Correction du TD ipt^2 n° 3: Révisions
Récursivité - Tris

Algorithmique simple de la récursivité

Exercice n°1: **Inversion récursive**

❶ La méthode la plus simple est de permuter les éléments situés aux extrémités du tableau (premier passage) et de recommencer récursivement pour la liste "intérieure", c'est à dire privée des éléments extrêmes déjà inversés. On peut aussi imaginer placer le dernier élément en tête, et traiter ensuite récursivement la liste privé de ce dernier élément;

❷ On peut imaginer plusieurs cas de base en fonction du choix de stratégie; pour le premier code ci-dessous qui permute premier et dernier éléments à chaque récursion avec un renvoi de la liste dans son intégralité à chaque récursion, le cas de base correspond à un indice de parcours à la moitié de la liste, pour lequel on peut renvoyer celle-ci puisqu'elle est totalement inversée.

Pour les autres codes, on évite l'emploi d'un indice et le cas de base correspond alors à une liste de longueur 1 ou 0.

❸ On propose les scripts suivants:

Listing 1:

```
def inversionrec(L,i):
    if i>=len(L)//2:
        return L
    else:
        L[i],L[len(L)-1-i]=L[len(L)-1-i],L[i]
        return inversionrec(L,i+1)

L=[1,2,3,4,5,6,7,8,9]
print (inversionrec(L,0))
```

qui renvoie bien:

[9,8,7,6,5,4,3,2,1]

Un version plus simple évitant d'employer un indice dans le premier appel:

Listing 2:

```
def inversionrec(L):
        if len(L)==1:
                return L
        else:
                L[0],L[-1]=L[-1],L[0]
                return inversionrec(L[1:-1])

L=[1,2,3,4,5,6,7,8,9]
inversionrec(L)
print(L)
```

La seconde méthode proposée donne:

Listing 3:

```
def inversionrec_V_2(L):
    if len(L)==0:
        return []
    else:
        return [L[-1]]+inversionrec_V_2(L[:-1])
```

On propose encore une dernière version:

Listing 4:

```
def inversionrec_V3(L):
    if len(L)<=1:
        return L
    else:
        L[0],L[-1]=L[-1],L[0]
        return [L[0]]+inversionrec_V3(L[1:-1])+[L[-1]]
```

Exercice n°2: **Fonction** `mystere`

❶ **a**· Dans le cas de l'appel `mystere(T,2)`, la pile d'execution contient:

`mystere(T,2)` ⟹ `mystere(T,3)` ⟹ `mystere(T,4)` ⟹ `True`

b· Dans le cas de l'appel `mystere(T,0)`, la pile d'execution contient cette fois:

`mystere(T,0)` ⟹ `mystere(T,1)` ⟹ `False`

❷ La fonction `mystere` vérifie que les entiers d'une liste à partir de l'indice k sont classés par ordre croissant. Elle renvoie alors `True` dans ce cas et naturellement `False` dans le cas contraire.

❸ Le nombre maximal d'appels récursifs est obtenu lorsque la liste à partir de l'indice k est effectivement classés en ordre croissant, situation pour laquelle on finit par rencontrer le cas de base ($k = len(T) - 1$). Dans ce cas, le nombre d'appels récursifs est $n - k$.

Listing 5: Script estCroissant(T)

❹
```python
def estCroissant(T):
    return mystere(T,0)
```

Listing 6: Codes_Python/Correction_ex_1_estDans_rec.py

❺
```python
# -*- coding: utf-8 -*-

def estDans(T,x,k):
    if k==len(T): #k vaut le dernier indice de liste+1
        return False
    if T[k]==x:
        return True
    else:
        return estDans(T,x,k+1)
Liste=[15,72,27,44,32,18,12,3]
print estDans(Liste,44,0)
print estDans(Liste,44,4)
```

La sortie est la suivante:

```
True
False
```

Exercice n°3: Suite de Fibonacci récursive rapide

❶ On a:

$$\begin{bmatrix}\mathcal{F}_n\\ \mathcal{F}_{n-1}\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\mathcal{F}_{n-1}\\ \mathcal{F}_{n-2}\end{bmatrix} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}\mathcal{F}_{n-2}\\ \mathcal{F}_{n-3}\end{bmatrix} = M^2\begin{bmatrix}\mathcal{F}_{n-2}\\ \mathcal{F}_{n-3}\end{bmatrix} = M^{n-1}\begin{bmatrix}\mathcal{F}_1\\ \mathcal{F}_0\end{bmatrix}$$

❷ On propose deux solutions; la première fait appel à un calcul naïf récursif de la multiplication matricielle, mais ne respecte pas la consigne d'énoncé sur la complexité en $O(\ln n)$:

