Remarque : les ex-MPSI avaient un cahier un plus complet avec notamment les parcours de graphes, mais **pour le test de jeudi il n'y aura rien sur les graphes**, nous nous en tenons aux scripts qui sont corrigés ci-dessous.

## Quelques corrections

### 1/. Rechercher un élément dans une liste

```
def cherche(L, elt) -> bool :

    """L : liste de valeurs (de tous types), elt : élément à chercher
    -> booléen indiquant si elt est présent dans L """

    for k in L :
        if k == elt :          # élément trouvé !
            return True
    return False               # élément absent de la liste
```

Complexité : Parcours de liste en $O\big(len(L)\big)$ et le corps de boucle (test) est en temps constant $O(1)$ (indépendant de len(L)), donc complexité en $O\big(len(L)\big)$.

```
def cherche_positions(L,elt) -> [int] :

    """L : liste d'éléments (de tous types), elt : élément à chercher
    -> liste des positions de cet élément dans L, et liste vide si
    élément absent de L """

    pos = []
    for k in range(len(L)) :
        if L[k] == elt :         # élément trouvé !
            pos.append(k)
    return L
```

Complexité :
Affectation, test et ajout avec `append` : complexité en $O(1)$.
Le coût est donné par le nombre d'itérations de la boucle donc $O\big(len(L)\big)$.

## 2/. Faire la somme des éléments d'une liste (sans `sum`)

```
def somme(L :[int]) -> int :

    """L : liste de valeurs numériques
    -> somme des éléments de L """

    S = 0
    for k in L :
        S = S + k
    return S
```

Complexité : Soit $n = len(L)$.
Une bouclé `for` réalisée $n$ fois, le reste étant en temps constant (indépendant de len(L)).
Donc **complexité linéaire $O(n)$.**

## 3/. Compter le nombre d'éléments d'une liste qui vérifient un critère donné

```
def compteur(L)->int :

    """L : liste de valeurs (ici nombres)
    -> nombre de valeurs positives dans L (par exemple) """

    cpt = 0
    for k in L :
        if k >= 0 :              # condition vérifiée
            cpt = cpt + 1        # on incrémente le compteur
    return cpt
```

Complexité : soit $n = len(L)$

Une boucle for réalisée $n$ fois, le reste étant en temps constant (affectations, comparaison, somme). Complexité linéaire en la longueur de la liste, donc $O(n)$.

## 4/. Rechercher le maximum minimum dans une liste

```
def maxi(L) :

    """L : liste de valeurs
    -> maximum des éléments et la position d'une occurrence du max """

    M = L[0]        # premier terme de la liste
                    # éventuellement float('inf') dans certains exos
    pos = 0
    for k in range(len(L)) :
        if L[k] > M :
            M = L[k]      # nouveau maximum temporaire trouvé
            pos = k       # indice mémorisé
    return M, pos
```

Complexité : comparaisons, affectations s'exécutent en temps constant donc $O(1)$ (c'est-à-dire indépendant de len(L))
Le coût vient du nombre de fois où la boucle est exécutée, soit $len(L)$ fois.
Complexité en $O\big(len(L)\big)$

```
def maxi2(L) -> [int] :

    """L : liste de valeurs comparables
    -> liste des positions du maximum dans L, en un seul parcours """

    pos = []
    M = L[0]
    for k in range(len(L)) :
        if L[k] > M :
            M = L[k]          # nouveau maximum trouvé et mémorisé
            pos = [k]         # on écrase les anciennes positions, on recommence
        elif L[k] == M :      # autre apparition du maximum temporaire
            pos.append(k)     # position mémorisée
    return M, pos
```

Complexité : La boucle for est exécutée $len(L)$ fois et le corps de boucle est en temps constant (affectations, comparaisons, ajout dans une liste, donc opérations indépendantes de len(L) ).
Complexité en $O(len(L))$.

## 5/. Rechercher le second maximum dans une liste

```
def second_maxi(L) :

    """L : liste de cvaleurs comparables
    -> les deux premiers maxima de la liste (éventuellement égaux) """

    assert len(L) >=2     # au moins deux éléments dans la liste !

    # initialisations
    if L[0] >= L[1] :
        M1, M2 = L[0],L[1]     # M1 le plus grand, M2  le 2e plus grand
    else :
        M1, M2 = L[1], L[0]

    for k in L :
        if k > M1 :            # encore plus grand que le + grand mémorisé
            M1, M2 = k, M1     # on mémorise et l'ancien + grand devient le 2e plus grand
        elif k > M2 :          # donc k entre M1 et M2
            M2 = k
    return M1, M2
```

Complexité :
Initialisations en $O(1)$ temps constant, c'est-à-dire indépendant de `len(L)` (comparaisons et affectations)
Boucle réalisée $len(L)$ fois et le corps de boucle s'exécute en temps constant (comparaisons et affectations, donc indépendant de len(L))
Donc la complexité est en $O\big(len(L)\big)$.

