TP3: Récursivité, tris, fractales

MP2I Option SI: Informatique tronc commun

1 Récursivité

Dans toute cette partie (jusqu'à la question 5), on d'interdit d'utiliser les boucles for et while. Pour écrire une fonction récursive, il faut avoir trouvé au préalable une relation de récurrence sur les objets que l'on manipule. Par exemple, si l'on veut écrire une fonction `puissance(a, b)` qui prend en entrée $a \in \mathbb{R}$ et $b \in \mathbb{N}$, et qui calcule a^b, on peut utiliser la formule :

$$a^b = \prod_{i=1}^{b} a$$

Cette formule peut directement se traduire en une boucle python :

```
def puissance(a, b):
  res = 1
  for i in range(b):
     res = res * a
  return res
```

Cette formule ne permet pas directement de trouver une fonction récursive. Cependant, on peut aussi remarquer qu'on a :

— $a^0 = 1$ pour tout $a \in \mathbb{R}$;

— $a^b = a^{b-1} \times a$ pour $a \in \mathbb{R}$ et $b \in \mathbb{N}^*$.

Ceci nous donne une version récursive, sans boucle, de la fonction puissance :

```
def puissance(a, b):
  if b == 0:
    return 1
  else:
    return puissance(a, b-1) * a
```

Q1. Écrire une fonction `reste(a, b)` qui calcule $a \bmod b$ pour $a \in \mathbb{N}, b \in \mathbb{N}^*$.

Q2. Écrire une fonction `coeff_binomial(k, n)` qui calcule $\binom{n}{k}$

Q3. Écrire une fonction `somme_chiffres(n)` qui calcule la somme des chiffres de n en base 10.

Q4. On considère la suite $(u_n)_{n\in\mathbb{N}}$ suivante, qui permet d'énumérer tous les éléments de $\mathbb{N}^2$:

$$\begin{array}{llll} u_0 = (0,0) & & & \\ u_1 = (1,0) & u_2 = (0,1) & & \\ u_3 = (2,0) & u_4 = (1,1) & u_5 = (0,2) & \\ u_6 = (3,0) & u_7 = (2,1) & u_8 = (1,2) & u_9 = (0,3) \\ \dots & & & \end{array}$$

Dessinez les premiers termes de cette suite dans le plan $\mathbb{N}^2$ sur papier comprendre la logique, puis codez une fonction `u(n)` permettant d'en calculer le n-ème terme (*Indication : trouvez une manière de déterminer* $u(n)$ *à partir de* $u(n-1)$ *lorsque* $n > 0$.

Q5. On note $P(n)$ l'ensemble des parties de $[\![0, n-1]\!]$. Écrivez une fonction récursive `parties(n)` qui renvoie $P(n)$ sous la forme d'une liste de listes. Par exemple, `parties(2) == [[], [0], [1], [0, 1]]` (à l'ordre près).

Dichotomie On rappelle le principe de la dichotomie : étant donné T un tableau trié dans l'ordre croissant de n éléments, et x un élément à chercher, la recherche par dichotomie consiste à :

1. Regarder la case $T[m]$ avec $m = \lfloor \frac{n-1}{2} \rfloor$ le milieu du tableau ;
   a) Si $T[m] = x$, alors la recherche termine ;
   b) Si $T[m] < x$, alors x n'est pas dans la première moitié de T ;
   c) Si $T[m] > x$, alors x n'est pas dans la deuxième moitié de T ;
2. Dans les deux derniers cas, on relance la recherche sur l'autre moitié de T, celle n'ayant pas été éliminée.

La dichotomie est une méthode très efficace : à chaque étape, on réduit la taille de la zone de recherche de moitié, si bien qu'au bout de $\lceil \log_2 n \rceil$ étapes au plus, l'algorithme se termine.

Q6. Écrire une fonction récursive `dichotomie(T, a, b, x)` qui cherche si l'élément x appartient à $T[a], \dots, T[b]$ en utilisant le principe de recherche par dichotomie. On supposera en précondition que T est trié par ordre croissant.

2 Tris

On se proposer d'étudier deux algorithmes de tri classique : le tri par sélection et le tri fusion.

Tri par sélection On rappelle le principe de ce tri : trouver le maximum du tableau et le mettre à la dernière place, puis trouver le maximum des éléments restants et le mettre à l'avant dernière place, et ainsi de suite.