Listing 7:

```python
import numpy as np
def Puissnaif(M,n):
    if n==1:
        return M
    else:
        return M.dot(Puissnaif(M,n-1))

mat=np.array([[1,1],[1,0]]) #définition d'une matrice d'essai
print (Puissnaif(mat,2)) # essai d'élévation au carré
```

La seconde possibilité ci-dessous exploite le même principe que l'exponentiation rapide afin d'abaisser la complexité en $O(\log_2(n))$ (en terme de produits matriciels):

Listing 8:

```python
import numpy as np
def Puiss_2(M,n):
    if n==1:
        return M
    else:
        q,r=n//2,n%2
        if r==0:
            return Puiss_2(M@M,q)
        else:
            return M@Puiss_2(M@M,q)
```

❸ On peut alors réaliser le calcul de $\mathcal{F}_n$ terme de rang n de la suite de Fibonacci de premier terme a et de second terme b avec la fonction `fiborapide(a,b,n)` suivante:

Listing 9:

```python
def fiborapide(a,b,n):
    mat=np.array([[1,1],[1,0]],int)
    if n==0:
        return a
```

```
    elif n==1:
        return b
    else:
        return Puiss(mat,n-1).dot(np.array([[b],[a]]))[0,0]
        # ou encore return Puiss(mat,n-1)@np.array([[b],[a]]))[0,0]
```

On peut remplacer la dernière ligne par

❹ *Question «pratique»:* présentation faite en live avec comparaison avec l'algorithme récursif classique (fichier disponible sur le site)!

EXERCICE N°4: Décomposition binaire récursive

❶ Quelques notions préliminaires

Listing 10:

a·
```
def uni_entier(n):
    if n<=0:
        return "n doit être strictement positif"
    p=0
    while 2**(p+1)<=n:
        p+=1
    return p
```

b·

- **Terminaison:** Si n est négatif ou nul, alors le programme renvoie un message d'erreur et termine; sinon, la boucle démarre avec la valeur $p = 0$ puis incrémente p d'une unité à chaque itération. Ainsi, il intervient un rang d'itération pour lequel 2^{p+1} est supérieure ou égal n, et le programme renvoie alors p et termine.

- **Correction:**
  INVARIANT DE BOUCLE: «Si la condition de boucle est satisfaite pour une valeur de p, alors, on a en fin d'itération $2^{p+1} \leq n$ et le compteur p qui passe à la valeur $p + 1$».

  Par exemple pour $n \geq 1$, on a $2^0 \leq n$: la propriété est donc vraie au rang $p = 0$.

  Si la condition de boucle est satisfaite pour $p + 1$ alors en fin d'itération on a 2^{p+2} <= n et l'exposant devient $p + 2$; c'est bien la propriété au rang $p + 1$; si la condition est violée, au moment du rang d'arrêt p_{max} on a $2^{p_{max}+1} > n$ on est donc assuré d'avoir $2^{p_{max}} \leq n$ et le programme renvoie p_{max} ce qui est bien le rang recherché.

c· Si on a $2^p \leq n < 2^{p+1}$ alors il faut au moins $p+1$ termes dans tableau (il ne faut pas oublier la valeur $p = 0$ qui représente un terme du tableau!). On a donc $t = p + 1$.

d· Tableau représentant 2^p: $L_1 = [1, 0, \ldots, 0]$
Tableau représentant $n - 2^p = \sum_{k=0}^{p} a_k \cdot 2^k - 2^p$: $L_2 = [a_p - 1, a_{p-1}, \ldots, a_0]$

Ainsi, le tableau représentant n est bien $L_n = L_1 + L_2 = [a_p, a_{p-1}, \ldots, a_0]$, l'opérateur "+" réalisant ici une sommation terme à terme et non une concaténation des listes.

❷ Construction du programme récursif

a· Etapes de l'algorithme récursif `decomp_bin(n,t)`:

- Si $n = 0$ on renvoie un tableau L de t zéros
- Sinon, on calcule p selon l'algorithme itératif précédent. On réserve un tableau L de taille suffisante $t \geq p + 1$. On affecte $L[t - p - 1] = 1$:

représent.nbre:	$t-1$,	$t-2$			**p**			1	0
	[0,	0,	$\ldots$,		**1**,	0,	$\ldots$,	0,	0]
indice tabl.:	0	1			**t − p − 1**			$t-2$	$t-1$

- On ajoute récursivement à L le tableau renvoyé par l'appel à `decomp_bin(n-2**p,t)` qui représente l'entier $n - 2^p$. On a donc au final le tableau attendu.

b· Attention: on doit réaliser une opération de sommation des tableaux élément par élément, ce qui nécessite l'emploi de tableau 1D du module `numpy` et non de simples listes qui subiraient alors une concaténation. On propose l'algorithme suivant:

Listing 11:

```
def decomp_bin(n,t):
    if n==0:
        return np.zeros(t)
    else:
        p=0
        while 2**(p+1)<=n:
            p=p+1
        L=np.zeros(t)
        L[t-p-1]=1
        return L+decomp_bin(n-2**p,t)
```

