



Récurtivité

Samedi 7 octobre 2016

Buts du TP

- Programmer effectivement des schémas de récursion vus en cours.
- Valider le fonctionnement, constater les dérapages dans des situations critiques.
- Se risquer à du bizarre (backtracking, programmation dynamique...)

Exercice 1. *Créer (au bon endroit) un dossier associé à ce TP. Y placer une copie du fichier récupéré dans le dossier partagé de travail de la classe : `squelette_cavalier.py`*
*Lancer Spyder/Pyzo/Idle, sauvegarder immédiatement au bon endroit le fichier `.py`; écrire une commande absurde, de type `print(5*3)` dans l'éditeur, sauvegarder et exécuter. Si vous êtes sous Spyder, changez (via F6) les options d'exécution, pour avoir à chaque exécution une nouvelle console et garder la main dessus. (il y a donc deux cases à vérifier/cocher).*

Les exercices marqués optionnels sont réservés à ceux qui vont vite.

1 Cas d'un paramètre entier

Tout d'abord, on reprend les classiques...

Exercice 2. Basics

Programmer (rapidement!) des fonctions récursives réalisant le calcul de $n!$, x^n en récursif basique puis avec l'algorithme d'exponentiation rapide.

```
>>> facto(5)
120
>>> expo_basique(42, 13)
1265437718438866624512L
>>> expo_rapide(42, 13)
1265437718438866624512L
```

Rappel : le L (en Python 2) signifie qu'il s'agit d'un entier « long »; pas d'état d'âme à avoir. En Python 3, le problème ne se posera pas.

Pour Fibonacci, on sait qu'un calcul récursif naïf induit un nombre d'appels récursifs de l'ordre de φ^n , avec $\varphi = \frac{1+\sqrt{5}}{2} \simeq 1,62$ le nombre d'or. Vérifions cela.

Exercice 3. Fibonacci au chronomètre

Écrire une fonction réalisant le calcul de f_n de façon récursive naïve.

```
>>> fibonacci(10)
55
```

Évaluer les temps de calcul de f_n pour $n \in \llbracket 30, 35 \rrbracket$ (suivant la machine, vous pourrez être amenés à changer cet intervalle).

```
from time import time
les_temps = []
for n in range(...):
    t0 = time()
    poubelle = fibonacci(n)
    t1 = time()
    les_temps.append(t1 - t0)
```

Évaluer ensuite les rapports successifs entre ces temps de calcul :

```
rapports = [les_temps[k+1]/les_temps[k] for k in range(len(les_temps) - 1)]
```

Exercice 4. Attention...

Écrire un programme récursif permettant de calculer la suite de premier terme $u_0 = 1$, et vérifiant la relation de récurrence $u_{n+1} = \frac{1}{2} \left(u_n + \frac{2}{u_n} \right)$.

Cette fonction devra retourner (presque instantanément) la valeur de u_{30} par exemple...

```
>>> [u(n) for n in range(10)]
[1.0, 1.5, 1.4166666666666665, 1.4142156862745097, ...]
```

Un exemple moins classique (que j'ai découvert un peu par hasard, pour occuper un TD!

Exercice 5. OPTIONNEL : nombres de Catalan

Les nombres de Catalan constituent la suite $(C_n)_{n \in \mathbb{N}}$, avec $C_0 = 1$, et :

$$\forall n \in \mathbb{N}, \quad C_{n+1} = \sum_{k=0}^n C_k C_{n-k}.$$

- Écrire une fonction récursive réalisant le calcul naïf de C_n (sans finasser, et en quatre lignes).
- Évaluer au chronomètre le temps de calcul de C_n , pour $n \in \llbracket 10, 12 \rrbracket$. Regarder l'évolution de ces temps.
- Prouver le résultat conjecturé!
- Comment calculer en temps raisonnable les C_n ?

Le dernier exercice de cette partie est important. Il faut y consacrer un peu de temps-cerveau.

Exercice 6. Décomposition en base 2

Écrire une fonction calculant la décomposition en base 2 d'un entier (sous la forme d'une liste de 0/1). Par exemple :

$$42 = 32+8+2 = 2^5+2^3+2^1 = \underline{101010}_2 \quad \text{et} \quad 1789 = 1024+512+128+64+32+16+8+4+1 = \underline{1101111101}_2$$

```
>>> decomposition(42)
[1, 0, 1, 0, 1, 0]
>>> decomposition(1789)
[1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1]
```

On commencera par donner une relation simple entre `decomposition(n)` et `decomposition(n//2)` (et ce sera alors presque fini).

