TD IPT^2 n° 3: Révisions
Récursivité - tris

Algorithmique simple de la récursivité

Exercice n°1: **Inversion récursive**

On cherche à écrire une fonction récursive qui inverse l'ordre des éléments dans un tableau d'entiers.

❶ Proposer une décomposition du problème permettant un appel récursif i.e. une technique qui amène les paramètres à converger vers au moins **un cas de base**.

❷ Quel(s) est (sont) le(s) cas de base?

❸ Elaborer un script Python récursif de l'inversion de liste.

Exercice n°2: **Fonction `mystere`**

On considère la fonction `mystere(t,k)` où T est un tableau d'entiers non vide et k vérifiant $0 \leq k < len(T)$.

Listing 1:

```
1 def mystere(T,k):
2     if k == len(T)-1 :
3         return True
4     if T[k]>T[k+1] :
5         return False
6     return mystere(T,k+1)
```

❶ Soit $T = [6, 9, 4, 8, 12]$

a· Que retourne `mystere(T,2)`? Indiquer en particulier le contenu complet de la pile d'exécution.

b· Que retourne `mystere(T,0)`? Indiquer là-encore le contenu de la pile d'exécution.

❷ Que fait la fonction `mystere` dans le cas général?

❸ Quel est le nombre maximum d'appels récursifs (en fonction de n et k) de la fonction `mystere(T,k)` si le tableau T est de longueur n?

❹ En utilisant la fonction `mystere`, écrire une fonction `estCroissant(T)` qui retourne `True` si la suite d'entiers contenue dans le tableau T est croissante et retourne `False` sinon.

❺ Sur le même schéma de principe que la fonction `mystere`, écrire une fonction récursive `estDans(T,x,k)` qui retourne `True` si x apparait dans le tableau T à partir de l'indice k, `False` sinon.

Exercice n°3: **Suite de Fibonacci récursive rapide**

Nous avons exhibé dans le cours le caractère «gourmand» du calcul des termes de la suite de Fibonacci par la récursivité, chaque appel récursif en engendrant deux! L'exercice qui suit propose d'étudier un algorithme permettant une complexité en $O(\ln n)$ de ce calcul.

On rappelle que la suite de Fibonacci $\mathcal{F}$ de premiers termes a et b se calcule selon le schéma suivant:

$$\mathcal{F}_1 = a \qquad \mathcal{F}_0 = b \qquad \mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2} \quad \text{pour } n \geq 2$$

La définition de la suite de Fibonacci peut également s'écrire matriciellement en remarquant que:

$$\begin{bmatrix} \mathcal{F}_n \\ \mathcal{F}_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \mathcal{F}_{n-1} \\ \mathcal{F}_{n-2} \end{bmatrix}$$

❶ Ecrire une relation matricielle entre $\begin{bmatrix} \mathcal{F}_n \\ \mathcal{F}_{n-1} \end{bmatrix}$ et $\begin{bmatrix} \mathcal{F}_1 \\ \mathcal{F}_0 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$.

❷ Ecrire une fonction récursive `Puiss(M,n)` qui renvoie M^n avec M une matrice carrée, et n un entier naturel. Cette fonction aura une complexité en $O(ln(n))$ produits matriciels.

❸ En déduire un calcul rapide de $\mathcal{F}_n$.

❹ *Question «pratique»:* on pourra, en utilisant le module `time`, comparer les performances du calcul classique des termes de la suite Fibonacci et de celui basé sur cette technique matricielle.

Exercice n°4: Décomposition binaire récursive

Cet exercice propose d'écrire une fonction récursive qui associe à un nombre entier sa représentation binaire. Soit $n \in \mathbb{N}$ un entier naturel; on rappelle que $n_{(b)}$ sa représentation en base $b > 1$ est la suite des symboles ou chiffres représentant des nombres de zéro à $b-1$ telle que:

$$n_{(b)} = a_p a_{p-1} \dots a_0 \;\Leftrightarrow\; n = \sum_{k=0}^{p} a_k b^k$$

On souhaite ici représenter n en base 2 par un tableau qui sera donc formé exclusivement de t chiffres 0 ou 1.

❶ **Quelques notions préliminaires**

- **a**· Écrire une fonction qui renvoie l'unique entier p tel que 2^p est la plus grande puissance de 2 inférieure ou égale à n.
- **b**· Prouver ce dernier algorithme.
- **c**· Cet entier connu, quelle est la longueur de la liste représentant n en binaire? Quelle relation cela impose-t-il entre t et p?
- **d**· En observant que $n = 2^p + (n-2^p)$, montrer que la liste représentant n est la somme de celles représentant 2^p et $n-2^p$.