On propose la variante suivante du code qui permet à la fois d'éviter l'évaluation d'une puissance de 2 à chaque itération de la boucle `while` chargée de déterminer la valeur de p pour la prochaine récursion, mais également de dissimuler le paramètre t en appelant la fonction `décomp_bin2(n,t)` dans une autre fonction (`def decomp_bin(n)`) qui renvoie `decomp_bin2(n,0)`:

Listing 12:

```
def decomp_bin(n):
    def decomp_bin2(n,t):
        if n==0:
            return np.zeros(t)
        else:
            p=0
            deux_puiss_p=2**p
            while 2*deux_puiss_p<=n:
                deux_puiss_p=2*deux_puiss_p
                p=p+1
            tp=max(t,p+1)
            L=np.zeros(tp)
            L[tp-p-1]=1
            return L+decomp_bin2(n-2**p,tp)
    return decomp_bin2(n,0)
```

Preuve de l'algorithme:

- Terminaison: dans le cas récursif, n évolue selon une suite d'entiers strictement décroissante, et par conséquent il en est de même pour l'exposant p, ce qui assure de converger vers le cas de base $n = 0$: l'algorithme termine.

- Correction: Supposons qu'à la première récursion, on ait un entier p_1 tel que $2^{p_1} \leq n \leq 2^{p_1+1}$ et l'on décompose n en somme de 2 tableaux comme $L_n = L_{2^{p_1}} + L_{(n-2^{p_1})}$. Au rang suivant, on décompose $n - 2^{p_1}$ avec le premier entier $p_2 < p_1$ tel que $2^{p_2} \leq n - 2^{p_1} \leq 2^{p_2+1}$; l'algorithme renvoie alors: $L_{(n-2^{p_1})} = L_{2^{p_2}} + L_{(n-2^{p_1}-2^{p_2})}$ et ainsi de suite jusqu'au renvoi de $L_{2^{p_{min}}} + [0, 0, \ldots, 0]$ (tableau vide du cas de base). La liste renvoyée sera donc bien la décomposition binaire de n avec $1 * 2^{p_1} + 1 * 2^{p_2} + \ldots + 1 * 2^{p_{min}}$: l'algorithme est prouvé. (attention: $p_{min} \ldots p_1$ n'est pas la suite des entiers!!!)

c· Encadrement du nombre d'appels récursifs:

- lorsque $n = 0$: aucun appel récursif donc $N_{min} = 0$.
- lorsque $n \geq 1$: on suppose que n possède une décomposition binaire telle que tous ses digits sont à 1, i.e. $n = 2^{p_{max}} - 1$.
  - au premier appel récursif l'argument devient: $n_1 = n - 2^p$
  - au second appel récursif l'argument devient: $n_2 = n_1 - 2^{p-1}$
  - au k^{ime} appel récursif, l'argument devient: $n_k = n_{k-1} - 2^{p-(k-1)} = n_{k-1} - 2^{p-k+1}$

  les appels se terminent alors lorsque $k = N_{max} = p+1$. Il suffit donc d'exprimer p en fonction de n:

On sait que $2^p \leq n < 2^{p+1}$ soit $p \ln 2 \leq \ln n \leq (p+1) \ln 2$
et donc $p \leq \ln_2 n \leq p + 1$
soit finalement:

$$\boxed{0 \leq N_{max} < 1 + \ln_2 n}$$

d· Il suffit simplement de tester lequel de l'entier t ou de l'entier $p + 1$ dégagé par la procédure itérative est le plus grand, et de donner au tableau final à renvoyer la bonne dimension i.e. t ou $p + 1$ suivant le gagnant. Cela donne:

Listing 13:

```
def decomp_bin2(n,t):
    if n==0:
        return np.zeros(t)
    else:
        p=0
        while 2**(p+1)<=n:
            p=p+1
        tp=max(t,p+1)
        L=np.zeros(tp)
        L[tp-p-1]=1
        return L+decomp_bin2(n-2**p,tp)

def decomp_bin(n):
    return decomp_bin2(n,0)
```

Exercice n°5: Recherche dichotomique par récursivité

❶ Un premier algorithme peut-être:

- on recherche la présence d'un élément e donné dans une liste initiale L triée donnée.
- on coupe la liste en deux (par division euclidienne) avec m indice entier du milieu de liste. Si l'élément est sur l'indice milieu de liste on renvoie `vrai` et on stoppe.
- on teste ensuite: si l'élément est supérieur à $L[m]$ alors il peut se trouver uniquement dans la sous liste de droite, donc appel récursif à la fonction avec comme argument l'élément e et la sous-liste allant de l'indice $m + 1$ à l'indice final de liste. Sinon, l'élément ne peut se trouver que dans la sous-liste de gauche, donc appel récursif à la fonction avec comme argument e et la sous-liste allant de l'indice initial à l'indice $m - 1$.