2 Récursion sur des couples, listes, chaînes

Exercice 7. Algorithme d'Euclide

On rappelle que si $a \geq b \geq 0$ et $(a, b) \neq (0, 0)$, alors :

$$\text{pgcd}(a, b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b, r) & \text{sinon} \end{cases}$$

avec (dans le second cas), r le reste dans la division euclidienne de a par b .

On pourra noter que si $b > a$, cette relation reste vraie.

Exploiter cette relation pour écrire une fonction calculant le pgcd de deux entiers positifs.

```
>>> pgcd(1789, 945)
1
>>> pgcd(1789*42, 645*42)
42
```

On peut faire un peu mieux, en explicitant une relation de Bezout :

Exercice 8. Coefficients de Bezout ; OPTIONNEL

Si $\text{pgcd}(a, b) = p$, alors il existe $u, v \in \mathbb{Z}$ tels que $au + bv = p$.

Si $a = bq + r$, on sait que $\text{pgcd}(a, b) = \text{pgcd}(b, r)$, et on peut facilement retrouver une relation de Bezout pour (a, b) à l'aide d'une relation de Bezout pour (b, r) .

— Expliciter le lien entre ces relations de Bezout, ainsi que les cas terminaux.

— Programmer effectivement une fonction calculant une relation de Bezout entre deux entiers.

```
>>> bezout(945, 1789)
(248, -131)
>>> 945*248 - 1789*131
1
```

Pour recomposer un entier à partir de sa représentation en base b , la récursion s'impose encore : pour $[1, 7, 8, 9]$, on recompose récursivement $[1, 7, 8]$. On multiplie le résultat 178 par 10 et on ajoute la décimale de poids faible 9.

Exercice 9. Recomposition d'un entier à partir de sa décomposition en base 2

Utiliser le principe expliqué avant cet exercice pour écrire une fonction calculant l'entier dont la représentation en base 2 est donnée.

```
>>> recomposition([1, 1, 0, 1])
13
>>> recomposition(decomposition(1789))
1789
>>> recomposition(decomposition(945))
945
```

Pour chercher x dans une liste triée de n éléments, ce n'est pas trop compliqué si $n = 0$ ou $n = 1$. Si $n \geq 1$, on peut aller voir au milieu (position $n//2$ en Python). Si on trouve x , c'est gagné. Si la valeur présente en cette position est strictement plus petite que x , alors on va chercher x à droite; sinon, à gauche : vive la récursivité (et le slicing¹)

Exercice 10. Recherche dichotomique dans une liste

Appliquer le principe précédemment décrit pour tester l'appartenance d'un entier/flottant à une liste triée par ordre croissant.

Quelques tests variés :

```
>>> appartient(5, list(range(6))) # à droite
True
>>> appartient(5, list(range(60))) # à gauche
True
>>> appartient(500, list(range(60))) # trop à droite
False
>>> appartient(51, list(range(0, 60, 200))) # pas au milieu
False
```

Dans l'exercice précédent, on réalise un slicing (découpe/recopie d'un morceau de liste) avant l'appel récursif, ce qui induit des coûts fâcheux : la complexité reste linéaire (alors que le but de la recherche dichotomique est de passer au dichotomique...).

Pour éviter cela, on peut écrire une fonction (récursive) travaillant sur des indices délimitant des morceaux de listes à traiter, plutôt que de réaliser des slicings.

Exercice 11. OPTIONNEL : passer des indices plutôt que slicer.

Réécrire de façon récursive, mais sans slicing, la fonction 10

1. Nous reviendrons sur ce point plus tard.

```
def recherche_dicho_entre(x, t, i, j):
    ...
def recherche_dicho_logarithmique(x, t):
    recherche_dicho_entre(x, t, 0, len(t)-1)
```

Exercice 12. Miroir d'une chaîne

Écrire de façon récursive une fonction calculant le miroir d'une chaîne (les mêmes lettres, à l'envers).

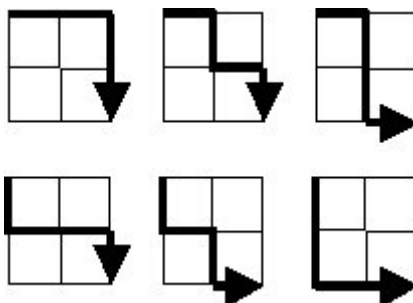
```
>>> miroir('PLOUF'), miroir('BARBARA')
('FUOLP', 'ARABRAB')
```

On rappelle qu'on peut concaténer des chaînes de caractères comme des listes via l'opérateur +

3 Programmation dynamique

Voici l'énoncé du quinzième problème de ProjectEuler :

Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.



How many such routes are there through a 20×20 grid?

