RÉCURSIVITÉ

1 Le principe

Regardez attentivement le programme suivant.

```
def depart(n):
    if n == 0:
        print("partez !")
    else:
        print(n)
        depart(n-1)
```

La fonction `depart` s'appelle elle-même ! Et oui. Si dans la console, on entre `depart(3)`, alors :

- `depart(3)` va afficher 3 puis appeler `depart(2)`.
- `depart(2)` va afficher 2 puis appeler `depart(1)`.
- `depart(1)` va afficher 1 puis appeler `depart(0)`.
- `depart(0)` affiche `partez !` et ne fait rien d'autre : fin de la procédure.

Définition 6.1

Une fonction est dite *récursive* si elle s'appelle elle-même au cours de son exécution.

Exercice 1. Que se passe-t-il si on appelle `depart(-1)` ? Et `depart(3.5)` ? Modifier la fonction pour éviter ces problèmes, et tester.

Il est primordial que la fonction se termine toujours et ne s'appelle pas à l'infini. Il faut donc une *condition d'arrêt* : lorsqu'elle est vérifiée, on arrête de s'appeler soi-même. Une fonction récursive commencera donc généralement par un `if <condition> :` suivi d'une commande `return`.

2 Itératif vs récursif

Quand une opération doit être répétée plusieurs fois, on peut le faire de manière *itérative* (avec une boucle `for` ou `while`) ou récursive.

Exemple. Voici un exemple classique pour calculer le pgcd de deux entiers naturels :

```
def pgcd_iter(a,b):      # version itérative
    if b == 0:
        return a
    while b != 0:
        a,b = b,a%b
    return a

def pgcd_recu(a,b):      # version récursive
    if b == 0:
        return a
    return pgcd_recu(b,a%b)
```

Exemple. Autre classique : le calcul d'une factorielle. Voici une version itérative :

```
def facto_iter(n):      # version itérative
    p = 1
    for k in range(n):
        p = p*(k+1)
    return p
```

Exercice 2. Écrire une version récursive de la fonction ci-dessus.

3 Récursivité et complexité

Le coût d'un algorithme récursif est lié au nombre d'appels récursifs. On peut l'obtenir par une relation de récurrence. Par exemple, pour la fonction `facto_recu`, si on note c_n le coût pour l'appel `facto_recu(n)`, on a :

- $c_0 = 1$ car on fait juste une comparaison.
- $c_n = 2 + c_{n-1}$ car on fait une comparaison, une multiplication, et l'appel de `facto_recu(n-1)`.

On montre alors facilement que $c_n = 2n+1$, si bien que la complexité est d'ordre n, donc linéaire. C'est aussi le cas de `facto_iter`.

Le piège de la récursivité. Voici un autre grand classique : la suite de Fibonacci.

```
def fibo_recu(n):
    if n==0 or n==1:
        return 1
    else:
        return fibo_recu(n-1) + fibo_recu(n-2)
```

Exercice 3. Taper et tester avec $n = 4$, $n = 25$ puis $n = 35$. Que remarquez-vous ?

Question 1. Déterminer la relation de récurrence de c_n pour la fonction `fibo_recu`.

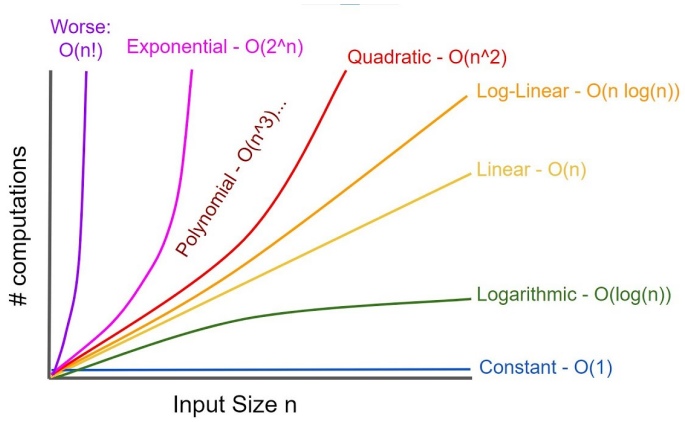
De cette relation de récurrence, on peut montrer que $c_n \geq 2c_{n-2}$, et en déduire que $c_n \geq (\sqrt{2})^n$. On peut de même montrer que $c_n \leq 2^n$. La complexité de `fibo_recu` est ainsi exponentielle, cf la définition ci-dessous.

Définition 6.2

Si le coût vérifie

$$a^n \leq c_n \leq b^n$$

avec $1 < a \leq b$, on dit que le coût est *exponentiel*.



Un coût exponentiel est encore pire qu'un coût quadratique. Il faut l'éviter à n'importe quel prix. Une version itérative de `fibo_recu` sera beaucoup plus rapide :

```
def fibo_iter(n):
    u,v = 1,1
    for k in range(2,n+1):    # si n vaut 0 ou 1, range(2,n+1) est vide
        u,v = u+v,u           # et on ne rentre pas dans la boucle !
    return u
```

Cette version a une complexité linéaire : on peut sans problème calculer `fibo_iter(10000)`.

En général, si à l'étape `n` on appelle plusieurs fois les étapes `n-1` ou `n-2`, cela conduit à une complexité exponentielle.

4 Exponentiation rapide

On cherche à calculer x^n, avec $n \in \mathbb{N}^*$ et x un nombre flottant non nul.

Méthode naïve. On calcule successivement x^2, x^3, etc. en se basant sur la définition mathématique $x^n = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$:

```
def expo_naif(x,n):
    p = 1
    for blabla in range(n):    # on n'utilise pas la variable blabla
        p = p*x
    return p
```

Question 2. Quelle est la complexité de l'algorithme ci-dessus ?