❷ Le premier cas de base correspond à un croisement des indices initial et final de la liste, i.e. l'indice initial supérieur à l'indice final.
Le second cas de base correspond à l'élément recherché à l'indice m.

❸ Un premier version du script est par exemple:

Listing 14: dichorec

```
def dichorec(L,e):
    if len(L)==0:
        return False
    m=len(L)//2
    if L[m]==e:
        return True
    else:
        if e<L[m]:
            return dichorec(L[:m],e)
        else:
            return dichorec(L[m+1:],e)
```

Variante possible:

Listing 15:

```
def dichorec(L,e):
    m=len(L)//2
    if L[m]=e:
        return True
    elif len(L)==0:
        return False
    else:
        if e<L[m]:
            return dichorec(L[:m],e)
        else:
            return dichorec(L[m+1:],e)
```

❹ Pour ce second script on a par exemple:

Listing 16: dichorec "variante" avec renvoi de l'indice

```
def dichorec(L,e,a,b):
    if a>b:
        return False
    m=(b+a)//2

    if L[m]==e:
        return True,m
    else:
        if e<L[m]:
            return dichorec(L,e,a,m-1)
        else:
            return dichorec(L,e,m+1,b)
```

Dans ce second script, il s'agit de renvoyer l'indice de l'élément recherché le cas échéant; ceci impose de **conserver la liste initiale dans les appels récursifs afin de ne pas perdre la possibilité de repérer l'élément recherché par son indice**. On doit ainsi travailler sur la "réduction" des indices extrêmes de la liste initiale, et non découper la liste initiale en une sous-liste envoyée dans la récursion.

Exercice n°6: Courbe de Koch et Flocon de Koch par approche récursive

❶ On propose le script suivant:

Listing 17:

```
def Koch(l,n):
    if n==0:
        forward(l)   #cas de base: trace le segment final
    else:
        Koch(l/3,n-1)#lance récursion premier segment
        left(60)     #tourne à gauche de 60 degrés
        Koch(l/3,n-1)#lance récursion second segment ayant "poussé"
        right(120)   #tourne à droite de 120 degrés
        Koch(l/3,n-1)#lance récursion second segment ayant "poussé"
        left(60)     #tourne à gauche de 60 degrés (horizontale)
        Koch(l/3,n-1)#lance  récursion dernier segment
```

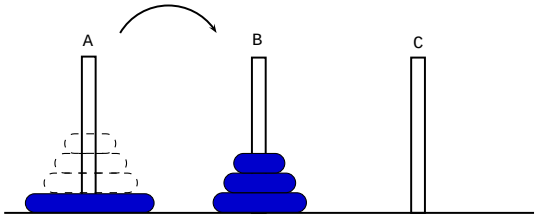
❷ La fonction `FloconKoch(ℓ,n)` exploite la fonction `Koch(ℓ,n)` pour transformer récursivement les 3 côtés d'un triangle équilatéral. Le script est donc:

Listing 18:

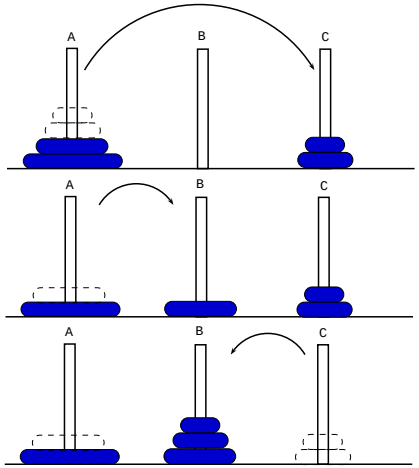
```
def FloconKoch(l,n):
    Koch(l,n) #lance récursion sur premier côté
    right(120)#tourne de 120 degrés
    Koch(l,n) #lance récursion sur second côté
    right(120)#tourne de 120 degrés
    Koch(l,n) #lance récursion sur troisième côté

FloconKoch(200,2) #trace le flocon de Koch de second ordre
mainloop()
```

Exercice n°7: **Les tours de Hanoï**

❶ On rappelle l'étape 1:



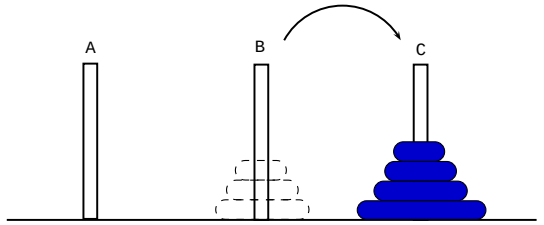
Que l'on peut donc décomposer en ces 3 étapes:



