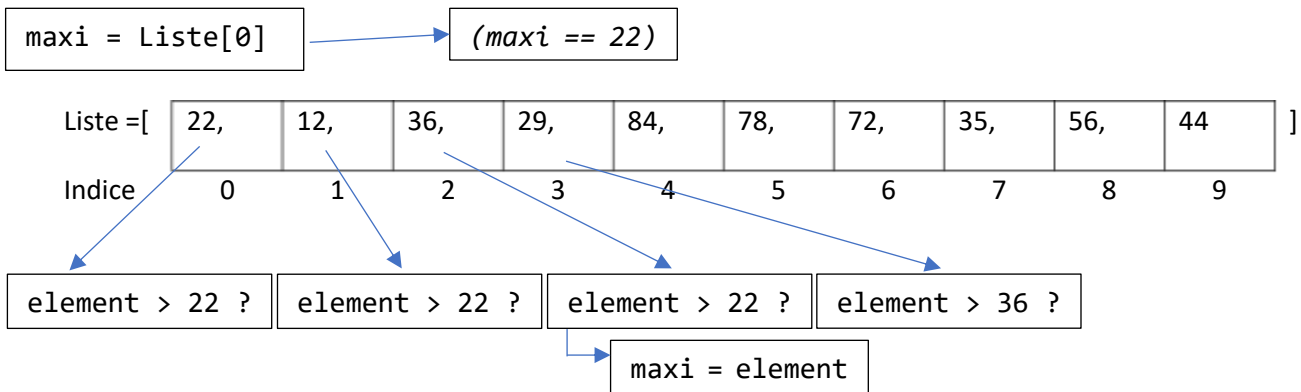
#### 2.1.1 Cahier des charges

Écrire une fonction `maximum` qui détermine la valeur maximale contenue dans une liste.

#### 2.1.2 Algorithme

Une variable `maxi` est initialisée à la valeur du premier élément de la liste. Chacun des éléments de la liste est ensuite tour à tour comparé à la variable `maxi`. Lorsque l'élément comparé est supérieur à la variable `maxi`, la variable `maxi` prend la valeur de cet élément.

**Exemple :**



#### 2.1.3 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous 4\_unidimensionnel /recherche\_maxi.py et tester.

#### 2.1.4 Implémentation en Python

```
def maximum(List):  
    maxi=List[0]  
    for element in List:  
        if element > maxi:  
            maxi=element  
    return maxi
```

maximum(Liste)  
84

Remarque : la première comparaison se fait avec l'élément qui se trouve en position 0. Or, la variable `maxi` ayant été affectée à cette valeur, cette première comparaison est inutile. Il est possible de sauter cette étape en utilisant le slicing :

```
for element in List[1:]
```

#### 2.1.5 Complexité

Les programmes `liste_aleatoire.py` et `temps.py` ci-dessous vous sont fournis dans le dossier `info_commune_ouils`.

```
import time  
  
#départ chrono  
t1=time.time() #nombre de secondes écoulées  
depuis le 1er janvier 1970  
#exécution de l'algorithme  
  
#fin chrono  
t2=time.time()  
#temps  
difference=t2-t1  
print(difference)
```

```
from random import randrange  
  
def liste_aleatoire(n):  
    #création d'une liste remplie  
    #de 0 et de taille n  
    liste=[0]*n  
    #remplacement des 0 par des  
    #valeurs prises aléatoirement  
    #entre 0 et n  
    for i in range(n):  
        liste[i]=randrange(1,n)  
    return liste
```

- Écrire ci-dessous le code permettant de déterminer les temps de recherche de la valeur maximale pour des listes de taille donnée. Enregistrer sous 4\_unidimensionnel/recherche\_maxi\_complexite.py
- Exécuter pour des listes de taille  $10^7$ ,  $2 \cdot 10^7$ ,  $4 \cdot 10^7$ ,  $6 \cdot 10^7$  et  $8 \cdot 10^7$ ,
- Compléter le tableau suivant.

Taille de la liste	$10^7$	$2 \cdot 10^7$	$4 \cdot 10^7$	$6 \cdot 10^7$	$8 \cdot 10^7$
Temps d'exécution en secondes	0.376	0.686	1.35	1.94	2.94

- Conclure.