❷ **Construction du programme récursif**

- **a**· Décrire les étapes d'un algorithme récursif, d'entrées deux entiers naturels n et t, et qui retourne dans un tableau de longueur t ne contenant que des 0 et 1 la suite des chiffres de n en base 2. On supposera dans un premier temps que la longueur de t est suffisante pour stocker des eniters positifs `int` codés en machine sur 32 bits.
- **b**· Ecrire cet algorithme et prouver ce dernier.
- **c**· Donner un encadrement du nombre d'appels récursifs en fonction de n.
- **d**· Modifier le programme de telle sorte que la valeur t soit arbitraire et retourne le tableau de longueur optimale représentant l'entier n en base 2.

Exercice n°5: Recherche dichotomique par récursivité

On veut établir ici une méthode récursive de la recherche dichotomique d'un élément **dans une liste triée**. On rappelle que la dichotomie consiste à diviser systématiquement la liste initiale en deux (par division entière) en retenant la sous-liste dans laquelle se trouve à priori l'élément; en diminuant ainsi la taille de la liste, on finit par tomber sur l'élément recherché ou bien obtenir une liste vide si l'élément n'est pas présent.

A chaque appel récursif, il faut donc savoir entre quels indices on recherche l'élément dans la liste.

❶ Ecrire un premier script Python récursif de la recherche dichotomique qui renvoie `True` si l'élément est présent et `False` sinon.

❷ Elaborer un second script Python récursif de la recherche dichotomique qui renvoie cette fois `True` ainsi que l'indice de l'élément si ce dernier est présent, et `False` dans le cas contraire. Commenter les différences des deux scripts.

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

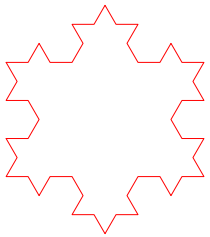
Le flocon de Koch constitue la première création de figure fractale; elle fut inventée par le mathématicien suédois Niels Fabian Helge von Koch en 1904, et sa construction informatique est particulièrement simple et adaptée lorsque l'on fait appel à la récursivité. L'exercice qui suit a pour but de rédiger un script python permettant le tracé de cette fractale.

La courbe de Koch, élément de base du Flocon de Koch, s'obtient de la manière suivante: on divise un segment en trois segments égaux, et on fait "pousser" un triangle équilatéral dont le segment médian des trois segments précédents est la base. On fait disparaître enfin le segment de base. Le processus est répété autant de fois que l'on souhaite.

On obtient le Flocon de Koch lorsque cette procédure est appliquée aux 3 côtés d'un triangle équilatéral.

On obtient ainsi après une première étape une ligne brisée constituée de 12 segments. On réitère ce processus sur ces 12 segments, etc...

Par exemple, après avoir appliqué deux fois ce procédé, on obtient la figure suivante:



On se propose de rédiger une fonction récursive en Python, prenant deux paramètres `longueur` et `n` et permettant de tracer le flocon obtenu après `n` itérations, chaque segment ayant une longueur ℓ. Pour ce faire, on peut utiliser le module `turtle` dont on importera toutes les fonctions en tapant:

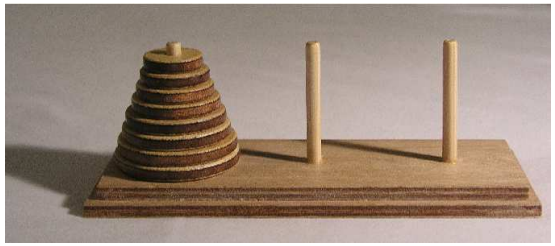
```
from turtle import *
```

Dans ce module on dirige une «tortue[1]», qui part du point $(0,0)$ avec pour angle initial zéro. Les commande `forward(d)` et `backward(d)` permettent respectivement de faire avancer ou reculer la tortue d'une distance `d`. Les commandes `left(a)` ou `right(a)` font tourner celle-ci d'un angle `a` (exprimé en degré) vers la gauche ou vers la droite (respectivement en sens trigo- et antitrigonométrique). Enfin, la commande `turtle.mainloop()` permet de garder le tracé à l'écran (en l'absence de cette commande, le tracé disparait dès que le programme se termine).

