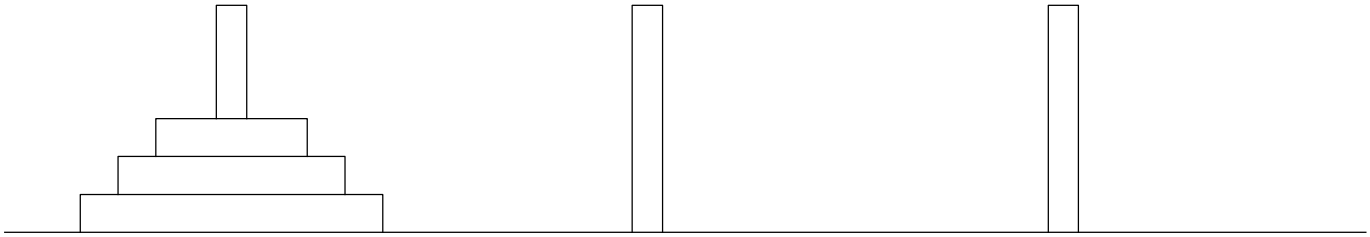
TP 6 - Récursivité

1 Une motivation : problème des tours de Hanoi

Vous connaissez sans doute ce puzzle dit des tours de Hanoi, où des disques sont initialement placés du plus grand en bas au plus petit en haut autour d'une première « tour » et qu'on cherche à déplacer de la première tour à la troisième en respectant quelques règles simples :

- On ne déplace qu'un disque à la fois, d'une tour vers une autre
- On ne peut placer un disque au-dessus d'un autre que s'il lui est de diamètre inférieur

Voici, pour trois anneaux, quelle pourrait être la configuration initiale :



Le problème est trivial avec un anneau, facile encore avec deux voire trois anneaux. Au-delà, c'est un peu plus pénible, mais au moins peut-on vite formaliser une démarche pour résoudre le problème pour n anneaux : en nommant A, B, C les trois tours, pour déplacer n disques de la tour A à la tour C, on voit qu'on ne peut procéder qu'ainsi :

- déplacer $n-1$ disques de la tour A à la tour B (en se servant bien sûr au passage de la tour C pour y déposer des disques au fur et à mesure des mouvements)
- déplacer le n-ième (et le plus large) disque de la tour A à la tour C
- déplacer $n-1$ disques de la tour B à la tour C (avec la tour A qui sert de tour « intermédiaire »)

Le faire à la main pour un nombre un peu plus important de disques reste un peu délicat, mais la démarche décrite ci-dessus se prête parfaitement à la programmation :

Supposons qu'on dispose d'une fonction d'entête `déplace(tour1, tour2)` et dont le rôle est de retirer le disque situé en haut de `tour1` pour le placer en haut de `tour2`, alors voici comment on pourrait écrire une fonction d'entête `hanoi(n, A, B, C)` dont le rôle va être de déplacer n disques situés en haut de la tour A vers la tour C en utilisant la tour intermédiaire B :

```
def hanoi(n, A, B, C):
    if n == 1: # un seul disque
        déplace(A, C)
    else:
        hanoi(n-1, A, C, B) # déplace n-1 plus petits disques de A vers B (avec C intermédiaire)
        déplace(A, C) # déplace le disque le plus grand, désormais en haut de A, vers C
        hanoi(n-1, B, A, C) # déplace les n-1 plus petits disques de B vers C (avec A intermédaire)
```

La fonction `hanoi` est une fonction que l'on dit **récursive**, car elle a la propriété de s'invoquer elle-même. Le principe d'une fonction récursive sera toujours le suivant :

- Si le problème à résoudre est si simple que sa solution est immédiate, on s'arrête (avec ou non une valeur de retour selon le traitement à effectuer bien sûr)

  Ne jamais oublier cette partie, qu'on nomme « le test d'arrêt »

- Sinon, on découpe le problème en un ou plusieurs problèmes plus simples mais de même nature et dont la solution, complétée de quelques opérations, donne la solution au problème initial. (C'est le paradigme « diviser pour régner » qui souvent est en œuvre ici)

  Pour résoudre ces problèmes plus simples mais de même nature notre fonction peut s'invoquer elle-même avec de nouveaux arguments, plus simples (jamais les mêmes bien sûr !)

2 Un exemple commenté

(Les plus rapides d'entre vous ont peut-être déjà abordé ceci au cours du premier TP, mais je pense qu'ils sont rares…)

Vous connaissez sans doute l'algorithme d'Euclide permettant de calculer le pgcd de deux entiers. On en rappelle le principe : si a et b sont deux naturels, et que $b \neq 0$, et si de plus la division euclidienne de a par b s'écrit $a = bq + r$, alors on observe qu'un diviseur commun de a et de b est aussi un diviseur de $r = a - bq$, et donc un diviseur commun de b et de r.

De même, un diviseur commun de b et de r est aussi un diviseur commun de $a (= bq + r)$ et de b.

De ce fait, a et b d'une part, b et r d'autre part ont les mêmes diviseurs communs, mais ils ont alors aussi le même pgcd : $a \wedge b = b \wedge r$.