La complexité consiste en l'étude formelle des performances d'un algorithme. Il existe 2 types de complexité :

- La complexité temporelle,
- La complexité d'espace mémoire.

Lors d'une recherche séquentielle dans un tableau unidimensionnel, la complexité temporelle est dite linéaire :  $O(n)$ .

**Remarque** : vous avez peut-être remarqué que la même mesure ne donne pas toujours le même résultat. Si on relance plusieurs fois le programme, la valeur mesurée va probablement varier, notamment selon l'occupation de l'ordinateur. Une solution consisterait à faire plusieurs fois la même mesure afin d'en tirer la moyenne.

#### #fonction à tester

```
def maximum(List):
    maxi=List[0]
    for element in List:
        if element > maxi:
            maxi=element
    return maxi
```

#### #fonction de création des listes

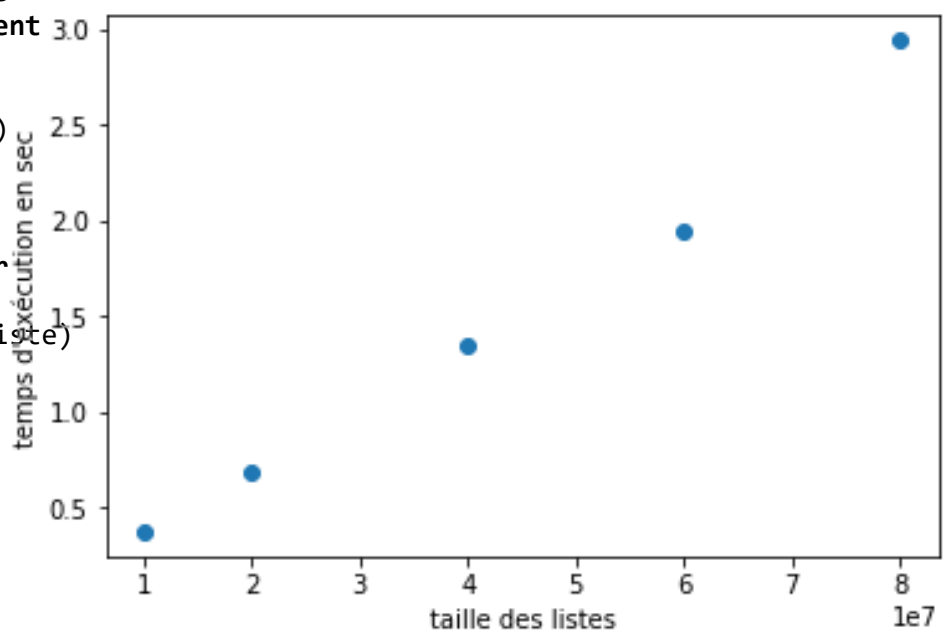
```
from random import randrange
def liste_aleatoire(n):
    #création d'une liste remplie
    #de 0 et de taille n
    liste=[0]*n
    #remplacement des 0 par des
    #valeurs prises aléatoirement
    #entre 0 et n
    for i in range(n):
        liste[i]=randrange(1,n)
    return liste
```

#### #PROGRAMME PRINCIPAL

```
##création de la liste à tester
taille_liste=2*10**7
liste=liste_aleatoire(taille_liste)
##temps de parcours
import time
###départ chrono
t1=time.time()
###exécution de l'algorithme
maximum(liste)
###fin chrono
t2=time.time()
###temps
difference=t2-t1
print(difference)
```

#### #tracé

```
import matplotlib.pyplot as plt
taille=[10**7,2*10**7,4*10**7,6*10**7,8*10**7]
temps=[0.376,0.686,1.35,1.94,2.94]
plt.plot(taille,temps,'o')
plt.ylabel("temps d'exécution en sec")
plt.xlabel('taille des listes')
plt.show()
```



### 3 Recherche séquentielle dans un tableau unidimensionnel : à retenir

- Algorithme dans lequel il y a 1 seule boucle d'instructions itératives (inconditionnelles `for` ou conditionnelles `while`),
- Dans cette boucle d'instructions itératives, il y a en général une instruction conditionnelle `if` ou une expression mathématique ou une affectation de variable,
- Complexité temporelle linéaire :  $O(n)$ .