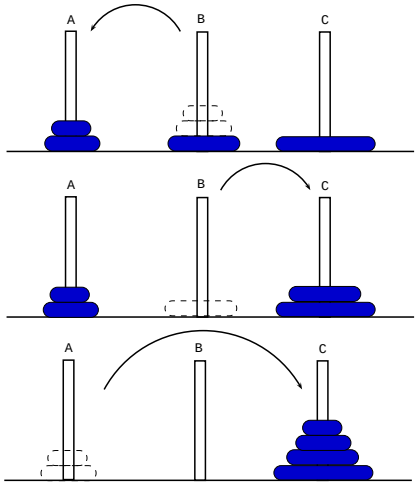
Donc: "déplacer $n-1$ disques de A vers B (en passant forcément par C)" se décompose en:

- Sous-étape 1: Déplacer les $n-2$ disques de A vers C. **NB:** il y aura de la récursivité ici car décomposable en multiples étapes!
- Sous-étape 2: Déplacer le $n-1^{\text{ième}}$ disque (le plus grand) de A vers B
- Sous-étape 3: Déplacer les $n-2$ disques de C vers B. **NB:** il y aura de la récursivité ici car décomposable en multiples étapes!

❷ On rappelle l'étape 3:



Que l'on peut donc décomposer en ces 3 étapes:



Donc: "déplacer $n-1$ disques de B vers C (en passant forcément par A)" se décompose en:

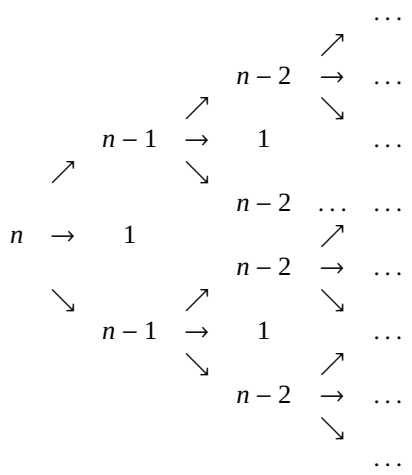
- Sous-étape 1: Déplacer les $n-2$ disques de B vers A. **NB:** il y aura de la récursivité ici car décomposable en multiples étapes!
- Sous-étape 2: Déplacer le $n-1^{\text{ième}}$ disque (le plus grand) de B vers C. **NB:** pas de récursivité ici car empilé sur plus grand disque.
- Sous-étape 3: Déplacer les $n-2$ disques de A vers C. **NB:** il y aura de la récursivité ici car décomposable en multiples étapes!

Listing 19:

❸
```
1            def Hanoi(n,init,final,aux)
2            if n>0: #donc lorsque l'on est en dehors du cas
    de base
3                Hanoi(n-1,init,aux,final)# on déplace les $n-1$
    disques de A vers B en passant par C (étape 1)
4            print(init,"␣vers␣",final)
```

```
Hanoi(n-1,aux,final,init)
```

❹ Pour une résolution à n disques, on déplace une tour de n-1 disques, puis 1 disque, et à nouveau n-1 disques. Pour chaque déplacement de n-1 disques, on fait un déplacement de n-2 disques, 1 disques, et encore n-2 disques; et ainsi de suite. Graphiquement cela donne:



Appelons p l'ordre de l'appel récursif (donc $n-p$ est le nombre de disques à déplacer restants). A partir du niveau p, si u_p désigne le nombre d'étapes du niveau p, alors le nombre d'étapes au niveau suivant $p+1$ est:

$$u_{p+1} = 2u_p + 1$$

Ainsi on a:
$u_1 = 1, u_2 = 2u_1 + 1, \ldots, u_{p-1} = 2u_{p-2} + 1, u_p = 2u_{p-1} + 1$
En multipliant chaque terme par 2^{p-1}, puis 2^{p-2}, ... il vient:

$$2^{p-1}u_1 = 2^{p-1}$$
$$2^{p-2}u_2 = 2^{p-1}u_1 + 2^{p-2}$$
$$\ldots$$
$$2^1 u_{p-1} = 2^2 u_{p-2} + 2^1$$
$$u_p = 2u_{p-1} + 1$$

Cette astuce permet le "téléscopage" correct de la somme membre à membre, qui donne:

$$u_p = 1 + 2^1 + 2^2 + \ldots 2^{p-1}$$

Ainsi u_p est une série de raison 2, de premier terme 1, et comportant p termes:

$$u_p = \frac{1-2^p}{1-2} = 2^p - 1$$

pour n disques on a $p = n$ donc:

$$\boxed{u_n = 2^n - 1}$$

Autre méthode: on peut procéder plus simplement par la recherche d'un point fixe pour dégager la forme explicite de cette suite:

$$x = 2x + 1 \quad \Longrightarrow \quad r = x = -1$$

Dans ces conditions: $u_n = 2^n(\underbrace{u_0}_{=0} - r) + r = 2^n - 1$

❺ Le premier appel récursif, qui correspond au déplacement de n-1 disques, est suivi d'une commande (`print`), puis d'un second appel récursif (toujours pour le déplacement de n-1 disques); une pile d'exécution sera donc nécessaire, et cette récursion est donc non terminale.