On peut alors continuer, jusqu'à tomber sur un reste nul, car si $r = 0$, d'une part on ne saurait diviser par r, et d'autre part le pgcd de b et de r est bien connu : il vaut b. (Tout entier divise 0, donc les diviseurs communs à b et r sont tous les diviseurs de b, et le plus grand d'entre-eux, c'est b)

En résumé : voici l'algorithme d'Euclide pour calculer $a \wedge b$:

- Si $b = 0$, alors $a \wedge b = b$
- Sinon, alors en notant r le reste de la division euclidienne de a par b, $a \wedge b = b \wedge r$ (et on continue !)

La raison objective du succès de l'algorithme précédent est qu'à chaque nouveau pgcd à calculer, le second entier (d'abord b, puis r, puis…) décroit strictement.

Voici une version dite itérative (sans récursivité) d'une fonction calculant le pgcd de deux entiers par l'algorithme d'Euclide :

```
def pgcd(a, b): # a et b sont des naturels
    while b > 0:
        r = a % b
        a, b = b, r
    return a
```

Exemple en exécutant `pgcd(27, 72)` : le premier reste calculé vaut 27 (car $27 = 72 \times 0 + 27$) et à la fin de la première itération, les valeurs de `a` et `b` deviennent 72 et 27 (elles ont donc été échangées)

Le second reste calculé vaut 18 et à la fin de la seconde itération : `a` et `b` valent respectivement 27 et 18.

Le troisième reste vaut 9 et à la fin de la troisième itération : `a` et `b` valent 18 et 9.

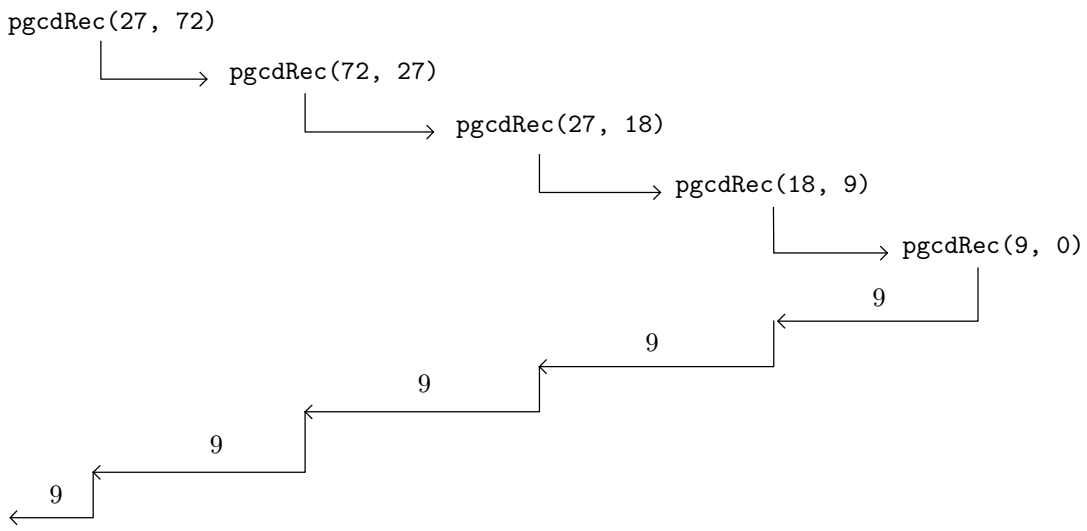
Le quatrième reste vaut 0 et à la fin de la quatrième itération : `a` et `b` valent 9 et 0.

La boucle `while` s'interrompt alors, et la valeur 9 est renvoyée par notre fonction.

L'algorithme d'Euclide qui transforme le problème initial en un problème plus simple est par nature particulièrement adapté à une implémentation récursive :

```
def pgcdRec(a, b):
    if b == 0:
        return a
    return pgcdRec(b, a % b)
```

laquelle au fond ne fait guère que paraphraser l'algorithme d'Euclide présenté un peu plus haut. Que se passe-t-il en exécutant `pgcdRec(27, 72)` ? Le schéma ci-dessous permet de le visualiser :



Remarque. On voit donc ici que les deux fonctions précédentes réalisent le même travail, en implémentant de deux manières différentes un même algorithme. Ce qui change est que dans la version récursive, les appels imbriqués de fonctions remplacent la boucle `while`. On le devine, il y a un léger surcoût à l'implémentation récursive de cet algorithme, car au fur et à mesure des appels imbriqués, on doit garder trace des variables locales qui auraient pu être définies dans l'une ou l'autre de ces fonctions, de l'instruction où l'exécution de la fonction appelante devra reprendre, lorsque la fonction appelée aura terminé son exécution et renvoyé une valeur. On négligera ce surcoût.

3 Calcul de suites définies par une relation de récurrence

Sans surprise, la récursivité permet de coder facilement le calcul du terme d'indice n d'une suite donnée par une relation de récurrence.