## 4 Exercices d'application

### 4.1 Calculer la somme et la moyenne des valeurs contenues dans une liste

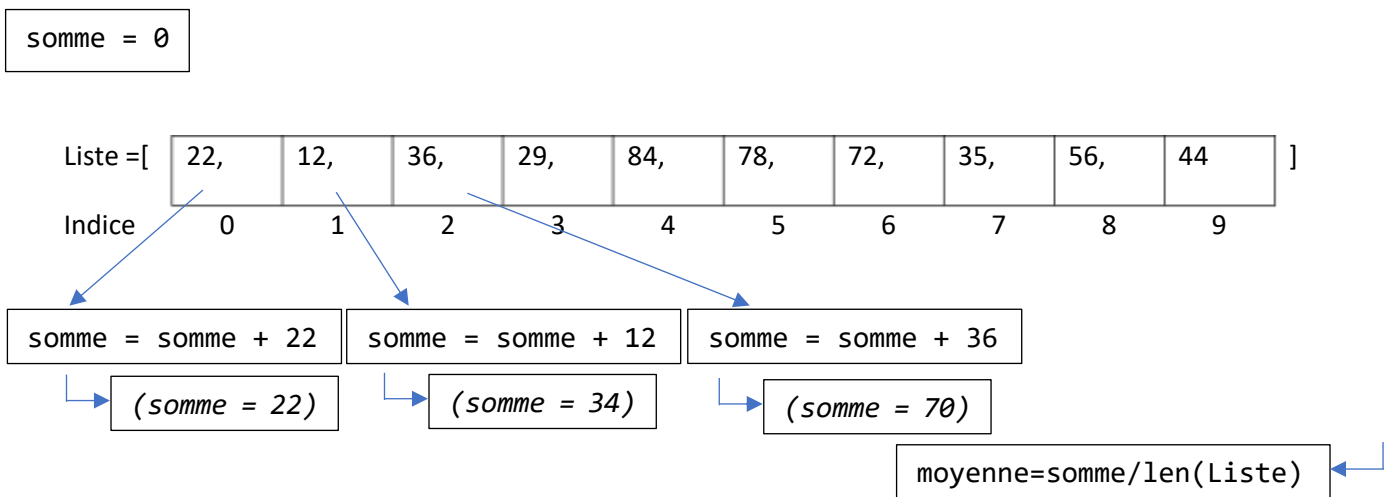
#### 4.1.1 Cahier des charges

Écrire une fonction `sommeEtMoyenne` qui détermine la somme des éléments contenus dans une liste et en fait la moyenne.

#### 4.1.2 Algorithme

Une variable `somme` est initialisée à 0. Chacun des éléments de la liste est ensuite parcouru tour à tour. À chaque itération, la valeur de l'élément est ajoutée à la variable `somme`. La moyenne est ensuite calculée.

**Exemple :**



#### 4.1.3 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous `4_unidimensionnel /somme_moyenne` et tester.

#### 4.1.4 Implémentation en Python

```
def sommeEtMoy(Liste):  
    somme=0  
    for element in Liste:  
        somme=somme+element  
    moyenne=somme/len(Liste)  
    return somme,moyenne
```