Si on note $\varphi(i, j)$ le nombre de chemins permettant d'atteindre (i, j) depuis $(0, 0)$, alors on a :

$$\forall i, j \in \mathbb{N}, \quad \varphi(i, j) = \begin{cases} 1 & \text{si } i = 0 \text{ ou } j = 0 \\ \varphi(i-1, j) + \varphi(i, j-1) & \text{sinon} \end{cases}$$

Exercice 13. Récursion naïve

Programmer la fonction φ sur ce principe. Vérifier que $\varphi(15, 10) = \frac{25!}{15!10!} = 184756$, mais que le calcul ne termine pas en des temps raisonnables pour $\varphi(20, 20)$.

Pour améliorer les choses, on peut mémoïzer : on travaille avec un tableau de valeurs, où on place 0 pour toutes les valeurs non calculées. À chaque appel à φ , on commence par chercher dans le tableau des valeurs calculées :

- s'il y a quelque chose de strictement positif, on le renvoie ;
- sinon, on calcule récursivement $\varphi(i-1, j) + \varphi(i, j-1)$, on le place dans le tableau de valeurs, et on renvoie le résultat.

Exercice 14. Programmation dynamique avec memoization

Mettre en œuvre ce principe, pour calculer effectivement $\varphi(20, 20)$.

On pourra réaliser l'initialisation du tableau via :

```
valeurs_chemins = [[1]*21]+[[1]+[0]*20 for _ in range(20)]
```

On va enfin s'intéresser au calcul de $p(n)$, défini comme le nombre de façons de décomposer n en sommes. Il en est question dans le problème 76 de Project Euler

It is possible to write five as a sum in exactly six different ways :

$$5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 + 1 + 1 + 1$$

How many different ways can one hundred be written as a sum of at least two positive integers ?

On demande ici $p(100) - 1$, la décomposition triviale n'étant pas comptée.

Si on note $\psi(n, m)$ le nombre de façons de décomposer n en somme, chaque terme de la somme étant majoré par m , alors on a pour tout $n, m \in \mathbb{N}^*$ tels que $n > m > 0$:

$$\psi(n, m) = \psi(n - m, m) + \psi(n, m - 1).$$

Exercice 15. Sur les traces de Mac Mahon²

- Traiter les «cas limites» ($m \leq 0$, $m \geq n$).
- En utilisant ce principe, écrire un programme récursif mémoïzé pour calculer des valeurs de ψ .
- Donner les valeurs de $p(100)$, $p(200)$ et $p(721)$.

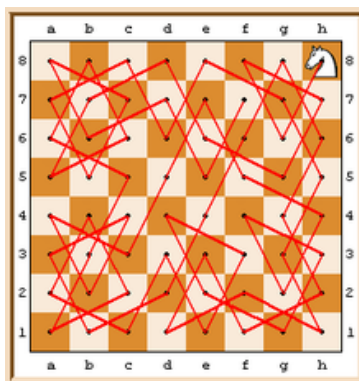
4 Problème du cavalier par backtracking

Sur un échiquier (*aka* : $[[1, 8]]^2$), un cavalier part de la position (1, 1) et veut parcourir toutes les cases exactement une fois (rappel : un cavalier se déplace en $L\dots$). La figure à la fin de l'énoncé (wikipedia) montre un tel parcours³.

Imaginons que le cavalier a parcouru 10 cases (on sait lesquelles). On peut regarder les voisins (encore accessibles) du cavalier. Pour chacun de ces voisins, on peut tenter le coup : on marque ce voisin, et on continue récursivement depuis cette configuration. On arrive parfois à un blocage, et parfois (on l'espère) on obtient une solution. En cas d'échec sur TOUTES les cases qui étaient envisageables, cela signifie qu'on est dans une situation perdante : on annonce un échec. À qui ? Mais à la configuration précédente, à 9 cases, qui avait tenté cette 10-ième case (bref, on revient en arrière en effaçant ses traces ; c'est le *backtracking*)

On donne sur le réseau un squelette de code permettant de résoudre le problème.

- Récupérer le fichier `squelette_cavalier.py`.
- Expliquer le rôle des différentes variables définies dans ce fichier.
- Compléter le programme `backtrack`
- Vérifier qu'il n'existe pas de parcours pour un échiquier (4, 4), mais qu'il en existe pour des échiquiers (6, 6) et (8, 8).
- Comment modifier le programme pour imposer un retour en position (1, 1) ?



Le lecteur intéressé pourra récupérer le sujet de F. Ridde sur le problème des reines, traité par backtracking...

2. Le major anglais (pas le maréchal français), qui a calculé à la main les $p(k)$ pour $k \leq 200$.

3. C'est même un cycle : à la 64ème case, on peut revenir sur la première