## 6/. Comptage des éléments d'un tableau à l'aide d'un dictionnaire

```
def comptage(L : list) -> dict :

    """L : liste de valeurs (pas forcément comparables)
    -> dictionnaire ayant pour clés les éléments distincts de L et pour valeurs
    le nb d'occurrences de chacune de ces valeurs

    Rappel : aucun ordre dans un dictionnaire !!! """

    d = {k : 0 for k in L}    # aucun risque de doublon sur les clés (structure d'ensemble)

    # remplissage du dictionnaire :
    for k in L :
        d[k] = d[k] + 1       # valeur k rencontrée une fois de plus
    return d
```

Complexité :

- Remplissage **initial** du dictionnaire par compréhension : $O\big(len(L)\big)$
- Compléter le dictionnaire en $O\big(len(L)\big)$ car un parcours de L et une affectation en $O(1)$

Donc complexité $= O(n) + O(n) = 2\,O(n) = O(n)$ avec $n = len(L)$.

## 7/. Recherche d'un facteur dans un texte

Version sans slicings :

```
def cherche_mot(T ,mot) -> [int] :

    """ T : texte ou liste de valeurs
    mot : texte ou liste de valeurs
    -> liste des positions (initiales) de toutes les occurrences de mot dans T """

    assert len(T) >= len(mot)    # sinon aucun intérêt de faire la recherche

    pos = []
    for k in range(len(T) - len(mot) +1) :  # on teste les positions de départ possibles

        # recherche du mot à partir de la position k
        cpt = 0                             # nb de caractères communs
        for i in range(len(mot)) :          # sur tout la taille du petit mot
            if T[k+i] == mot[i] :           # caractères égaux
                cpt = cpt + 1               # un caractère commun en plus

        if cpt == len(mot) :                # mot trouvé !
            pos.append(k)                   # ajout de la position de la 1ère lettre du mot
                                            # dans le texte
    return pos
```

Complexité dans le pire des cas :
On note $n = len(T)\ et\ \ m = len(mot)$.
Boucle externe réalisée $n - m$ fois.

Pour chacune de ces fois, une boucle réalisée m fois et des opérations à coût constant

Une comparaison de `cpt` et un ajout en temps constant

Complexité globale : $O(n-m) \times \big(O(m) + \ O(1)\big) = O(n-m) \times O(m)$
Donc complexité en $O\big((n-m) \times m\big)$.

Version avec slicings :

```
def cherche_mot2(T, mot) -> [int] :

    """T : texte ou liste de valeurs
    mot : texte ou liste de valeurs
    -> liste des positions de départ de toutes les occurrences de mot dans T"""

    assert len(T) >= len(mot)

    pos = []
    for k in range(len(T) - len(mot) +1) :
        if T[k : k+len(mot)] == mot :
            pos.append(k)
    return pos
```

## 8/. Recherche des deux valeurs les plus proches dans un tableau

```
def deux_val_proches(L) -> list :

    """L : liste de valeurs comparables
    -> les deux valeurs les plus proches et leur distance, et leurs positions """

    assert len(L) >= 2    # sinon aucun intérêt au problème

    dist_min = float('inf')           # ou abs(L[1]-L[0])
    pos = []                          # ou pos = [0,1]

    for k in range(len(L)) :
        for i in range(k+1, len(L)) : # on envisage tous les doublons mais
                                      # tout en limitant le parcours ; on évite
                                      # de comparer un élément avec lui même car
                                      # l'écart ferait évidemment 0
            if abs(L[k]-L[i]) < dist_min :
                dist_min = abs(L[k]-L[i])
                pos = [i,k]
    return dist_min, [L[pos[0]], L[pos[1]]], pos
           # ecart mini, valeurs associées, positions associées
```

Complexité :

Le coût est donné par les deux boucles imbriquées, puisque les autres instructions (structure conditionnelle, comparaison, affectations) sont en temps constant $O(1)$ (temps indépendant de $len(L)$). Reste à savoir combien il y aura d'itérations…

Boucle externe réalisée `n = len(L)` fois.

A la première itération (lorsque $k = 0$), on fait $n - 1$ itérations,

A la 2e (lorsque $k = 1$), on fait $n - 2$ itérations,

…. Et à la dernière, lorsque $k = len(L) - 1$, on fait 0 itérations.

Soit un total de $\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$ itérations.

La complexité est donc en $O(n^2)$, complexité quadratique en la taille de la liste.