Result: `sommeEtMoy(Liste): (468, 46.8)`

## 4.2 Déterminer l'indice d'un élément dans une liste

### 4.2.1 Cahier des charges 1

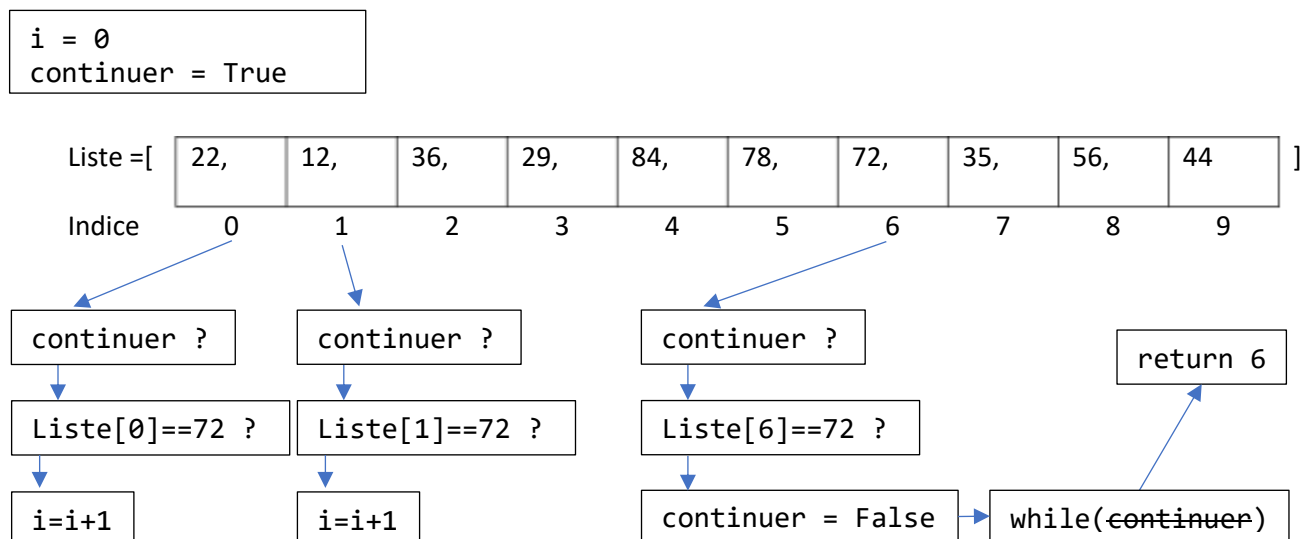
Écrire une fonction recherche qui prend 2 arguments et qui détermine l'indice d'un élément présent dans une liste.

### 4.2.2 Algorithme 1

Un « pointeur »  $i$  représentant l'indice est initialisé à 0. À chaque itération, l'élément d'indice  $i$  est comparé avec l'élément recherché. Les itérations continuent tant que le résultat de cette comparaison ne donne pas satisfaction. Si l'élément d'indice  $i$  et l'élément recherché sont égaux, la boucle s'arrête, sinon, le « pointeur »  $i$  est incrémenté.

Pour arrêter une boucle `while`, il est commode d'utiliser un booléen.

**Exemple** : recherche de 72.



### 4.2.3 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous `4_unidimensionnel/recherche_element` et tester.

### 4.2.4 Implémentation en Python

```
def recherche(element, List):  
    i=0  
    continuer=True  
    while continuer:  
        if List[i]==element:  
            continuer=False  
        else:  
            i=i+1  
    return i
```

`recherche(72, Liste)`  
6

Remarque : l'algorithme ci-dessous est plus court et renvoie le même résultat. Il est cependant plus difficilement exploitable pour le cahier des charges 2.

```
def recherche(element, List):  
    i=0  
    while List[i]!=element :  
        i=i+1  
    return i
```