❶ Rédiger un premier script Python récursif `Koch (`ℓ`,n)` permettant de tracer la courbe de Koch correspondant à n étape(s) de "cassure" de ligne.

❷ Rédiger alors une fonction Python `FloconKoch(`ℓ`,n)` exploitant `Koch(`ℓ`,n)` et permettant de réaliser un Flocon de Koch après n applications du procédé, chaque segment ayant la longueur ℓ.

Exercice n°7: Les tours de Hanoï



Le jeu des `tours de Hanoi` est un casse-tête inventé par le mathématicien français Edouard Lucas en 1883 lors d'exercices "récréatifs" inspirés par un prétendu ami, un certain N. Claus de Siam, plus probablement un simple anagramme de Lucas d'Amiens (ville d'origine d'Edouard Lucas).

Le jeu consiste à déplacer des disques de diamètres différents d'une tour de départ nommée A (celle de gauche) à une tour d'arrivée nommée C (celle de droite) en passant par une tour auxiliaire nommée B, ceci en respectant deux règles:

- on ne peut déplacer qu'un disque à la fois.
- on ne peut poser sur un disque qu'un disque de diamètre inférieur.

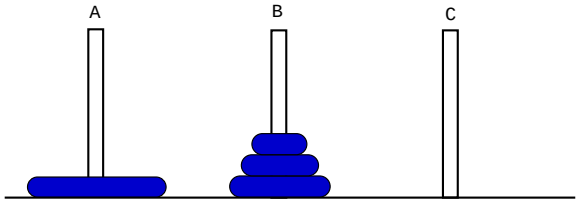
NB: nous partirons d'une situation de départ dans laquelle cette dernière règle est vérifiée, à savoir tous les disques sont correctement empilés sur la tour de gauche (photographie ci-dessus ou schéma plus bas).

L'objectif de cet exercice est de programmer une **fonction** qui donne la suite des manipulations à effectuer pour terminer le jeu c'est à dire obtenir tous les disques correctement empilés sur la tour de droite.

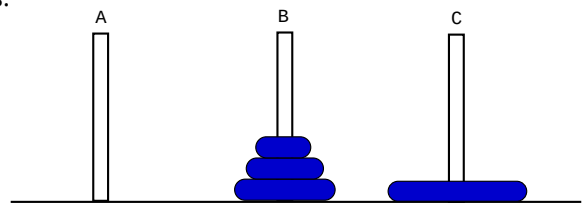
La démarche de déplacement est la suivante:

- **s'il n'y a qu'un seul disque, on le déplace de la tige A à la tige C**
- **si on suppose être capable de déplacer une pyramide de $n-1$ disques d'une tige sur une autre, alors pour déplacer n disques de la tige A vers la tige C, il faut:**
  - déplacer la pyramide des $n-1$ disques de la tige A vers la tige B, sans toucher au disque n (le plus grand), la situation étant alors par exemple avec n=4 disques:

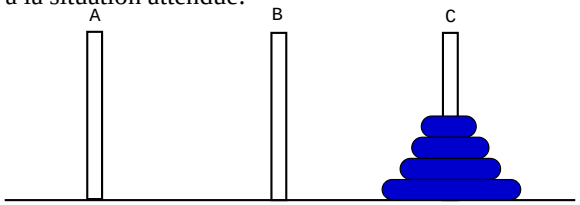
[1] c'est un héritage du langage pédagogique Logo inventé dans les années 60 au MIT et qui fut très employé comme outil pédagogique à destination des écoliers du primaire dans les années 80; il est encore aujourd'hui en constante évolution.



– puis de déplacer le disque n de la tige A vers la tige C, cette situation intermédiaire étant alors:



– et enfin de déplacer la pyramide de la tige intermédiaire B sur la tige C pour aboutir finalement à la situation attendue:



Résumons-nous; 3 étapes essentielles:

- Etape 1: déplacer $n-1$ disques de A vers B (en passant forcément par C)
- Etape 2: déplacer le plus grand disque (n) de A vers C.
- Etape 3: déplacer $n-1$ disques de B vers C (en passant forcément par A)

Naturellement, les règles du jeu imposent de réaliser les étapes 1 et 3 en plusieur sous-étapes.

❶ Décomposer l'étape 1 en 3 sous étapes, ceci afin de déplacer $n-1$ (ici 3) disques de la tige A vers la tige B selon le même protocole

❷ De même, décomposer l'étape 3 en 3 sous-étapes. On pourra naturellement reprendre les sous étapes du ❶ en modifiant correctement le nom des tiges.