Algorithme 1 : `tri_selection`(T)

Entrée(s) : T un tableau de taille n
Sortie(s) : T a été modifié, et est trié

1 **pour** $i = n-1$ **à** 0 **faire**
2     $j_0 \leftarrow$ indice du maximum de $T[0], \ldots, T[i]$;
3   Échanger $T[i]$ et $T[j]$;

Q7. Écrire une fonction `selection(T, i)` qui, étant donné T un tableau et i un indice valide de T, renvoie l'indice j_0 du maximum de $T[0], \ldots, T[i]$.

Q8. Proposer un invariant de boucle pour l'algorithme de la fonction précédente.

Q9. Implémenter le tri par sélection. Quelle est sa complexité ?

Tri fusion Le tri fusion est un algorithme de tri récursif. Contrairement au tri par sélection, il ne modifie pas le tableau en entrée, mais en crée une **copie triée**. Le fonctionnement est comme suit :

Algorithme 2 : `tri_fusion`(T)

Entrée(s) : T un tableau de taille n
Sortie(s) : Copie triée de T

1 **si** $n <= 1$ **alors**
2   **retourner** *une copie de T* // en Python, `T[:]` fait une copie du tableau `T`
3 $T_1, T_2 \leftarrow T[0 : \frac{n}{2}], T[\frac{n}{2} : n]$;
4 $T_1 \leftarrow$ `tri_fusion`(T_1);
5 $T_2 \leftarrow$ `tri_fusion`(T_2);
6 **retourner** *fusion*(T_1, T_2)

où `fusion` est une fonction permettant de fusionner deux listes triées en une seule.

Q10. Déterminer (sans tester sur ordinateur) laquelle de ces deux versions de la fonction `fusion` est correcte :

```python
def fusion(T1, T2):
    T = []
    n1 = len(T1)
    n2 = len(T2)
    # pour parcourir T1 et T2
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        # on comparer les deux premiers
        # éléments non-traités de T1 et T2
        if T1[i1] < T2[i2]:
            T.append(T1[i1])
            i1 += 1
        elif T1[i1] >= T2[i2]:
            T.append(T2[i2])
            i2 += 1
    # sortie: une des deux listes a été
    # complètement traitée
    return T + T1[i1:] + T2[i2:]
```

```python
def fusion(T1, T2):
    T = []
    n1 = len(T1)
    n2 = len(T2)
    # pour parcourir T1 et T2
    i1, i2 = 0, 0
    while i1 < n1 and i2 < n2:
        # on compare les deux premiers
        # éléments non-traités de T1 et T2
        if T1[i1] < T2[i2]:
            T.append(T1[i1])
            T.append(T2[i2])
        elif T1[i1] >= T2[i2]:
            T.append(T2[i2])
            T.append(T1[i1])
        i1 += 1
        i2 += 1
    # sortie: une des deux listes a été
    # complètement traitée
    return T + T1[i1:] + T2[i2:]
```

Q11. Écrire une fonction `tri_fusion(T)` implémentant le tri fusion.

La librairie `time` possède une fonction `time` renvoyant la date actuelle, exprimée en secondes passées depuis une date particulière[1]. Cette fonction permet notamment de calculer le temps d'exécution de code :

```
from time import time
debut = time()
(code à chronométrer)
fin = time()
duree = fin - debut
print("Le code a mis {duree} secondes à s'exécuter")
```

Q12. Écrire une fonction `compare_tests(n)` qui :

1. Crée 100 tableaux de taille n remplis de valeurs aléatoires entre 0 et 100 ;
2. Les trie avec un tri sélection ;
3. Chronomètre le temps total t_{select} qu'a pris l'opération ;
4. Fait la même chose avec un tri fusion, obtenant un temps t_{fusion} ;
5. Affiche le temps moyen que chaque algorithme a pris pour trier un tableau (i.e. $\frac{t_{\text{select}}}{n}$ et $\frac{t_{\text{fusion}}}{n}$).

On remarque que le tri fusion est bien plus efficace que le tri sélection. En terme de complexités asymptotique, le tri sélection est en $\mathcal{O}(n^2)$, là où le tri fusion est en $\mathcal{O}(n \log n)$.

