Cahier de vacances - Les algorithmes à maîtriser en fin de sup

On reprend un à un les points cités par le programme…

1/. Rechercher un élément dans une liste

Ecrire une fonction **`cherche(L,elt)`** qui renvoie un booléen indiquant si l'élément `elt` est présent ou non dans `L`, sans utiliser le test `in`.
Donner la complexité en temps dans le pire des cas.
Variante : qui renvoie la position de l'élément (ou ***la liste de toutes les positions de l'élément***)

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

Ecrire une fonction **`somme(L :[int])->int`** qui renvoie la somme des éléments de la liste L.
Donner la complexité en temps de cette fonction.

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

Ecrire une fonction **`compter(L)->int`** qui renvoie le nombre d'éléments positifs de la liste L ; on supposera que L ne contient que des flottants.
Donner la complexité en temps de cette fonction.

4/. Rechercher le maximum minimum dans une liste

Ecrire une fonction **`maxi(L)`** qui renvoie le maximum des éléments de `L`, ainsi qu'une de ses positions dans la liste.
Donner la complexité en temps dans le pire des cas.
Variante : qui renvoie ***toutes*** les positions du maximum

5/. Rechercher le second maximum dans une liste

Ecrire une fonction **`second_maxi(L)`** qui renvoie les deux valeurs maximales de la liste (éventuellement égales).
Donner la complexité en temps dans le pire des cas.
Variante : qui renvoie ***aussi leurs positions***

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

Ecrire une fonction **`compte(L) -> dict`** qui renvoie un dictionnaire indiquant le nombre d'occurrences de chaque élément de L.
Donner la complexité en temps dans le pire des cas.

7/. Recherche d'un facteur dans un texte

Ecrire une fonction **`cherche_mot(T,mot)`** qui renvoie l'indice de la première occurrence du mot dans le texte si le mot y est, et `False` ou `None` sinon.
Variante : qui renvoie les indices de ***toutes*** les occurrences

Donner la complexité en temps dans le pire des cas pour la première version.

Version sans slicings :

Version avec slicings :

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

Ecrire une fonction **`deux_val_proches(L)`** qui renvoie les deux valeurs les plus proches de L (ainsi que leurs positions).

Donner la complexité en temps dans le pire des cas.

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

a/. Ecrire une fonction **`tri_bulles(L)-> None`** qui trie en place la liste L ; on supposera que L est une liste de valeurs numériques.

Donner la complexité de ce tri dans le meilleur et dans le pire des cas.

a/. Ecrire une fonction **`tri_insertion(L)-> None`** qui trie en place la liste L avec un tri par insertion (le plus classique aux écrits !!!) ; on supposera que L est une liste de valeurs numériques.

Donner la complexité de ce tri dans le meilleur et dans le pire des cas, ainsi que la complexité moyenne.

10/. Recherche dichotomique dans un tableau trié

Ecrire une fonction `cherche_dicho(L,a)` qui recherche si l'élément `a` est présent dans la liste L. On supposera la liste `L` triée dans l'ordre croissant. Si l'élément `a` est présent, on renverra sa position dans la liste, sinon on renverra `False`.

Donner la complexité en temps de cet algorithme. Justifier brièvement ; on pourra supposer que la liste L contient 2^p éléments (plus simple à montrer).

11/. Quelques algorithmes de conversion sur les graphes

On suppose avoir un graphe G donné sous la forme $G = (S, A)$ avec S la liste de ses sommets, et A la liste de ses arêtes, données sous la forme `[s1,s2]`.

On supposera ici le graphe G **non orienté**. Peu de choses changeraient dans le cas orienté.

Q1/. Ecrire une fonction **`listeAdjNO(G : (list,list))->dict`** qui à partir d'un graphe G non orienté donné sous la forme (S, A) construit et renvoie le dictionnaire par listes d'adjacence du graphe.

Donner la complexité en temps de cet algorithme.