❸ A partir de cette analyse, programmer une fonction Python récursive `Hanoi(n,init, final, aux)` qui prend en arguments:

- `n`: le nombre de disques
- `init`: la chaine de caractères désignant la tour de départ, soit "A" lors du premier appel à la fonction.
- `final`: la chaine de caractères désignant la tour d'arrivée, soit "C" lors du premier appel à la fonction.
- `aux`: la chaine de caractères désignant la tour auxiliaire, soit "B" lors du premier appel à la fonction.

et **qui affiche tous les déplacements de disques nécessaires pour résoudre le jeu**.

❹ Déterminer le nombre de déplacements nécessaires pour terminer le jeu. On pourra confirmer cela avec un compteur et le faire afficher en fin de jeu.

❺ Cette récursivité est-elle terminale?

Exercice n°8: Complément sur les tours de Hanoï: matérialisation par des piles

On souhaite compléter le code récursif de résolution des tours d'Hanoï afin de permettre l'affichage du numéro du disque que l'on déplace d'une tour à l'autre et également l'état de remplissage de chaque tour à tout instant. On propose pour cela d'exploiter des piles.

❶ Proposer une fonction `afficher_pile(p)` permettant l'affichage de la pile `p` en respectant le format suivant:

Contenu de la tour i:
1
2
3
4
5

❷ On propose de matérialiser les tours A,B, et C sous forme de trois piles `pA`, `pB`, et `pC`. Les disques sont initialement empilés sur la tour A dans l'ordre de taille décroissante. Si n est le nombre de disques à déplacer, on notera n la taille du plus grand, $n-1\ldots$ 1 celle des suivants. On veut modifier le code récursif des tours de Hanoï (cf ex n°7) afin d'afficher d'une part par le numéro du disque à déplacer d'une tour à l'autre, et également l'état de remplissage des trois tours après chaque déplacement.

Algorithmique type «Diviser pour régner»

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

La technique habituelle de calcul d'un produit matriciel $A \times B$ consiste à évaluer les coefficients de la matrice résultante un à un, soit:

$$C = A \times B \quad \Leftrightarrow \quad \begin{bmatrix} c_{11} & c_{12} & \dots \\ c_{21} & c_{22} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots \\ a_{21} & a_{22} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \dots \\ b_{21} & b_{22} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

avec $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj}$

❶ Compléter l'implémentation proposée de la technique "conventionnelle":

Listing 2:

```
def prodmatclassique(A,B):
    if A.shape[1]!=B.shape[0]:
        print(u"Matrices incorrectes!!")
        return
    C=numpy.zeros((A.shape[0],B.shape[1]))
    for i in range(.....):
        for j in range(.....):
        ......
    return ......
```

❷ Evaluer la complexité du calcul classique d'un produit matriciel de deux matrices carrées $n \times n$.

❸ En 1969, le mathématicien allemand Volker Straßen propose un algorithme de calcul du produit de deux matrices carrées $n \times n$ de complexité améliorée par rapport au calcul naïf.

- Les matrices A et B (et donc la matrice résultat C) sont toujours carrées de taille $n \times n$ où n est une puissance de 2 (si ce n'est pas le cas originellement, il suffit de compléter les matrices par des 0 pour se ramener à cette situation!)
- On divise les trois matrices A, B, et C en blocs de matrices de taille 2×2, soit:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad , \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad , \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

structures dans lesquelles $A_{i,j}, B_{i,j}, C_{i,j} \in R^{n/2} \times R^{n/2}$
Dans ces conditions, on a:

$$\begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22} \end{cases}$$

En procédant récursivement, on peut reproduire ce calcul jusqu'à ce que les matrices A et B soient de taille 1.

On constate qu'à chaque récursion cette technique met en oeuvre 8 multiplications de matrices pour calculer chaque matrice bloc C_{ij}.