Exercice n°8: Usage des piles dans les tours d'Hanoï

❶ Fonction d'affichage de pile:

Listing 20:

```
def afficher_pile(p):
    paux=[]
    if pile_vide(p): #si pile vide on affiche une colonne vide
        print "|","|"
    else: #sinon
        while not(pile_vide(p)): #on dépile p et on affiche
    chaque élément tant que la pile n'est pas vide
            elt=depile(p)
            print "|",elt,"|"
```

```
            empile(paux,elt)
        while not(pile_vide(paux)): #on reconstruit la pile originelle p
            empile(p,depile(paux))
```

❷ Code des tours de Hanoï modifié:

Listing 21:

```
def Hanoi(n,init,final,aux):
    if n>0:
        Hanoi(n-1,init,aux,final)
        disque=sommet_pile(init[1])
        print u"Disque n°",disque,"de",init[0],"vers",final[0]
        empile(final[1],depile(init[1]))
        print u"Contenu de la tour", init[0],":"
        afficher_pile(init[1])
        print u"Contenu de la tour",aux[0],":"
        afficher_pile(aux[1])
        print u"Contenu de la tour",final[0],":"
        afficher_pile(final[1])
        Hanoi(n-1,aux,final,init)

                ### Création des piles ###
                pA=("pA",[])
                pB=("pB",[])
                pC=("pC",[])
                N=5

### remplissage initial pile A ###
for i in range(N,0,-1):
    empile(pA[1],i)

### Lancement résolution ###
Hanoi(N,pA,pC,pB)
```

Exercice n°9: Produit matriciel récursif: exploitation de l'algorithme de Strassen (source Wikipédia)

❶ Ci-dessous le Script Python de la méthode de produit matriciel classique complété:

Listing 22:

```
def prodmatclassique(A,B):
    if A.shape[1]!=B.shape[0]:
        print(u"Matrices incorrectes!!")
        break
    C=numpy.zeros((A.shape[0],B.shape[1]))
    for i in range(A.shape[0]):#on balaie toutes les lignes de A
        for j in range(B.shape[1]):# on balaie toutes les colonnes de B
            for k in range(A.shape[1]):# ou bien B.shape[0]# boucle de calcul de l'élément C[i,j]
                C[i,j]=C[i,j]+A[i,k]*B[k,j]
    return C
```

❷ Si l'on procède par exemple au calcul d'un produit de 2 matrices carrées $n \times n$, alors le calcul de chaque terme de la matrice produit nécessite n multiplications et cette matrice est constituée de n^2 termes; il faudra donc n^3 multiplications. La complexité est donc en $O(n^3)$.

❸ **a**· Script de l'algorithme de Staßen complété:

Listing 23:

```
import numpy
def prodStrassen(A,B):
    dimmat=A.shape[0]
    if dimmat>1:
        dimssmat=dimmat/2

        A11=A[:dimssmat,:dimssmat]
        A21=A[dimssmat:,:dimssmat]
        A12=A[:dimssmat,dimssmat:]
        A22=A[dimssmat:,dimssmat:]
        B11=B[:dimssmat,:dimssmat]
        B21=B[dimssmat:,:dimssmat]
        B12=B[:dimssmat,dimssmat:]
        B22=B[dimssmat:,dimssmat:]

        M1=prodStrassen(A11+A22,B11+B22)
        M2=prodStrassen(A21+A22,B11)
        M3=prodStrassen(A11,B12-B22)
        M4=prodStrassen(A22,B21-B11)
        M5=prodStrassen(A11+A12,B22)
        M6=prodStrassen(A21-A11,B11+B12)
        M7=prodStrassen(A12-A22,B21+B22)
```

```
        #Calcul final des coefficients
        C=numpy.zeros((dimmat,dimmat))
        C[:dimssmat,:dimssmat]=M1+M4-M5+M7
        C[:dimssmat,dimssmat:]=M3+M5
        C[dimssmat:,:dimssmat]=M2+M4
        C[dimssmat:,dimssmat:]=M1-M2+M3+M6
    else:
        return A[0,0]*B[0,0]
    return C
```

b· Lors de chaque appel à la fonction `prodStrassen` pour des matrices carrées de dimension n, celle-ci procède à 7 appels récursifs avec des matrices de dimensions moitié, soit $\frac{n}{2}$. En outre, le bloc de calcul final des coefficients C calcule les $n \times n = n^2$ termes (tous les termes au rang n de la récursion sont évalués dans les 4 lignes de calculs), sauf dans le cas de base où un seul terme est calculé; on a donc:

$$C(n) = 7 \times C\left(\frac{n}{2}\right) + O(n^2)$$

c· En ne gardant que le terme lié au multiplications successives dans la relation de récurrence précédente, que l'on supposera d'ordre le plus élevé (en comparaison du terme $O(n^2)$), on obtient:

$$C(n) = 7 \cdot C\left(\frac{n}{2}\right)$$

Sachant par ailleurs que $n = 2^N$ avec $N \in \mathbb{N}^*$ (hypothèse sur le format des matrices) cette relation devient:

$$C\left(2^N\right) = 7C\left(2^{N-1}\right)$$

et cette récurrence conduit sans peine à:

$$C\left(2^N\right) = 7^N C\left(2^0\right) = 7^N C(1)$$

Sachant que $C(1)$ correspond dans l'algorithme au cas de base pour lequel on procède à une seule multiplication soit $C(1) = 1$ on a:

$$C\left(2^N\right) = 7^N$$

Enfin comme $n = 2^N \implies N = \log_2 n$ il vient:

$$C(n) = 7^{\log_2 n} = n^{\frac{\ln 7}{\ln 2}} \simeq n^{2,8}$$

Ainsi (et c'est en tout cas la promesse du calcul!), l'algorithme de Straßen permet une petite amélioration de complexité par rapport au calcul classique du produit matriciel. L'écart étant ténu, et la complexité évaluée ne tenant pas compte des opérations réalisées en dehors des multiplications, l'amélioration devrait être perceptible uniquement pour des matrices de grandes tailles.

Exercice n°10: Tri par insertion dichotomique

❶ Algorithmique

a·

Listing 24:

```
def position(x,y):
    if x<y:
        return -1
    elif x==y:
        return 0
    else:
        return 1
```

b·

Listing 25:

```
def rechdicho(Ltri,el):
    g,d=0,len(Ltri)-1
    while g<=d:
        m=(g+d)//2
        pos=position(Ltri[m],el)
        if pos==0:
            return m
        if pos==1:
            d=m-1
        else:
            g=m+1
    return g
```

c· Code complété:

Listing 26:

```
def tridicho(L):
    Ltri=[]
    Ltri.append(L[0])
    for el in L[1:]:
```

```
        pos=rechdicho(Ltri,el)
        Ltri.insert(pos,el)
    return Ltri
```

❷ Complexité

a·

- **au rang** 1**:**

$$d_1 - g_1 \overset{d_1=m-1}{=} \left(E\left[\frac{g_0+d_0}{2}\right]-1\right) - g_0 < \frac{d_0+g_0}{2} - g_0 = \frac{d_0-g_0}{2} < \frac{n}{2}$$

ou

$$d_1 - g_1 \overset{g_1=m+1}{=} d_0 - \left(E\left[\frac{g_0+d_0}{2}\right]+1\right) < d_0 - \frac{g_0+d_0}{2} = \frac{d_0-g_0}{2} < \frac{n}{2}$$

- **au rang** 2**:**

$$d_2-g_2 \overset{d_2=m-1}{=} \left(E\left[\frac{g_1+d_1}{2}\right]-1\right)-g_1 < \frac{d_1+g_1}{2}-g_1 = \frac{d_1-g_1}{2} < \frac{1}{2}\frac{d_0-g_0}{2} < \frac{n}{2^2}$$

ou

$$d_2-g_2 \overset{g_2=m+1}{=} d_1-\left(E\left[\frac{g_1+d_1}{2}\right]+1\right) < d_1-\frac{g_1+d_1}{2} = \frac{d_1-g_1}{2} =< \frac{1}{2}\frac{d_0-g_0}{2} < \frac{n}{2^2}$$

- **à fortiori au rang** k**:**

$$d_k - g_k < \frac{d_{k-1}-g_{k-1}}{2} < \frac{d_{k-2}-g_{k-2}}{2^2} < \frac{d_0-g_0}{2^k} < \frac{n}{2^k}$$

$\Rightarrow$ on prouve la proposition demandée.

b· Calcul immédiat: si N représente le nombre maximal d'itérations nécessaires à la recherche d'un élément, alors à la $N-1^{\text{ième}}$ itération l'intervalle entre g et d est réduit à 1, soit: $d_{N-1}-g_{N-1}=1$:

$$1 < \frac{n}{2^{N-1}} \Rightarrow N < ln_2(n)+1$$

d'où une estimation de la complexité maximale:

$$\boxed{C(n) = O(\ln_2 n)}$$

c· La fonction de `tridicho(L)` balaie la liste à trier par une boucle simple, donc de complexité linéaire. Elle fait appel à la fonction de recherche dichotomique N fois au total, et à chaque appel la complexité de la recherche est de $O(\ln N_i)$ avec N_i le nombre d'éléments de la liste à trier (i.e. en construction) $N_i = 2, 3 \ldots N$ (la liste initialisée avec 1 seul élément n'est pas à trier). La complexité est donc au total **dans le pire des cas**:

$$C(N) = \sum_{N_i=2}^{N} \ln N_i$$

En supposant un nombre importants d'éléments dans la liste à trier, l'approximation de Stirling donne:

$$\boxed{C(N) \sim N\ln N - N \sim N \ln N}$$

Exercice n°11: Tri par dénombrement ou tri fréquence

❶ Fonction `compte(L,N)` 1ère version avec 2 boucles (la plus lourde):