Variante : idem dans le cas où on a des pondérations sur les arêtes, qui sont alors données sous la forme de triplets (`s1,s2,poids`)

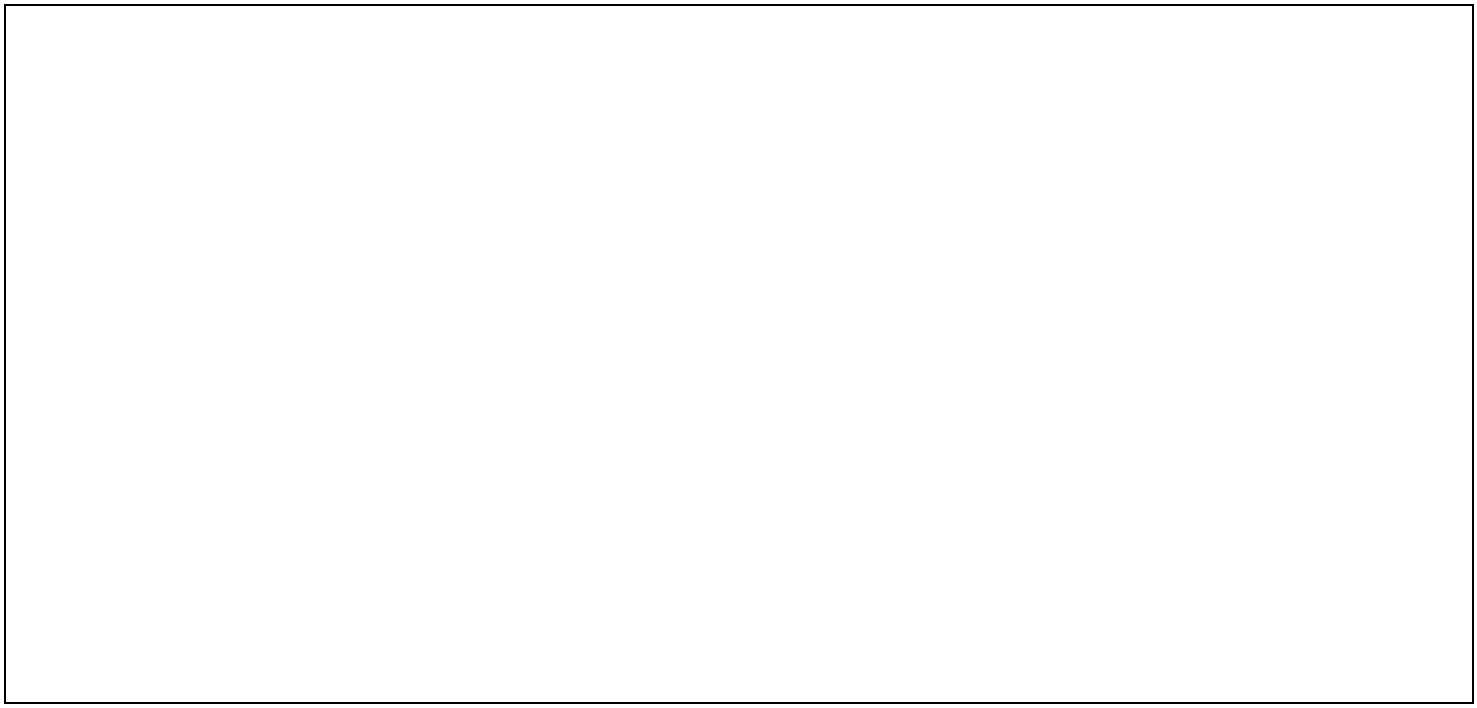
Q2/. Ecrire une fonction **`MatAdjNO(G : (list,list))->dict`** qui à partir d'un graphe G non orienté donné sous la forme (S, A) construit et renvoie la matrice d'adjacence du graphe.
On pourra dans un premier temps considérer que les sommets sont les entiers de $[\![0, p]\!]$, pou un certain entier p.
On pourra ensuite généraliser à des sommets de tous types…

Donner la complexité en temps de ces algorithmes. Justifier un peu.

