TD IPT^2 n° 3: Récursivité 1

Principes fondamentaux et mise en oeuvre

Preuve, complexité, 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: Retour sur l'exponentiation rapide - Preuve, complexité

Pour $x \in \mathbb{R}$ et $n \in \mathbb{N}$, le calcul de x^n peut être réalisé simplement par une méthode récursive naïve exploitant le fait que $x^n = x \times x^{n-1}$.

❶ Proposer une fonction Python `exponaive(x,n)` exploitant ce schéma élémentaire.

❷ Quelle en est la complexité?

Tout entier n peut s'écrire comme une somme de puissances de 2 (décomposition binaire); de cette remarque, il découle que:

$$x^n = x^{\sum_{i=0}^{d} b_i 2^i} = \prod_{i=0}^{d} x^{b_i \cdot 2^i} = \prod_{i=0}^{d} \left(x^{2^i}\right)^{b_i} \text{ avec } b_i = \{0,1\}$$

Par cette écriture, on remarque que x^n s'écrit comme un produit de puissance de x^{2^i}. On en tire alors le principe d'exponentiation rapide que l'on rappelle:

$$x^n = \begin{cases} 1 \text{ si } n = 0 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}} \text{ si } n \text{ pair} \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}} \text{ si } n \text{ impair} \end{cases}$$

On propose le script suivant:

Listing 1:

```
def exprapide(x,n):
    if n==0:
        return 1
    else:
        q,r=n//2,n%2
        p=exprapide(x,q)*exprapide(x,q)
        if r==1:
            p=p*x
        return p
```

❸ Prouver l'algorithme. On procèdera comme d'habitude par récurrence, en exploitant ici le fait que $n = 2q + r$ avec $r = \{0, 1\}$ (définition de la division Euclidienne par 2)

❹ Montrer par une approche simple que la complexité en terme de nombre d'appels récursifs est de l'ordre de $O(\log_2 n)$.

Exercice n°3: 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 2:

```
def mystere(T,k):
    if k == len(T)-1 :
        return True
    if T[k]>T[k+1] :
        return False
    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°4: Série de décomposition d'un entier

On considère $(k,n) \in \mathbb{N}^{2*}$. On appelle décomposition de l'entier n en k termes, une suite $(x_1, x_2, \ldots, x_k)$ d'entiers supérieurs ou égaux à 1 tels que:

$$n = \sum_{i=1}^{k} x_i$$

On remarquera que la décomposition de $n > 1$ en 2 termes peut s'écrire: $[n-1,1]$, $[n-2,2]$,$[n-3,3]$,... $[1,n-1]$, puis que celle de $n-1$ donne $[n-2,1]$,$[n-3,2]$... $[1,n-2]$, et qu'on obtient une décomposition de n en k termes en concaténant i à la décomposition de $n-i$ comportant $k-1$ termes de manière récursive.

❶ Procéder "à la main" à la décomposition en $k = 3$ entiers de $n = 5$.

❷ Ecrire une fonction récursive `decomp(n,k)` qui retourne une liste des décompositions de n en k termes ($k \geq 1$).

Exercice n°5: 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:

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

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°6: 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°7: 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 l'algorithme récursif de la recherche dichotomique.

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

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

❹ 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°8: Exemple de suite récurrente

On considère la suite (u_n) définie par :

$$u_0 = 1 \quad \text{et} \quad u_{n+1} = \sum_{k=0}^{n} \binom{n}{k} u_k \text{ si } n \geqslant 0$$

Il est facile de calculer les termes de cette suite en programmant une fonction récursive, en supposant avoir défini une fonction `fact` qui calcule les factorielles.

Listing 3: Suite récurrente