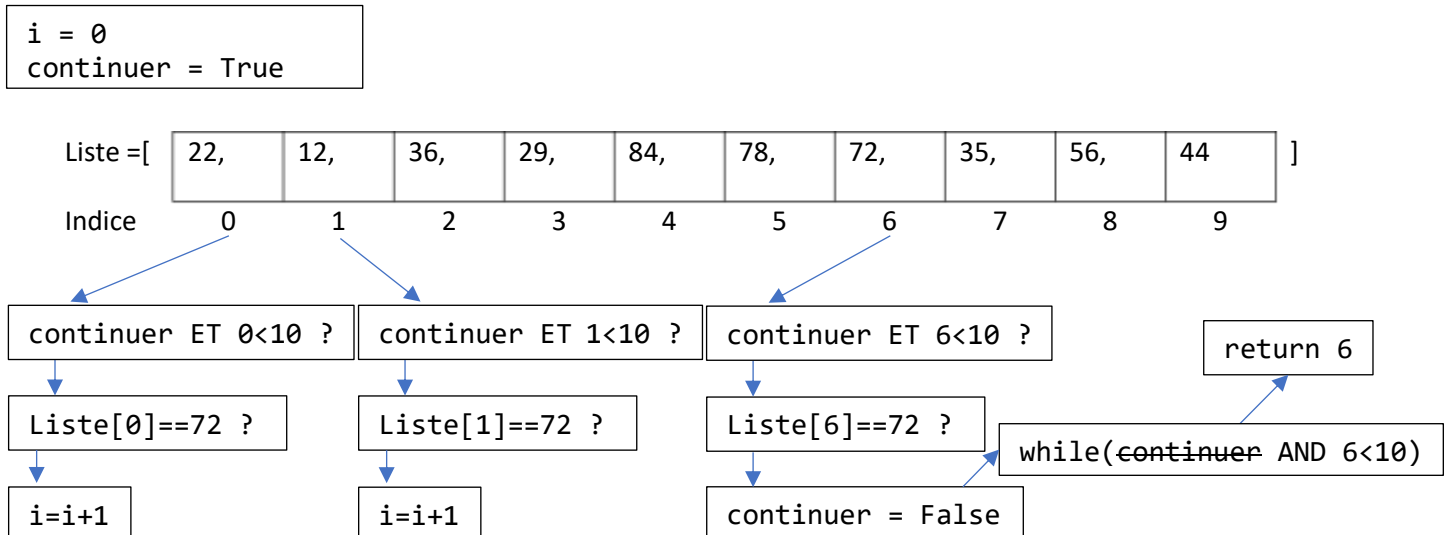
#### 4.2.5 Cahier des charges 2

Écrire une fonction recherche2 qui détermine l'indice d'un élément présent dans une liste. Si l'élément n'est pas présent dans la liste, le pointeur va jusqu'au bout de la liste et finit par prendre la valeur de la longueur de la liste.

#### 4.2.6 Algorithme 2

Les itérations continuent tant que le résultat de cette comparaison ne donne pas satisfaction et que l'indice i n'est pas au bout de la liste.

**Exemple** : recherche de 72.



#### 4.2.7 Activité

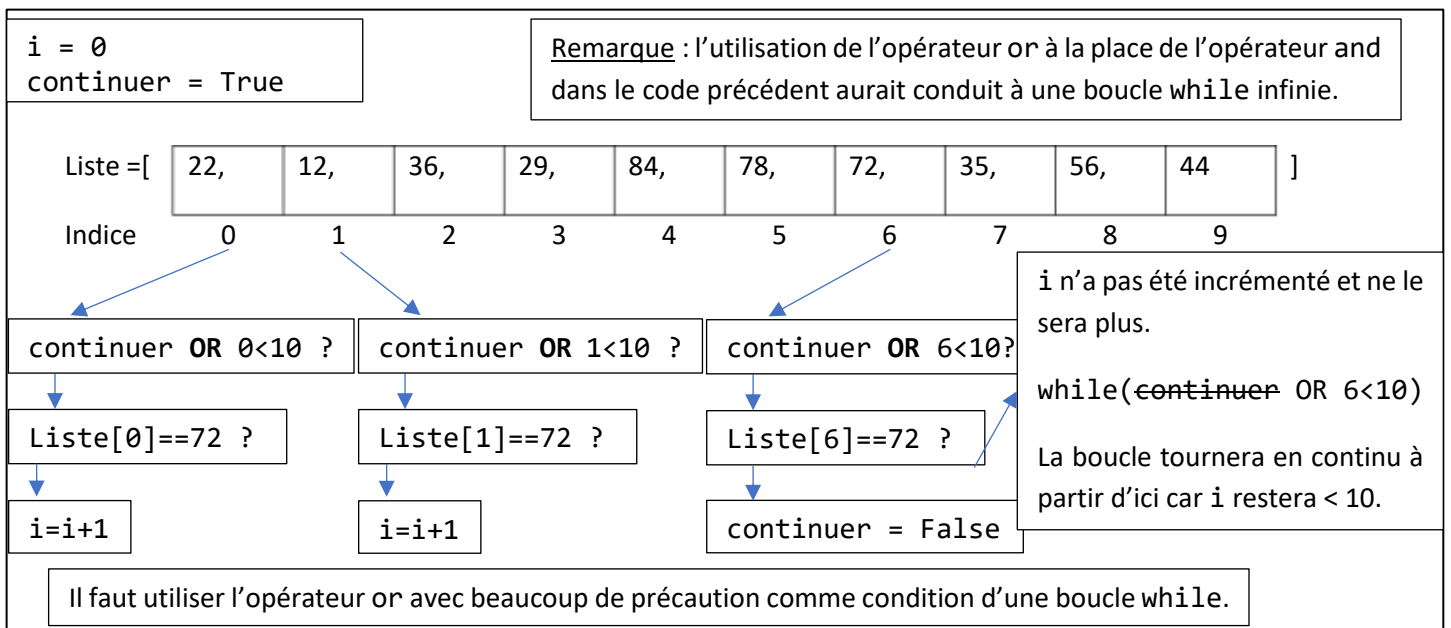
Écrire ci-contre le code demandé. Saisir sous python, enregistrer sous 4\_unidimensionnel/recherche\_element2 et tester.

