

Exercice - plus grand élément et occurrences

Plus grand élément

Écrire une fonction qui prend en argument une liste L et qui renvoie un couple constitué

- du plus grand élément de la liste
- du nombre de fois que ce plus grand élément apparaît

une contrainte importante : la fonction ne peut utiliser **qu'une seule** boucle (et évidemment ne pas utiliser la fonction *max* de Python)

```
Entrée[1]: def plus_grand(L):  
            n = len(L)  
            maxi = L[0]  
            nb = 1  
            for i in range(1,n):  
                a = L[i]  
                if a>maxi:  
                    maxi = a  
                    nb = 1  
                elif a==maxi:  
                    nb += 1  
            return maxi,nb
```

```
Entrée[2]: plus_grand([3,8,4,2,5,8,5,6,2,3,8,5])
```

```
Sortie[2]: (8, 3)
```

```
Entrée[3]: plus_grand([3,8,4,2,5,8,5,6,2,3,8,5,8])
```

```
Sortie[3]: (8, 4)
```

```
Entrée[4]: plus_grand([3,8,4,2,5,8,5,6,2,3,8,5,9])
```

```
Sortie[4]: (9, 1)
```

Exercice - recherche des p plus grands

Autour de la méthode d'insertion - recherche des p plus grands éléments

On réutilise la fonction d'insertion que vous avez vue lors du tri par insertion dans une autre situation

fonction d'insertion

Écrire une fonction d'insertion dans une liste, triée de façon croissante, d'un élément si celui ci est plus grand que le minimum de la liste en renvoyant toujours une liste triée

Par exemple *inserer(6,[3,5,7,9])* devra renvoyer *[5,6,7,9]*, alors que *inserer(2,[3,5,7,9])* renverra la liste non modifiée.

```
Entrée[5]: def inserer(x,T):
            if x>T[0]: # sinon on ne fait rien
                i = 1
                n = len(T)
                while i<n and x>T[i]:
                    T[i-1] = T[i]
                    i += 1
                T[i-1] = x
```

```
Entrée[6]: A = [3,5,7,9]
            inserer(2,A)
            A
```

```
Sortie[6]: [3, 5, 7, 9]
```

Fonction p plus grands

On l'utilise pour la fonction de recherche des p plus grands :

- on crée une liste de taille p avec des éléments suffisamment petits
- on insère les éléments de la liste les uns après les autres

Infini

Python dispose d'un infini : il s'obtient par la conversion en *float* de la chaîne de caractère "inf" (ou "-inf")

```
Entrée[7]: A = float("inf")
            300000000000 < A
```

```
Sortie[7]: True
```

```
Entrée[8]: A
```

```
Sortie[8]: inf
```

```
Entrée[9]: "inf"
```

```
Sortie[9]: 'inf'
```

```
Entrée[10]: -2*A+4
```

```
Sortie[10]: -inf
```

```
Entrée[11]: 2*A-A
```

```
Sortie[11]: nan
```

```
Entrée[12]: def P_plus_grands(L,p):
            A = [float("-inf")]*p
            for x in L:
                inserer(x,A)
            return A
```

```
Entrée[13]: P_plus_grands([3,8,4,6,1,9,23,11,4,5,3,9,7,12],5)
```

```
Sortie[13]: [9, 9, 11, 12, 23]
```


Sans chevauchement

si on cherche la chaîne de caractères 'aba' dans la phrase 'cababad aba', on considère que la première apparition dans 'ababa' compte mais qu'ensuite on continue après et ainsi le 'a' commun n'est plus pris en compte

Écrire une fonction qui détermine le nombre d'occurrences d'une chaîne de caractères dans une autre sans chevauchement possible

```
Entrée[19]: def occurrence_sans(phrase,mot):
    n = len(phrase)
    l = len(mot)
    nb = 0
    i = 0
    while i<=n-l:
        if phrase[i:i+l]==mot:
            nb += 1
            i += l
        else:
            i += 1
    return nb
```

```
Entrée[20]: occurrence_sans("cababaabaabbcababadaba","abaa")
```

```
Sortie[20]: 2
```

```
Entrée[ ]:
```

Exercice - Recherche dichotomique

Recherche dichotomique dans une liste triée

On se donne une liste d'entiers **triée**. On veut savoir si l'élément x appartient à cette liste (en exploitant le fait que la liste soit triée, ce qui permet d'avoir une complexité logarithmique)

Écrire la fonction de recherche

```
Entrée[21]: def recherche_dicho(L,x):
    n = len(L)
    g,d = 0, n-1
    while g<=d:
        m = (g+d)//2
        if L[m]==x:
            return True
        if L[m]>x:
            d = m-1
        if L[m]<x:
            g = m+1
    return False
```

Quelques tests :

Entrée[22]: `L_test = [2,6,8,11,14,19,23,27]`

Entrée[23]: `recherche_dicho(L_test,14)`

Sortie[23]: `True`

Exercice - Tri d'une liste 0,1

Tri d'une liste à deux éléments

On dispose d'une liste de 0 et de 1 (autre chose si vous préférez). On veut la trier *sur place* en un seul passage de boucle (et donc sans créer de nouvelle liste ou sous-liste)

Méthode

on va gérer plusieurs compteurs :