```
def suite(n) :
    if n == 0 :
        return 1
    else :
        s = 0
        for k in range(n) :
            s = s + fact(n-1) * suite(k) / fact(k) / fact(n-1-k)
        return s
```

❶ Programmer effectivement la fonction `fact`.

❷ Pour mesurer l'efficacité de cette fonction, on détermine le nombre de fois où la fonction est exécutée pour calculer effectivement le terme u_n. On note α_n le nombre de fois où l'on fait appel à la fonction `suite` pour calculer le terme u_n. On a $\alpha_0 = 1$ et $\alpha_1 = 2$ (le calcul de u_1 exécute un appel à `suite`, qui nécessite de connaître u_0 donc un deuxième appel de la fonction `suite`).
Déterminer la valeur de α_{n+1} en fonction de $\alpha_0, \alpha_1, \ldots, \alpha_n$.

❸ En déduire que $\alpha_n = 2^n$ pour $n \geqslant 1$.
Que peut-on en déduire quant à l'efficacité de cette fonction ?

❹ La lenteur du calcul provient du fait que les premiers termes de la suite sont calculés un grand nombre de fois alors qu'ils suffirait de les mémoriser afin de les réutiliser. On peut alors proposer une autre fonction récursive qui calcule les termes de la suite (u_n) mais qui créée une liste contenant ces termes :

- On initialise avec la liste [1] qui ne contient que u_0.
- On calcule u_1 à partir de cette liste à un élément et on le rajoute à la fin de la liste : on obtient $[1, 1]$, c'est la liste $[u_0, u_1]$
- Si on suppose avoir construit la liste $[u_0, u_1, \ldots, u_{n-1}]$, pour calculer u_n, il suffit d'utiliser les éléments de cette liste puis de rajouter le résultat en queue de liste.

Ecrire une fonction recursive qui suit cet algorithme et comparer son efficacité avec la fonction `suite`.

Exercice n°9: 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.

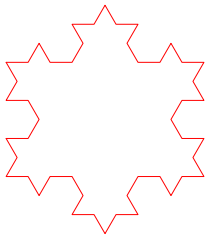
La courbe de Koch, élément de base du Flocon de Koch, s'obtient de la manière suivante: on divise un segment en trois segments égaux, et on fait "pousser" un triangle équilatéral dont le segment médian des trois segments précédents est la base. On fait disparaître enfin le

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.

Algorithmique type «Divide ut imperes[2]»

Exercice n°10: **Recherche récursive du maximum d'une fonction échantillonnée - extension au cas de la recherche du minimum d'une fonction par méthode du nombre d'or (algorithme de Jack Kiefer (1953))**

On considère une fonction $f(x)$ unimodale présentant un minimum sur l'intervalle de travail considéré ici. On échantillonne cette fonction avec un pas quelconque (échantillonnage régulier ou pas) sur cet intervalle pour former **une liste** de valeurs $fech$.

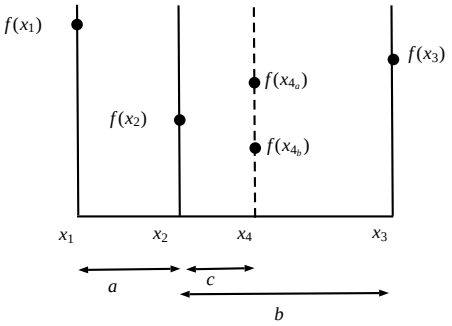
On souhaite déterminer la valeur maximale échantillonnée .

❶ Ecrire une fonction récursive `echant_mini(fech)` exploitant le paradigme «diviser pour régner» et renvoyant l'échantillon de valeur maximale.

❷ Estimer la complexité de cette fonction.

❸ On souhaite désormais employer une méthode récursive du même type permettant la recherche du minimum de la fonction f (non échantillonnée cette fois) et on se propose d'exploiter pour cela la méthode dite "du nombre d'or" dont le principe est le suivant:

- soit trois valeurs x_1, x_2, x_3 de la variable x telles que $x_1 < x_2 < x_3$.
- On suppose que $f(x_2) < f(x_1)$ et $f(x_2) < f(x_3)$. Comme f est unimodale sur l'intervalle $]x_1, x_3[$ on sait alors que le minimum se trouve dans l'intervalle $]x_1; x_3[$.



[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.
[2] «Diviser pour régner»: technique algorithmique de subdivision d'un problème en sous-problèmes plus petits afin de limiter l'«effort». Cette technique vise notamment à améliorer la complexité temporelle.

- On prend un point x_4 soit dans l'intervalle $]x_1; x_2[$ ou bien dans $]x_2; x_3[$. **Il est plus "rentable" de le prendre dans l'intervalle le plus grand**, soit dans notre exemple graphique ci-contre, l'intervalle $]x_2; x_3[$.

Par exemple, en prenant $x_4 \in]x_2; x_3[$, deux cas de figure se présentent:

- **cas a**: si $f(x_4) > f(x_2)$ alors le minimum est dans $]x_1; x_4[$
- **cas b**: si $f(x_4) < f(x_2)$ alors le minimum est dans $]x_2; x_3[$

Le choix de la "sonde" x_4 doit désormais être optimisé. Une idée est de la choisir telle que $]x_1; x_4[$ et $]x_2; x_3[$ aient la même longueur. On a donc:

$$x_4 - x_1 = x_3 - x_2 \quad \text{soit} \quad x_4 = x_1 + (x_3 - x_2)$$

si l'on désire toujours garder la même proportion entre la largeur du segment à l'étape i et à l'étape $i+1$, alors dans le cas a il faut assurer:

$$\frac{x_4 - x_2}{x_2 - x_1} = \frac{x_2 - x_1}{x_3 - x_2}$$

soit avec les notations de la figure:

$$\frac{c}{a} = \frac{a}{b} \quad \text{et dans le cas } b: \quad \frac{c}{b-c} = \frac{a}{b}$$

En éliminant c dans ces équations, on obtient: $\left(\frac{b}{a}\right)^2 = \frac{b}{a} + 1$

ce qui donne le nombre d'or $\varphi = \frac{b}{a} = \frac{1+\sqrt{5}}{2} = 1,618033....$

Si l'intervalle de départ $[x_1; x_3]$ alors la première sonde x_2 est donc prise en:

$$\varphi = \frac{x_3 - x_2}{x_2 - x_1} \Rightarrow \boxed{x_2 = x_1 + \frac{x_3 - x_1}{1+\varphi}}$$

Proposer une fonction récursive `fonct_mini_nbor(f,a,b,epsilon)` s'appuyant sur la méthode du nombre d'or et renvoyant la valeur du minimum de la fonction f à la précision *espilon* près.

Exercice n°11: 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 $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}$

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

Listing 4:

```
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(.....):
        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 $k = \{1 \dots 7\}$ limitant le calcul des matrices blocs C_{ij} à 7 multiplications:

$$\left[\begin{array}{l} M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22}) \times B_{11} \\ M_3 = A_{11} \times (B_{12} - B_{22}) \\ M_4 = A_{22} \times (B_{21} - B_{11}) \\ M_5 = (A_{11} + A_{12}) \times B_{22} \\ M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{array}\right.$$

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

$$\left[\begin{array}{l} C_{11} = M_1 + M_4 - M_5 + M_7 \\ C_{12} = M_3 + M_5 \\ C_{21} = M_2 + M_4 \\ C_{22} = M_1 - M_2 + M_3 + M_6 \end{array}\right.$$

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 5:

```
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.