Variante : idem dans le cas où on a des pondérations sur les arêtes, qui sont alors données sous la forme de triplets (`s1,s2,poids`)

Avec des sommets de $[\![1\,;p]\!]$:

Avec des sommets quelconques :



12/. Quelques algorithmes de parcours de graphes

Sera repris en spé… Mais ça ne fait pas de mal de relire l'idée générale…

Parcours en largeur (structure de FILE)

a/. **Parcours en largeur classique** qui prend en paramètre un graphe représenté par un dictionnaire par listes d'adjacence et un sommet du graphe et renvoie une liste donnant l'ordre de visite des sommets selon un algorithme de parcours en largeur.

```
def parcours_largeur(d : dict, s) -> list :

    """d : représentation du graphe sous forme d'un dictionnaire par
    listes d'adjacence
    s : sommet de départ du parcours
    -> liste des sommets visités avec ordre de visite selon un algorithme de
    parcours en largeur.

    Code donné avec une structure de liste ; possibilité, si le sujet le mentionne,
    de travailler avec des deque pour gagner en temps d'exécution sur le retrait
    en tête de file """

    visites = []                    # sommets visités
    file_attente = [s]              # seul le sommet de départ en attente
    marquage = {s : True}           # dictionnaire pour limiter la complexité

    while len(file_attente) > 0  :  # tant qu'il reste des sommets accessibles à voir
        x = file_attente.pop(0)     # Complexité en O(len(file_attente))
                                    # et complexité O(1) si on utilise une deque
                                    # avec x = file_attente.popleft()
        visites.append(x)
        for vois in d[x] :              # pour chacun des voisins
            if vois not in marquage :   # uniquement voisins non marqués
                                        # test en O(1) puisque marquage est un dict
                file_attente.append(vois)
                marquage[vois] = True
    return visites
```

Complexité temporelle : $\boldsymbol{O(n+m)}$ si $n = len(S)$ et $m = len(A)$

En effet, chaque sommet est visité une fois

Chaque arête est considérée une seule fois (ou deux si non orientée)

b/. Variante du parcours en largeur avec construction d'un « *dictionnaire des pères* » pour garder un historique du parcours : d'où venait-ton avant de découvrir tel sommet.
Dico_peres = {sommet : sommet_d'ou_on_venait}

Puis **reconstruction du plus court chemin entre les sommets s et v** (si ce chemin existe !) :

```
def parcours_largeur_pere(d : dict, s) -> list :

    """d : représentation du graphe sous forme d'un dictionnaire par
    listes d'adjacence
    s : sommet de départ du parcours
    -> dictionnaire des pères selon un algorithme de
    parcours en largeur.

    Code donné avec une structure de liste ; possibilité, si le sujet le mentionne,
    de travailler avec des deque pour gagner en temps d'exécution sur le retrait
    en tête de file """

    file_attente = [s]                # seul le sommet de départ en attente
    Pere = {s : None}                 # dictionnaire des pères ; s n'a pas de père (départ)
                                      # sert aussi de marquage...

    while len(file_attente) > 0  :    # tant qu'il reste des sommets accessibles à voir
        x = file_attente.pop(0)       # Complexité en O(len(file_attente))
                                      # et complexité O(1) si on utilise une deque
                                      # avec x = file_attente.popleft()
        for vois in d[x] :            # pour chacun des voisins
            if vois not in Pere :     # uniquement voisins non marqués
                                          # test en O(1) puisque Pere est un dict
                file_attente.append(vois)
                Pere[vois] = x
    return Pere
```

