Devoir surveillé n°1 - Test de rentrée - Version n°1- Corrigé

Exercice 1 : 1. 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`.

```
def cherche(L, elt ) :
    for x in L : #on parcourt tous les éléments de la liste
        if x==elt:
                return True # on arrête la recherche si on a trouvé elt
    return False # la liste a été parcourue entièrement et on n'a jamais trouvé elt
```

2. Donner, en justifiant, la complexité en temps dans le pire des cas, en fonction de la taille n de la liste `L`.

Le pire des cas est celui où tous les éléments de la liste vont être parcourus. Comme chaque tour de boucle est de complexité constante, *la complexité totale sera en O(n), c'est-à-dire linéaire*.

Exercice 2 : Ecrire une fonction `maximum(L)` qui renvoie **un couple** constitué de la valeur de l'élément maximum de la liste `L`, et à la position de ce maximum dans la liste (si ce maximum est atteint plusieurs fois, l'indice renvoyé sera le plus petit).

```
def maximum(L) :
    ind_maxi=0 #pour stocker l'indice où se trouve le maximum
    maxi=L[0] #pour stocker la valeur du maximum
    for i in range(len(L)) : #on parcourt tous les indices de la liste
        if L[i]>maxi: #on a rencontré un élement strictement plus grand
                maxi=L[i]
                ind_maxi=i
    return maxi,ind_maxi
```

Exercice 3 (*Tri par insertion*) : 1. .

```
def tri_insertion (L) :
    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
```

2. Donner, en justifiant, la complexité en temps dans le pire des cas, en fonction de la taille n de la liste `L`.

Dans le pire des cas, la complexité est quadratique, en $O(n^2)$:
Le pire des cas est 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. La boucle while tourne un maximum de fois.

- Au premier tour de la boucle for, $k=1$, la boucle while tourne 1 fois, et comprend un nombre constant d'opérations.
- Au deuxième tour de la boucle for, $k=2$, la boucle while tourne 2 fois, et comprend un nombre constant d'opérations.
- ....

- Au dernier tour de la boucle for, $k = n-1$, la boucle while tourne $n-1$ fois, et comprend un nombre constant d'opérations.

Le nombre total d'opérations est donc : $1+2+\cdots+(n-1) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$

Devoir surveillé n°1 - Test de rentrée - Version n°2 - Corrigé

Exercice 1 : 1. Ecrire une fonction `somme_positifs(L)` qui renvoie la somme des éléments **positifs** de `L`, sans utiliser le test `sum`.

```
def somme_positifs(L) :
    somme=0
    for x in L : #on parcourt tous les éléments de la liste
        if x>=0:
                somme+=x
    return somme
```

2. Donner, en justifiant, la complexité en temps dans le pire des cas, en fonction de la taille n de la liste `L`.

   Ici, on est toujours dans le pire des cas, car de toute façon tous les éléments de la liste vont être parcourus. Comme chaque tour de boucle est de complexité constante, *la complexité totale sera en O(n), c'est-à-dire linéaire*.

Exercice 2 : Ecrire une fonction `minimum(L` qui renvoie **un couple** constitué de la valeur de l'élément minimum de la liste `L`, et à la position de ce minimum dans la liste (si ce minimum est atteint plusieurs fois, l'indice renvoyé sera le plus petit).

```
def minimum(L) :
    ind_mini=0 #pour stocker l'indice où se trouve le minimum
    mini=L[0] #pour stocker la valeur du minimum
    for i in range(len(L)) : #on parcourt tous les indices de la liste
        if L[i]<mini: #on a rencontré un élement strictement plus petit
                mini=L[i]
                ind_mini=i
    return mini,ind_mini
```

Exercice 3 (*Recherche par dichotomie dans un tableau trié*) : .

1. On suppose que la liste `L` est **triée selon les valeurs croissantes**.

```
def cherche_dicho(L,elt) :
    ind_deb=0
    ind_fin=len(L)-1
    while ind_deb <= ind_fin: # tant que les compteurs ne se sont pas rejoints
        m = (ind_deb+ind_fin)//2 # indice médian
        if L[m] == elt : # elt trouvé à la position m
                return True
        elif L[m] < elt : # on garde la partie droite
            ind_deb = m+1
        else: # on garde la partie gauche
            ind_fin = m-1
    return False
```

2. Dans le pire des cas , la complexité est logarithmique, en $\boldsymbol{O(\log(n))}$.

   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 la complexité est en $O(\log(n))$.