Question 1. On définit pour tout $n \in \mathbb{N}$, la factorielle de n et on note $n!$ l'entier défini par récurrence par :

$$0! = 1 \text{ et } \forall n \in \mathbb{N}, (n+1)! = (n+1) \times n!$$

En déduire une fonction d'entête `def factorielle(n):`, récursive, qui admet pour unique argument un naturel n et qui renvoie $n!$.

Question 2. Qu'advient-il si on exécute la fonction précédente avec pour argument -1, ou 0.5 ?

On le devine, une fonction ne saurait s'appeler elle-même à l'infini, à la fois parce qu'on ne peut laisser un temps infini à ce que notre algorithme s'exécute, mais aussi car les appels imbriqués ont aussi un coût en mémoire et que pas plus que le temps, la mémoire n'est infinie ! Pour éviter des plantages plus sérieux, python définit un nombre maximal d'appels imbriqués de fonctions et déclenche une erreur (que vous avez dû voir apparaître) dès que ce nombre est atteint.

Si vous voyez apparaître cette erreur, cela signifie que, ou bien :

- votre fonction est mal écrite et il arrive que le test d'arrêt ne soit jamais réalisé
- votre fonction est correcte mais que l'appel qui en est fait n'en respecte pas les spécifications
- le choix d'une fonction récursive n'est pas adapté au problème posé

Question 3. La suite de Fibonacci est la suite (F_n) définie par $F_0 = 0$, $F_1 = 1$ et $\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$.

Ecrire une fonction d'entête `def fibonacci(n):`, récursive, qui admet pour unique argument un naturel n et qui renvoie F_n.

Question 4. Caculer F_{30}, F_{35}, F_{40} avec la fonction précédente.

Que constate-t-on ? Quelle explication peut-on donner à cette grande lenteur ?

4 Exponentiation rapide

Soit x un nombre (entier, réel ou complexe, peu importe) et $n \in \mathbb{N}$, on cherche à calculer, aussi rapidement que possible x^n. La première idée consiste à calculer $x^2 = x \times x$, puis $x^3 = x^2 \times x$, jusqu'à $x^n = x^{n-1} \times x$ ce qui conduit à effectuer $n-1$ multiplications. On va voir que c'est très loin d'être optimal. Déjà $x^4 = x^2 \times x^2$ s'obtient donc en deux multiplications, x^8 en trois multiplications, x^{16} en 4 multiplications, x^{32} en 5 multiplications… $x^{40} = x^{32} \times x^8$ peut à son tour s'obtenir en 6 multiplications.

L'algorithme d'exponentiation rapide repose sur la discussion suivante :

- Si n est pair, $x^n = (x^{n/2})^2$
- Si n est impair, $x^n = x\,(x^{(n-1)/2})^2$

Question 5. En vous aidant de la discussion ci-dessus, écrire une fonction d'entête `def expoRapide(x, n):` récursive qui admet en entrée deux arguments `x` (un nombre) et `n` (un entier naturel) et qui renvoie la valeur de x^n.

Indication : on n'oubliera pas le test d'arrêt !!!

Bien sûr, on teste. Il pourra être utile de rajouter au début de la fonction un `print(n)` pour voir combien de fois et avec quels arguments la fonction est invoquée. Testez par exemple avec `expoRapide(2, 1000)`.

5 Sous-listes d'une liste donnée

On dit de deux listes L et K que K est une sous-liste de L si et seulement si on obtient K à partir de L en retirant un certain nombres d'éléments de L (pas forcément à la fin). Par exemple `[2, 1, 3]` est une sous-liste de `[0, 2, 5, 1, 4, 3]` mais `[0, 1, 2]` n'en est pas, en revanche, une.

Comment déterminer alors la liste des sous-listes d'une liste donnée `L` ? On supposera dans ce qui suit que `L` désigne une liste d'éléments deux à deux distincts.

- Ce n'est pas bien difficile si `L` est vide…
- Sinon, en notant `x` le denier terme de `L`, et `K` la liste obtenue à partir de `L` en supprimant son dernier élément, et si on connaît toutes les sous-listes de `K`, il suffit pour chacune d'entre-elles de la conserver telle quelle ou d'y ajouter `x` pour connaître les sous-listes de `L` (de ce raisonnement, on déduit d'ailleurs qu'il y a exactement 2 fois plus de sous-listes de L qu'il y a de sous-listes de `K`)

  Ainsi, par exemple, ayant noté que les sous-listes de `[1,2]` sont `[], [1], [2]` et `[1,2]`, alors les sous-listes de `[1, 2, 4]` sont `[],[4], [1], [1,4], [2], [2,4], [1,2], [1,2,4]`.

Question 6. Ecrire une fonction d'entête `def sousListes(L):`, récursive, qui admet pour unique argument une liste `L` et qui renvoie la liste des sous-listes de `L`.

Question 7. En s'inspirant de la question précédente, écrire une fonction d'entête `def sousListes2(L, k):`, toujours récursive, qui admet deux arguments : une liste `L` et un entier naturel `k` et qui renvoie l'ensemble des sous-listes de `L` ayant pour longueur `k`.