<u>*Reconstruction*</u> :

```
def chemin(d,s,v) :

    """ d : représentation d'un graphe G non pondéré, donné par un dictionnaire
    par listes d'adjacence
    s,v : deux sommets du graphe, supposés "connectés"
    -> plus court chemin de s à v """

    P = parcours_largeur_pere(d,s)      # on récupère le dictionnaire des pères
    che = [v]                           # construction (à l'envers !) du chemin
                                        # en remontant de proche en proche
                                        # l'historique du parcours
    while P[v] != None :                # P[s] = None donc on aura alors fini...
        che.append(P[v])
        v = P[v]
    che.reverse()                       # on remet la liste dans le bon ordre
    return che
```

Parcours en profondeur (Structure de PILE : récursivité)

```
def visite_profondeur(d,s,visite,marquage) :

    """ d : graphe représenté par un dictionnaire d'adjacence
    s : sommet du graphe
    visite : liste des sommets visités
    marquage : dictionnaire des marquages pour ne pas revenir sur un sommet

    -> None
    MAIS modifie en place la liste visite et le dictionnaire
    marquage en fonction du parcours et des sommets découverts """

    if s not in marquage :
        marquage[s] = True
        visite.append(s)

    for vois in d[s] :              # pour chaque voisin non marqué de s
        if vois not in marquage :
            visite_profondeur(d,vois,visite,marquage)   # on lance un parcours
                                                        # en profondeur
```

```
def parcours_profondeur(d,s) :

    """ d :  graphe représenté par son dictionnaire d'adjacence
    s : sommet du graphe
    Effectue un parcours en profondeur du graphe à partir du sommet s et
    renvoie l'ordre des sommets visités
    Remarque : si le graphe est "connexe", c'est-à-dire en un seul morceau,
    tous les sommets sont atteints par un tel parcours """

    visite = []
    marquage = {}
    visite_profondeur(d,s,visite,marquage)   # remplit visite et marquage
    return visite                            # renvoie l'ordre de visite
```

```
def parcours_tout_graphe_profondeur(d) :

    """ Ne diffère de la version précédente QUE SI  GRAPHE NON  CONNEXE, donc
    pas un seul morceau ; quand l'algo ne peut plus rien visiter à partir d'un
    sommet, il repartira d'un autre sommet non visité """

    visite = []
    marquage = {}
    for s in d :
        if s not in marquage :
            visite_profondeur(d,s,visite,marquage)
    return visite
```

Complexité : idem

Variante avec le dictionnaire des pères :

```
def profondeur_rec_pere(d : dict, s , P = None) -> dict :

    """ d : graphe représenté avec un dictionnaire d'adjacence
    s : sommet du graphe
    P : futur dictionnaire des pères
    -> dictionnaire des pères  """

    if P == None :
        P = {s : None}          # initialisation du dictionnaire, uniquement au
                                # 1er appel récursif

    attente = [v for v in d[s] if v not in P]    # voisins non marqués
    for v in attente :
        if v not in P :
            P[v] = s           #  s est le père de v donc a permis de découvrir v
            profondeur_rec_pere(d,v,P)    # on fait un parcours en profondeur
                                          # au départ de v
    return P
```

Graphes pondérés : Recherche d'un plus court chemin avec Dijkstra

(revu en spé)

```
def mini_dico(d)  :

    """ d : dictionnaire représenté par liste d'adjacence
    -> clé correspondant à la plus petite valeur """

    mini = float('inf')        # + infini
    for k in d.keys() :
        if d[k] < mini :
            mini = d[k]
            cle = k
    return cle
```

```
def dijkstra(dico,s) :

    """ dico : graphe PONDERE représenté par dictionnaire d'adjacence
    s : sommet de départ
    -> dictionnaire donnant la plus courte distance entre s et
    tous les autres sommets du graphe """

    d = {k : float('inf') for k in dico}      # distances initiales
    d[s] = 0
    D = {}                                    # distances finales au sommet s
    while len(d) > 0 :
        k = mini_dico(d)                      # sommet de distance minimale
        for i in dico[k] :                    # on regarde les voisins de k
            vois, dist = i                    # les voisins sont des couples !
            if vois not in D :
                d[vois] = min(d[vois],d[k] + dist)   # mise à jour des distances
        D[k] = d[k]                                  # on copie le sommet et distance
        del d[k]
        #print(D)
    return D
```

Complexité : $O(n^2 + m)$ où $n = len(S)$ et $m = len(A)$

Le fonction `mini_dico` est appelée à chaque itération de la boucle principale de `Dijkstra` et a une complexité en $O(n)$ (parcours des clés).