#### 4.2.8 Implémentation en Python

```
def recherche2(element,List):
    i=0
    continuer=True
    while continuer and i<len(List) :
        if List[i]==element:
            continuer=False
        else:
            i=i+1
    return i
```

recherche2(72,Liste)  
6

recherche2(99,Liste)  
10



## 4.3 Compter le nombre d'occurrences dans une chaîne de caractères

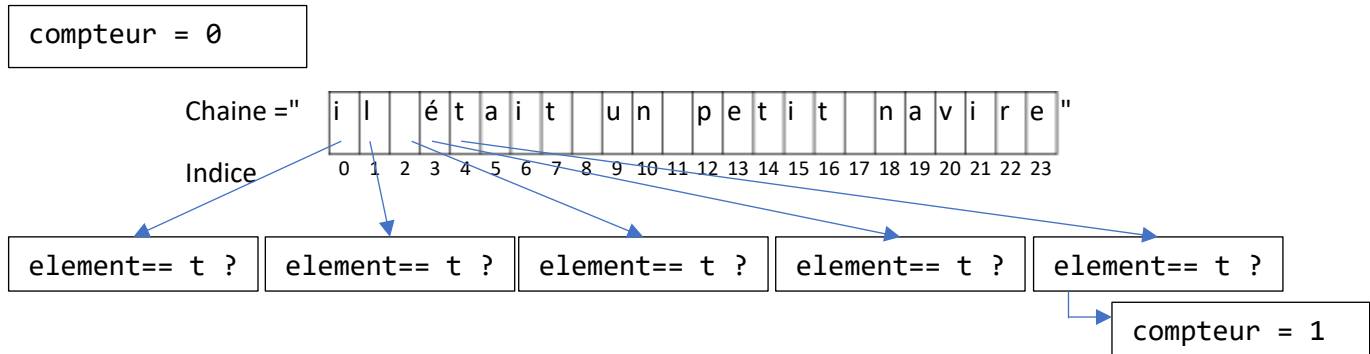
### 4.3.1 Cahier des charges 1

Écrire une fonction `compter` qui prend 2 arguments et qui permet de compter le nombre de fois qu'un caractère apparaît dans une chaîne de caractères.

### 4.3.2 Algorithme 1

Une variable `compteur` est initialisée à zéro. Chacun des éléments de la liste est ensuite tour à tour comparé au caractère à compter. Lorsque l'élément comparé est le même que le caractère à compter, la variable `compteur` est incrémentée.

**Exemple :** compter le nombre de fois que `t` apparaît dans la chaîne de caractères.



La structure de l'algorithme est la même que la structure d'un algorithme de recherche d'un extremum.

### 4.3.3 Activité

Écrire ci-contre le code demandé. Saisir sous python, enregistrer sous 4\_unidimensionnel/compter\_occurrences et tester.

### 4.3.4 Implémentation en Python

```
def compter(String, caractere) :  
    compteur=0  
    for element in String:  
        if element == caractere :  
            compteur=compteur+1  
    return compteur
```

compter(Chaîne, 't')  
4

### 4.3.5 Cahier des charges 2

Écrire une fonction `compter_pos` qui permet de compter le nombre de fois qu'un caractère apparaît dans une chaîne de caractères et qui consigne la position desdits caractères dans une liste.

### 4.3.6 Algorithme 2

Une variable `compteur` est initialisée à zéro. Une liste `position` vide est créée. Chacun des éléments de la liste est ensuite tour à tour comparé au caractère à compter. Lorsque l'élément comparé est le même que le caractère à compter, son indice de position est ajouté à la liste `position` et la variable `compteur` est incrémentée.

### 4.3.7 Activité

Écrire ci-contre le code demandé. Saisir sous python, enregistrer sous 4\_unidimensionnel/compter\_occurrences2 et tester.