Straßen a proposé en 1969 l'introduction de 7 matrices intermédiaires M_k $k = \{1 \ldots 7\}$ limitant le calcul des matrices blocs C_{ij} à 7 multiplications:

$$\begin{cases} M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22}) \times B_{11} \\ M_3 = A_{11} \times (B_{12} - B_{22}) \\ M_4 = A_{22} \times (B_{21} - B_{11}) \\ M_5 = (A_{11} + A_{12}) \times B_{22} \\ M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{cases}$$

et les C_{ij} qui s'expriment alors:

$$\begin{cases} C_{11} = M_1 + M_4 - M_5 + M_7 \\ C_{12} = M_3 + M_5 \\ C_{21} = M_2 + M_4 \\ C_{22} = M_1 - M_2 + M_3 + M_6 \end{cases}$$

NB: *on constate cependant que s'il y a réduction du nombre de multiplications, le nombre d'additions matricielles est bien plus important (mais on le postule dans tous les cas moins coûteux).*

a. Compléter le code ci-dessous qui implémente l'algorithme de Staßen:

Listing 3:

```
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=.....
        M2=.....
        M3=.....
        M4=.....
```

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

b· Justifier simplement que la complexité de l'algorithme de Straßen vérifie la relation de récurrence suivante:

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

c· En déduire que la complexité est en $O(n^{\log_2 7}) \simeq O(n^{2,80})$. Conclure.

Révisions sur les tris

Exercice n°10: Tri par insertion dichotomique

Le tri par insertion dichotomique consiste à bâtir, à partir d'une liste à trier *L*, une liste *Ltri* dans laquelle on insère les nouveaux éléments extraits de *L* par dichotomie; ce n'est donc pas un tri sur place.

L'algorithme en langage naturel est le suivant:

- on initialise la liste *Ltri* avec le premier élément de *L*, donc $Ltri = [L[0]]$
- on parcourt *L* et l'on extrait tous ces éléments en les insérant à la bonne place dans *Ltri* par technique de dichotomie puisque la liste en construction est par définition triée.

On propose de construire un script python réalisant ce tri

❶ Ecrire une fonction python `position(x,y)` qui renvoie $-1, 0, 1$ suivant que respectivement $x < y$, ou $x = y$, ou $x > y$.

❷ Ecrire une fonction python `rechdicho(Ltri,el)` qui prend en argument la liste `Ltri`, un élément `el` de la liste `L` à trier, et la fonction `position`, et qui en utilisant la fonction `position` et la technique de dichotomie renvoie la position à laquelle on devra placer `el` dans `Ltri`.

❸ Compléter enfin le code suivant de la fonction `tridicho(L)` pour qu'elle renvoie la liste triée:

Listing 4:

```
def tridicho(L):
    Ltri=[]
    Ltri.append(L[0])
    for el in ......:
        pos=.......
        Ltri.insert(pos,el)
    return Ltri
```

❹ Déterminer la complexité au pire du tri dichotomique. Conclure quant à son efficacité. On pourra exploiter l'approximation de Stirling rappelée ci-dessous:

$$\lim_{n\to+\infty} \sum_{k=2}^{n} \ln k = n \cdot \ln n - n$$

Exercice n°11: Tri par dénombrement ou tri fréquence

On se propose de trier une liste `L` d'entiers positifs strictement inférieurs à *N* par la méthode dite de "tri par dénombrement" (intéressante lorsque *N* n'est pas trop grand):

❶ Ecrire une fonction `compte(L,N)` renvoyant une liste `frequence` dont le `k`$^{\text{ième}}$ élément désigne le nombre d'occurences de l'entier `k` dans la liste `L`.

❷ Bâtir une fonction `tri(L,N)` exploitant la fonction `compte(L,N)` et qui renvoie finalement la liste triée sous forme d'une liste auxiliaire `aux`. (ce tri ne se fait donc pas "sur place" puisqu'il exploite une liste supplémentaire, augmentant ainsi la complexité spatiale).

❸ On propose le code alternatif suivant pour la fonction `tri2(L,N)`:

Listing 5:

```
def tri2(L,N):
    P=compte(L,N)
    i=0
    for j in range(N):
        x=P[j]
        for k in range(x):
            L[i]= j
            i+=1
    return L
```

Expliquer l'intérêt de cet algorithme par rapport à celui développé en question ❷. Dérouler cet algorithme sur la liste suivante:

$$L = [7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3]$$

❹ Question optionnelle à faire sur machine:
Tester la fonction `tri(L,N)` sur une liste de 20 entiers inférieurs ou égaux à 5 tirés aléatoirement. On rappelle que la fonction *randint*(N_i, N_f) renvoie un nombre entier compris entre N_i et N_f inclus.

❺ Calculer la complexité en gros de la fonction `tri(L,N)`. Faire de même avec la fonction alternative `tri2(L,N)`.