Pour `Dijkstra`, la boucle principale tourne n fois, et on supprime un sommet à chaque étape.

A chaque itération :

- Appel à mini_dico : $O(n)$
- Boucle sur les voisins de k : **en tout,** chaque arête est examinée une fois, donc au total **sur toutes les itérations** : $O(m)$

Bilan :

- Les appels successifs à mini_dico sur un dictionnaire de plus en plus petit :

$$O(n) + O(n-1) \ldots\ldots + O(1) = O(n^2)$$

- Boucle sur les voisins : $O(m)$ au total

Donc $O(n^2 + m)$

13/. Les algos de tri récursifs (Complexité temporelle moyenne $O(n\,log(n))$)

(revus en spé)

a/. Tri rapide ou Quicksort (sur le principe du « diviser pour régner »)

Principe :

- Choisir un pivot (souvent premier ou dernière valeur)
- Réaliser une partition : les éléments plus petits que le pivot à gauche, ceux plus grands à droite)
- Appliquer récursivement le tri sur les deux sous-listes

```
def partition(L : list)-> tuple :

    """L : liste d'éléments comparables avec <, >
    -> partition partiellement triée :
        (liste des éléments < pivot, pivot, lite des éléments >= pivot) """

    pivot = L[0]                # choix à discuter...
    ppetit , pgrand = [],[]
    for k in L[1:] :            # on fait défiler tous les autres éléments
        if k < pivot :
            ppetit.append(k)    # les éléments plus petits que le pivot
        else :
            pgrand.append(k)    # les éléments sup. ou égaux au pivot
    return ppetit, pivot, pgrand
```

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

    """ L : liste d'éléments comparables
    -> copie de la liste triée (ordre croissant) """

    if len(L) <= 1 :                                # critère d'arrêt
        return L

    Avant, P, Apres = partition(L)                  # diviser pour régner
    return quicksort(Avant) + [P] + quicksort(Apres)  # reconstruction
```

Complexité :

- Complexité moyenne (et meilleur des cas) : $\boldsymbol{O(n\log(n))}$
- Complexité dans le pire des cas : $O(n^2)$ (si le pivot est toujours le + petit ou + grand élément, les partitions sont très déséquilibrées)

b/. Tri fusion ou Merge Sort (sur le principe du « diviser pour régner »)

Principe :

- Diviser la liste en deux moitiés (*diviser pour régner)*
- Trier récursivement chaque moitié
- Fusionner les deux listes triées

```
def fusion(L1 : list, L2 : list) -> list :

    """L1, L2 deux listes triées (par ordre croissant)
    -> fusion des listes L1 et L2 en une liste croissante """

    i,j = 0,0               # compteurs pour parcourir L1 et L2
    L = []                  # liste fusionnée
    while i < len(L1) and j < len(L2) :
        if L1[i] <= L2[j] :
            L.append(L1[i])
            i = i+1
        else :
            L.append(L2[j])
            j = j+1
    # on rajoute ce qui reste de la liste non entièrement parcourue :

    if i == len(L1) :               # L1 vidée, mais pas L2
        L = L + L2[j:]
    else :                          # L2 vidée mais pas L1
        L = L + L1[i:]
    return L
```

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

    """ L liste d'éléments comparables
    -> copie de la liste triée récursivement (ordre croissant) par
    un algo de tri fusion """

    if len(L) <= 1 :          # critère d'arrêt
        return L
    L1 = tri_fusion(L[:len(L)//2] )       # première moitié de liste
    L2 = tri_fusion(L[len(L)//2 : ] )     # 2e moitié
    return fusion(L1,L2)                  # reconstruction
```

Complexité : $O(n\log(n))$ dans tous les cas

Le découpage est toujours en moitié, quel que soit le contenu ; idem pour les fusions.