### 4.3.8 Implémentation en Python

```
def compter_pos(String, caractere):  
    compteur=0  
    position=[]  
    for i in range(len(String)):  
        if String[i]==caractere:  
            compteur=compteur+1  
            position.append(i)  
            i=i+1  
    return compteur, position
```

compter\_pos(Chaîne, 't')  
(4, [4, 7, 14, 16])

Remarque : dans l'exemple précédent, le parcours est fait par valeur. Ici, le parcours doit être fait par indice.

## 4.4 Séparer une liste

### 4.4.1 Cahier des charges

Écrire une fonction `separer` qui prend deux arguments et qui permet, à partir d'une liste de nombres, d'obtenir deux listes. La première comporte les nombres inférieurs ou égaux à un nombre séparateur donné, la seconde les nombres qui lui sont strictement supérieurs.

### 4.4.2 Algorithme

Deux listes vides sont créées : `listeInferieure` et `listeSuperieure`. Chacun des éléments de la liste à séparer est ensuite tour à tour comparé au nombre séparateur. Si l'élément est inférieur ou égal au nombre séparateur, il est ajouté en fin de `listeInferieure`. Dans le cas où il est supérieur, il est ajouté en fin de `listeSuperieure`.

### 4.4.3 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous `4_unidimensionnel/separer_liste` et tester.

### 4.4.4 Implémentation en Python

```
def separer(List,separateur):
```

```
    listeInferieure=[]
```

```
    listeSuperieur=[]
```

```
    for nombre in List:
```


```
        if nombre<=separateur:
```

```
            listeInferieure.append(nombre)
```

```
        else :
```

```
            listeSuperieur.append(nombre)
```

```
    return listeInferieure,listeSuperieur
```



```
separer(Liste,48)  
([22, 12, 36, 29, 35, 44], [84, 78, 72, 56])
```

La structure de cet algorithme semblable à la structure des algorithmes de recherche d'un extremum et de comptage d'un nombre d'occurrences dans une liste.



## 4.5 Recherche de deux valeurs les plus proches dans une liste

### 4.5.1 Cahier des charges 1

Écrire une fonction `plus_proche` qui permet de rechercher dans une liste la plus proche valeur d'un nombre à approcher donné en argument.

### 4.5.2 Algorithme 1

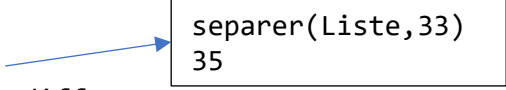
La structure de cet algorithme semblable à la structure des algorithmes de recherche d'un extremum, au comptage d'un nombre d'occurrences dans une liste et à la séparation d'une liste.

### 4.5.3 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous `4_unidimensionnel/plus_proche` et tester.

### 4.5.4 Implémentation en Python

```
def plus_proche(List,NbApprocher):  
    diff=abs(List[0]-NbApprocher)  
    for nombre in List:  
        if abs(nombre-NbApprocher)<=diff:  
            diff=abs(nombre-NbApprocher)  
            PlusProche=nombre  
    return PlusProche
```



```
separer(Liste,33)  
35
```

### 4.5.5 Cahier des charges 2

Écrire une fonction `plus_proche_pos` qui permet de rechercher dans une liste la plus proche valeur d'un nombre à approcher donné en argument et qui consigne sa position dans une liste.

### 4.5.6 Algorithme 2

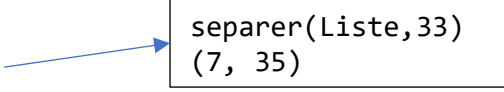
Pour cet algorithme, s'inspirer des deux premiers exemples concernant le comptage d'un nombre d'occurrences dans une liste.

### 4.5.7 Activité

Écrire ci-dessous le code demandé. Saisir sous python, enregistrer sous `4_unidimensionnel/plus_proche_2` et tester.

### 4.5.8 Implémentation en Python

```
def plus_proche_pos(List,NbApprocher):  
    diff=abs(List[0]-NbApprocher)  
    for i in range(len(List)):  
        if abs(List[i]-NbApprocher)<=diff:  
            diff=abs(List[i]-NbApprocher)  
            plusProche=List[i]  
            indicePlusProche=i  
    return indicePlusProche,plusProche
```



```
separer(Liste,33)  
(7, 35)
```