## 9/. Algorithmes de tri quadratiques (en $O(len(L)^2)$)

a/.

```
def tri_bulles(L)-> None :

    """L : liste de valeurs comparables
    -> None ;
    tri de la liste dans l'ordre croissant ; la liste L sera modifiée
    en place """

    for i in range(len(L)-1) :
        for k in range(len(L)-i-1) :
            if L[k+1] < L[k] :                  # pas dans le bon ordre
                L[k+1],L[k] = L[k], L[k+1]      # on les permute


# tests
from randint import randint
L_test = [randint(-10,10) for k in range(10)]
print(L_test)                                   # liste d'entiers pseudi-aléatoires
tri_bulles(L_test)                              # on trie la liste
print(L_test)                                   # on vérifie ...
```

Complexité : notons $n = len(L)$

- **Dans le pire des cas, la complexité est quadratique $= O(n^2)$ :**

  En effet, dans le pire des cas, la liste est initialement triée dans l'ordre décroissant ; chaque paire d'éléments consécutifs est dans le mauvais ordre, donc chaque comparaison entraîne une permutation.
  - La boucle externe s'exécute $n-1$ fois.
  - Pour chaque itération $i$, la boucle interne s'exécute $n-i-1$ fois
  - Le nombre total de comparaisons est donc :

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} \in O(n^2)$$

- **Dans le meilleur des cas, la complexité est quadratique , $O(n^2)$.**

  Le meilleur des cas est lorsque la liste est déjà triée ; la version ci-dessus n'est pas optimisée ; on pourrait écrire une version qui détecte quand le tri est terminé (un parcours sans la moindre modification), et on gagnerait alors en calculs. Ici, si la liste est déjà triée, toutes les comparaisons seront effectuées, sans faire les permutations. Donc on effectue autant de comparaisons que dans le pire des cas. Donc complexité en $O(n^2)$.

b/. Tri par insertion

```python
def tri_insertion(L)  -> None :

    """L : liste de valeurs comparables
    -> None ; tri en place de la liste L par ordre croissant """

    for k in range(1, len(L)) :
        x = L[k]                        # élément à placer
        j = k

        # on cherche la place de x parmi le début de la liste, triée.
        while j > 0 and x < L[j-1] :    # tant que pas au début de la liste, et
                                        # que la future place de x n'est pas trouvée
            L[j] = L[j-1]               # on décale le terme d'avant
            j = j-1                     # on décale vers la gauche le compteur
                                        # qui cherche la future place de x
        L[j] = x                        # on place x dans la position libérée
```

Complexité : notons $n = len(L)$

- **Dans le pire des cas, la complexité est quadratique $= \boldsymbol{O(n^2)}$ :**

  Pire des cas quand la liste est initialement triée dans l'ordre décroissant ; donc chaque nouvel élément doit être comparé à tous ceux d'avant et inséré au début.
  - A chaque itération il y a $k$ comparaisons et $k$ décalages.
  - Le nombre total d'opérations est donc :

$$\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} \in O(n^2)$$

- **Dans le meilleur des cas, la complexité est linéaire , $\boldsymbol{O(n)}$.**

  Le meilleur cas se produit lorsque la liste est déjà triée. Dans ce cas :
  - Condition `x < L[j-1]` toujours fausse, donc le `while` ne s'exécute jamais.
  - Il y a donc exactement $n - 1$ comparaisons, et aucun décalage.

  Donc une complexité en $O(n)$.

- **Complexité moyenne** : $O(n^2)$

## 10/. Recherche dichotomique dans un tableau trié

```
def cherche_dicho(L,a) :

    """L : liste triée dans l'ordre croissant, a: élément à chercher
    -> position de a dans L si a est dans L, et False sinon """

    d,f = 0, len(L)-1          # dichotomie sur les positions
    while d <= f :             # tant que les compteurs ne se sont pas rejoints
        m = (d+f)//2           # indice médian
        if L[m] == a :         # a trouvé à la position m
            return m
        elif L[m] < a :        # on garde la partie droite
            d = m+1
        else :                 # on garde la partie gauche
            f = m-1
    return False               # élément a pas dans L
```

Complexité :

- **Dans le pire des cas** : $\boldsymbol{O(\log(n))}$ avec $n = len(L) = 2^p$

  En effet, si $n = 2^p$ ; à chaque itération, la recherche conserve la moitié de la liste précédente.
  Taille initiale : $2^p$
  Taille après une itération : $2^{p-1}$
  Taille après 2 itérations : $2^{p-2}$
  …
  Taille après $k$ itérations : $2^{p-k}$
  Arrêt de la boucle quand il n'y a plus qu'un seul élément, donc quand $2^{p-k} = 1$ ce qui donne $p = k$, donc après maximum p itérations.
  Or, si $n = 2^p$, alors $p = \log_2(n)$.
  Donc dans le pire des cas, la recherche effectue $\log_2(n)$ comparaisons.
  Donc complexité en $O(\log(n))$.

- **Complexité moyenne : $\boldsymbol{O(\log(n))}$**

- **Complexité dans le meilleur des cas** : $\boldsymbol{O(1)}$**,** donc indépendante de la taille de la liste.

  C'est le cas où l'élément cherché est à la position médiane donc trouvé du premier coup.
  C'est un cas très favorable qui ne nécessite qu'une seule comparaison.