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.

11/. Quelques algorithmes de conversion sur les graphes

Q1/. Dictionnaire d'adjacence :

```
def listeAdjNO(G)->dict :

    """G graphe non orienté donné sous la forme (S,A)
    -> dictionnaire par listes d'adjacence du graphe """

    S,A = G                         # on sépare sommets et arêtes
    d = {k : [] for k in S}         # initialisation du dictionnaire final
    for s1,s2 in A :                # parcours des arêtes
        d[s1].append(s2)            # s2 est un voisin de s1
        d[s2].append(s1)            # UNIQUEMENT  SI  NON  ORIENTE !
    return d
```

Complexité :

- Initialisation du dictionnaire en $O\big(len(S)\big)$
- Boucle sur les arêtes (et corps de boucle en temps constant pour les ajouts dans une liste) en $O\big(len(A)\big)$.

Complexité globale en $O\big(len(A) + len(S)\big)$

Q2/. Matrice d'adjacence :

Avec les sommets dans $[\![0\,, p]\!]$:

```
def MatAdjNO(G) -> [list] :

    """G graphe non orienté donné sous la forme G = (S,A)
    -> matrice d'adjacence du graphe
    On suppose ici que les sommets sont les entiers de [0 ; p]"""

    S,A = G             # on sépare sommets et arêtes
    M = [[0]*len(S) for k in range(len(S))]
    for s1,s2 in A :
        M[s1][s2] = 1
        M[s2][s1] = 1   # UNIQUEMENT  SI  NON  ORIENTE
    return M
```

Complexité :

Notons $n = len(S)\ \ et\ \ m = len(A)$

- Initialisation de la matrice : $O(n^2)$ car `range(n)` itérable de longueur n donc $O(n)$ et pour chaque élément du `range(n)` on a `[0]*n` qui crée une liste de n zéros : $O(n)$, ce qui fait une complexité quadratique en le nombre de sommets : $O(n^2)$.
- Remplissage de la matrice : pour chaque arête (boucle en $O(m)$), il y a deux affectations, donc coût constant.

Conclusion : Complexité globale en $O(n^2 + m)$.

Avec des sommets quelconques :

```python
def MatAdjNO_2(G) -> [list] :

    """G graphe non orienté donné sous la forme G = (S,A)
    -> matrice d'adjacence du graphe
   Sommets quelconques """

    S,A = G             # on sépare sommets et arêtes
    M = [[0]*len(S) for k in range(len(S))]
    for s1,s2 in A :
        pos_s1 = S.index(s1)     # position de s1 dans la liste S pour
                                 # savoir numéro ligne et colonne dans la matrice
        pos_s2 = S.index(s2)
        M[pos_s1][pos_s2] = 1
        M[pos_s2][pos_s1] = 1   # UNIQUEMENT  SI  NON  ORIENTE
    return M
```

Complexité en temps : on note $n = len(S) = |S|$ et $m = len(A) = |A|$

- Affectation $S, A = G$ en temps constant $O(1)$
- Initialisation de M en $O(n^2)$ comme vu dans le code précédent.
- Boucle sur les arêtes (il y en a m) :
    - `S.index(s1)` parcourt la liste S donc $O(n)$
    - Idem pour `S.index(s2)` : $O(n)$
    - Affectations dans la matrice : $O(1)$ temps constant

  Donc pour chaque arête, on a un coût de $O(n) + \ O(n) + O(1) = O(n)$
  Donc pour m arêtes, on a un $O(mn)$

Donc la complexité globale est : $O(n^2 + mn)$