- un compteur *i* qui correspond à la position qu'on est en train d'étudier - arrivé à cette position les *i-1* premiers 0 et 1 sont supposés avoir été triés
- un compteur *milieu* qui désigne la position du premier 1 dans les éléments triés (les 0 vont de la position 0 à *j-1* et les 1 de la position *j* à *i-1*)
- on regarde l'élément en position *i* : si c'est un 1, il est bien placé, si c'est un 0, on "l'échange" avec le premier 1 de la partie triée

On testera la fonction sur un ensemble suffisamment significatif d'exemples (en pensant bien à tous les cas particuliers qui peuvent arriver... liste à 0 ou 1 élément ? le dernier élément est-il bien pris en compte...

```
Entrée[24]: def tri_liste_01(L):
            milieu = 0
            for i in range(len(L)):
                if L[i]==0: #sinon on ne fait rien
                    L[milieu] , L[i] = L[i],L[milieu]
                    milieu += 1
```

```
Entrée[25]: A = [0,1,1,1,0,1,0,1,1,0,0,1,0,0,1,0]
            B = [1,1,0,0,1,0,1,0,0,0,1,1,0]
```

```
Entrée[26]: tri_liste_01(A)
            A
```

Sortie[26]: `[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]`

```
Entrée[27]: tri_liste_01(B)
            B
```

Sortie[27]: `[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]`

Exercice - récursivité, dictionnaires avec la suite de Syracuse

Objectif

On cherche à explorer différentes propriétés de la suite de Syracuse qui est définie par

- un premier terme $u_0 = p \in \mathbb{N}^*$,
- une relation, pour tout $n \in \mathbb{N}$, $u_{n+1} = u_n / 2$ si u_n est pair et $u_{n+1} = 3u_n + 1$ si u_n est impair

définition de la suite

Écrire une fonction `liste(p, n)` qui prend deux entiers en entrée et qui renvoie les termes $(u_0, \dots, u_n)$ de la suite avec $u_0 = p$. Tester cette fonction. Que constate-t-on ?

```
Entrée[28]: def liste(p,n):
              u = p
              L = [p]
              for i in range(n):
                  if u%2==0:
                      u = u//2
                  else:
                      u = 3*u+1
                  L.append(u)
              return L
```

```
Entrée[29]: liste(5,10)
```

```
Sortie[29]: [5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2]
```

```
Entrée[30]: liste(7,20)
```

```
Sortie[30]: [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4]
```

On constate que la suite devient périodique à partir d'un certain rang (elle semble atteindre 1 et ensuite tourner sur 1,4,2)

Temps d'atteinte de 1

Temps de vol: écrire une fonction `TempsDeVol(p)` qui renvoie le nombre d'étapes pour arriver à la valeur 1 pour la première fois lorsque la suite commence à la valeur p (on retournera la valeur 0 si $p = 1$). Quel est l'entier inférieur à 100 (puis 1000) qui a le plus long temps de vol ?

```
Entrée[ ]:
```

On se rend compte que si on arrive sur un entier pour lequel on connaît le temps pour arriver à 1, il est inutile de continuer à calculer les termes de la suite.

Temps d'atteinte de 1 - amélioration

En utilisant un dictionnaire, écrire une fonction améliorée qui détermine le temps de vol des entiers inférieurs à un entier donné.

```
Entrée[31]: def f(a): # pour ne pas avoir à le refaire à chaque fois
            if a%2==0:
                return a//2
            else:
                return 3*a+1

            def TempsDeVol(p):
                v = p
                n = 0
                while v>1:
                    v = f(v)
                    n +=1
                return n

            def MaxTemps(n):
                M = 0
                idx = 1
                for i in range(2,n+1):
                    a = TempsDeVol(i)
                    if a>M:
                        M = a
                        idx = i
                return idx,M
```

Entrée[32]: TempsDeVol(7)

Sortie[32]: 16

Entrée[33]: MaxTemps(100)

Sortie[33]: (97, 118)

Pour les entiers jusqu'à 100, le temps maximal est atteint pour 97 (avec 118 termes avant d'arriver à 1).

Altitude

écrire une fonction `Altitude(p)` qui renvoie le maximum des termes de la suite qui commence à p . On le fera de 3 façons différentes :

- une version impérative qui prend p en argument et qui détermine le maximum des termes rencontré jusqu'à ce qu'on obtienne 1,
- une version récursive : si on note u_0 et u_1 les deux premiers termes de la suite, quelle est la relation entre $altitude(u_0)$ et $altitude(u_1)$? Exploiter cette remarque pour écrire une version récursive de l'altitude,
- avec un dictionnaire : on stocke les altitudes rencontrées et on les réutilise au besoin

Quelle est l'altitude maximale pour les 100 (resp. 1000) premiers entiers ? pour quels entiers de départ sont-elles atteintes ?

```
Entrée[34]: def Altitude(p):
            alt = p
            u = p
            while u>1:
                u = f(u)
                if u>alt:
                    alt = u
            return alt
```

Entrée[35]: Altitude(97)

Sortie[35]: 9232

On peut alors rechercher l'altitude maximale pour les premiers entiers :

```
Entrée[36]: def MaxAlt(n):  
    M = 0  
    idx = 1  
    for i in range(2,n+1):  
        a = Altitude(i)  
        if a>M:  
            M = a  
            idx = i  
    return idx,M
```

Entrée[37]: MaxAlt(100)

Sortie[37]: (27, 9232)

Entrée[38]: MaxAlt(1000)

Sortie[38]: (703, 250504)