Listing 27:

```
def compte(L,N):
    frequence=[0]*N
    for i in range(0,N):
        j=0
        while j<len(L):
            if L[j]==i:
                frequence[i]=frequence[i]+1
            j+=1
    return frequence

```

Fonction `compte(L,N)` 2nde version exploitant la commande `in` bien pratique mais spécifique à Python:

Listing 28:

```
def comptage(L,N):
    frequence = [0] * N
    for n in L:
        frequence[n] += 1
    return frequence
```

`compte(L,N)` 3ième version:

Listing 29:

```
def comptage(L,N):
```

```
    frequence = [0] * N
    for k in range(len(L)):
        frequence[L[k]] += 1
    return frequence
```

et enfin 4ième version:

Listing 30:

```
def compte(L):
    N=max(L)
    frequence=(N+1)*[0]
    for k in L:
        frequence[k]+=1
    return frequence
```

❷ Fonction `tri(L,N)`:

Listing 31:

```
def tri(L,N):
    aux=[]
    frequence=compte(L,N)
    for i in range(N):
        aux=aux+frequence[i]*[i]
    return aux
```

❸ Cet algorithme réalise un tri "en place":seule la liste `L` est exploitée pour être renvoyée triée $\Longrightarrow$ complexité spatiale réduite.

La liste `L` donne le tableau de fréquence suivant: $P = [0, 2, 2, 2, 2, 1, 0, 2]$. Les premières étapes d'application de l'algorithme `tri2(L,N)` sur la liste `L` sont les suivantes:

$i = 0$ $j = 0$ $x = 0$	k=0	L=[7,1,3,1,2,4,5,7,2,4,3]
$i = 0$ $j = 1$ $x = 2$	k=0	L=[**1**,1,3,1,2,4,5,7,2,4,3]
$i = 1$ $j = 1$ $x = 2$	k=1	L=[**1**,**1**,3,1,2,4,5,7,2,4,3]
$i = 2$ $j = 2$ $x = 2$	k=0	L=[**1**,**1**,**2**,1,2,4,5,7,2,4,3]
$i = 3$ $j = 2$ $x = 2$	k=1	L=[**1**,**1**,**2**,**2**,2,4,5,7,2,4,3]
$i = 4$ $j = 3$ $x = 2$	k=0	L=[**1**,**1**,**2**,**2**,**3**,4,5,7,2,4,3]
$i = 5$ $j = 3$ $x = 2$	k=1	L=[**1**,**1**,**2**,**2**,**3**,**3**,5,7,2,4,3]
$i = 6$ $j = 4$ $x = 2$	k=0	L=[**1**,**1**,**2**,**2**,**3**,**3**,**4**,7,2,4,3]
...		

❹ Question optionnelle:

Listing 32:

```
def genliste(N):
    i=0
    L=[]
    while i<N:
        L.append(rd.randint(0,5))
        i+=1
    return L
```

❺ **Calcul de complexité de `compte(L,N)` 1ère version:**

On a déjà N opérations pour créer la liste `frequence`, donc $O(N)$. Ensuite cette fonction exploite 2 compteurs: l'un (indice i) qui balaie les valeurs de tous les entiers que l'on peut rencontrer dans la liste, soit de 0 à $N-1$ donc N itérations; et l'autre (indice j) qui balaie tous les éléments de la liste, soit $n = len(L)$: ainsi, la complexité est en $O(max(N, n \times N)) = O(n \times N)$

Calcul de complexité de `compte(L,N)` 2de version:

Il y a toujours la création de la liste donc $O(N)$; on ajoute ensuite la complexité de la boucle soit $O(n)$, donc finalement:

$$O(max(N, n))$$

Complexité de la fonction `tri2(L,N)`:

on compte déjà la complexité de la fonction `compte(L,N)`; il faut lui ajouter la complexité du mécanisme de reconstitution de la liste; or il n'est pas immédiat de calculer celle-ci en sommant les opérations de boucles, car la seconde boucle de bornes maximale `range(x)` dépend de la structure de la liste. On peut faire beaucoup plus simple: le nombre d'affectations nécessaires pour `L` est égal au nombre d'éléments dans la liste soit `n` donc $O(n)$; par ailleurs la première boucle est de toute façon en $O(N)$.

Finalement, l'ensemble est en $O(max(N, n * N)) = O(max(n * N))$.