1. Le premier janvier 1970 à 00h00m00s précisément !

3 Turtle

Le module `turtle` permet de tracer des dessins simples. On manipule un curseur, une petite tortue, qui laisse une trace en avançant. Elle peut tourner à gauche et à droite d'un angle donné et avancer d'un certain nombre de pixels. Par exemple, le code suivant dessine une étoile à 5 branches :

```
from turtle import *

TurtleScreen._RUNNING = True # démarrer la tortue

forward(200) # avancer de 200 pixels

right(144) # tourner de 144 degrés vers la droite
forward(200)

right(144)
forward(200)

right(144)
forward(200)

right(144)
forward(200)

done() # terminer l'exécution de la tortue
```

On peut le factoriser ainsi :

```
def avancer_et_tourner(l, alpha):
  """ Avance la tortue de l pixels, et la fait tourner de alpha degrés vers la droite"""
  forward(l)
  right(alpha)

def etoile(l):
  """Dessine une étoile à 5 branches, avec chaque segment de longueur l"""
  for i in range(5):
    avancer_et_tourner(l, 144)

TurtleScreen._RUNNING = True
etoile(200)
done()
```

On peut ajouter au début du code l'instruction `tracer(1, 0)` pour supprimer l'animation lorsque la tortue tourne, et donc accélérer le dessin.

Q13. Écrire une fonction `polygone(n)` qui dessine un polygone régulier à n côtés.

Utilisons turtle pour dessiner des fractales. Commençons par la *courbe de Koch*.
La courbe de Koch d'ordre de 0 est un segment, et la courbe de Koch d'ordre 1 est un segment coupé par les deux côtés d'un triangle équilatéral en son milieu :

Figure 1 – Courbe de Koch d'ordre 0

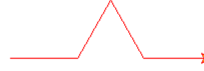


Figure 2 – Courbe de Koch d'ordre 1

Pour $n \in \mathbb{N}$, la courbe de Koch d'ordre $n+1$ est obtenue en prenant la courbe d'ordre 1, où chaque segment a été remplacé par la courbe de Koch d'ordre n.

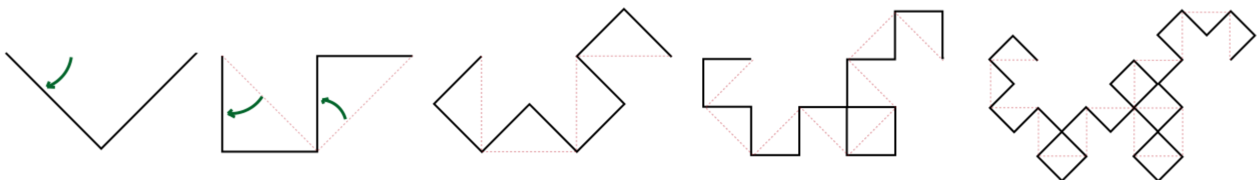
Q14. Dessiner sur papier la courbe de Koch d'ordre 2, puis celle d'ordre 3.

Q15. Écrire une fonction `courbe_koch(l, n)` qui dessine une courbe de Koch d'ordre n, avec l pixels entre le point d'arrivée et le point de départ, dans la direction actuelle de la tortue.

Le *flocon de Koch* est obtenu en remplaçant chaque côté d'un triangle équilatéral par une courbe de Koch.

Q16. Écrire une fonction `flocon(l, n)` qui dessine un flocon de Koch d'ordre n de longueur l.

On veut maintenant dessiner la *courbe du dragon*. Le principe est similaire à celui de la courbe de Koch. La courbe du dragon d'ordre 0 est un segment, et la courbe du dragon d'ordre $n+1$ est obtenue en remplaçant chaque segment de la courbe d'ordre n par un angle droit, en tournant alternativement à droite et à gauche :



Q17. Écrire une fonction `dragon(l, n, gauche_droite)` qui dessine une courbe du dragon d'ordre n, de longueur l. Si `gauche_droite` vaut `True`, la courbe tournera à gauche d'abord, et sinon elle tournera à droite d'abord.

La courbe du dragon possède de très nombreuses propriétés mathématiques intéressantes. Par exemple, on peut montrer qu'elle ne possède aucun croisement !
Il existe de nombreuses courbes fractales pouvant être construites de manières similaires. Par exemple, si vous prenez la courbe du dragon en la faisant toujours faire des angles droits vers la gauche (au lieu d'alterner droite/gauche), on obtient la courbe de Lévy !