Méthode rapide. On veut calculer x^{37}. Avec `expo_naif`, il faut 36 multiplications. Voici une méthode plus rapide :

$$\begin{aligned}
x^{37} &= x \cdot x^{18} \cdot x^{18} && \text{(2 mult., reste à calculer } x^{18}\text{)}\\
x^{18} &= x^9 \cdot x^9 && \text{(1 mult., reste à calculer } x^9\text{)}\\
x^9 &= x \cdot x^4 \cdot x^4 && \text{(2 mult., reste à calculer } x^4\text{)}\\
x^4 &= x^2 \cdot x^2 && \text{(1 mult., reste à calculer } x^2\text{)}\\
x^2 &= x \cdot x && \text{(1 mult.)}
\end{aligned}$$

Ce qui fait seulement 7 multiplications ! Il s'agit d'une méthode par dichotomie : à chaque étape la puissance de x à calculer est (à peu près) divisée par 2.
On rappelle que $n//2$ désigne le quotient de la division euclidienne de n par 2. La méthode rapide repose sur le fait que

$$x^n = \begin{cases} (x^2)^{n//2} & \text{si } n \text{ est pair,} \\ x \cdot (x^2)^{n//2} & \text{si } n \text{ est impair.} \end{cases}$$

On passe donc du calcul de x^n à celui de $(x^2)^{n//2}$. Puis, pour calculer $(x^2)^{n//2}$, on réutilise la formule ci-dessus en remplaçant x par x^2 ainsi que n par $n//2$. À chaque étape, la puissance en exposant est remplacée par son quotient entier par 2. On continue jusqu'à ce que la puissance devienne 0, et dans ce cas le calcul est immédiat. L'algorithme (récursif) correspondant est le suivant :

```
def expo_rapide(x,n):
    if n == 0:           # condition d'arrêt
        return 1
    if n%2 == 0:
        return expo_rapide(x*x,n//2)
    else:
        return x*expo_rapide(x*x,n//2)
```

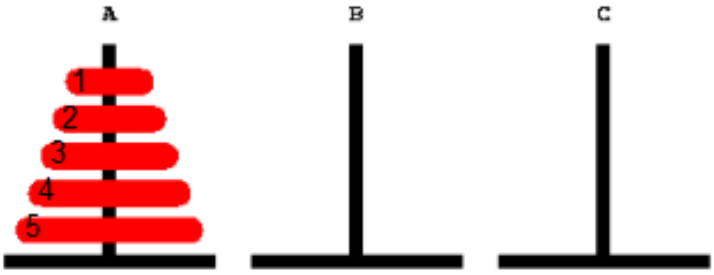
Question 3. Montrer que $c_n \leq 6 + c_{n//2}$. En déduire une majoration de c_n lorsque $n = 2^p$ avec $p \in \mathbb{N}$.

Bien que la question ci-dessus ne le montre pas entièrement, on voit qu'on a affaire à une complexité logarithmique, donc bien inférieure à la complexité linéaire de la méthode naïve.

5 Exercices d'approfondissement

Exercice 4. Écrire une fonction récursive qui calcule le n-ième terme de la suite $\begin{cases} u_{n+1} = \frac{1}{2}\left(u_n + \frac{4}{u_n}\right) \\ u_0 = 1 \end{cases}$ Vérifier que $u_n \to 2$.
S'assurer que le coût est linéaire ! *Indication : à l'étape n, on pourra stocker le résultat de l'étape n − 1 dans une variable.*

Exercice 5 (Tours de Hanoi)**.** Le problème des tours de Hanoi est un jeu de réflexion imaginé par le mathématicien français Edouard Lucas. Il y a trois tours nomées A, B, C. Au départ, des disques troués de taille croissante sont empilés dans la tour A.



Le but est de déplacer ces disques de la tour A à la tour C en respectant deux règles :

- On ne peut déplacer qu'un disque à la fois.
- On ne peut placer un disque que sur un disque plus grand ou sur une tour vide.

Le problème se résout de manière récursive. On sait résoudre le problème s'il n'y a qu'un disque : il suffit de le déplacer directement à la tour C. Maintenant, si on a n disques et qu'on sait résoudre le problème pour $n-1$ disques, alors :

1. **Étape 1 :** On déplace les $n-1$ premiers disques de la tour A à la tour B.
2. **Étape 2 :** On déplace le grand disque (numéro n) de la tour A à la tour C.
3. **Étape 3 :** On déplace les $n-1$ disques de la tour B à la tour C.

Pour traiter le problème, nous allons utiliser des listes pour représenter les tours. Avec 3 disques :

```
A = ['A',3,2,1]  # Le disque 3 est tout en bas, le disque 1 est en haut
B = ['B']        # aucun disque au début
C = ['C']        # idem
```

On veut construire une fonction récursive `hanoi(n,A,B,C)` qui réalise les déplacements de `n` disques de la tour A à la tour C. On utilise l'algorithme suivant, où l'étape 2 est déjà codée. Compléter les arguments des fonctions `hanoi` correspondant aux étapes 1 et 3, et tester avec les listes `A,B,C` ci-dessus et l'instruction `hanoi(len(A),A,B,C)`.

```
def hanoi(n,A,B,C):                  # déplace n disques de la tour A à la tour C
    if n>0:
        hanoi(...)                   # étape 1, à compléter
        if len(A)>1:
            print(A[-1],':',A[0],'-->',C[0])
            C.append(A.pop())

        hanoi(...)                   # étape 3, à compléter
```

Note : `A.pop()` retourne le dernier élément de la liste `A`, càd `A[-1]`, et en plus l'enlève de la liste `A`.