TP 5 : Récursivité

I. Définition générale de la récursivité

I.1. Définition

Un objet est dit **récursif** lorsque sa définition fait appel à cet objet lui-même. Ce schéma qui peut paraître surprenant est en réalité assez classique en mathématiques.

- C'est par exemple le cas des suites récurrentes : la valeur de la suite à un certain rang est défini par la valeur de la suite aux rangs précédents. On peut considérer par exemple les suites (u_n) et (v_n) définies par :

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N},\ u_{n+1} = 3 \times u_n + 1 \end{cases} \qquad \begin{cases} v_0 = 1 \\ v_1 = 1 \\ \forall n \in \mathbb{N},\ v_{n+2} = v_{n+1} + v_n \end{cases}$$

  Dans le premier exemple, la valeur de la suite $u = (u_n)$ en un point est définie par la valeur de la suite u en un autre point. Il semble donc que pour connaître la suite u, il faut ... connaître la suite u ! Définir un objet à l'aide de la définition de cet objet semble être une manière de procéder assez peu pertinente et qui a peu de chance d'aboutir. En réalité, les suites u et v présentées ci-dessus sont parfaitement bien définies (on précisera pourquoi dans le paragraphe suivant).
- Comme autre exemple classique, on peut citer la suite $(n!)_{n\in\mathbb{N}}$ pour laquelle on donne généralement la définition suivante :

$$\boxed{\begin{array}{rcl} 0! & = & 1 \\ \forall n \in \mathbb{N},\ n! & = & n \times (n-1)! \end{array}}$$

I.2. Calculabilité d'une fonction récursive

I.2.a) Une première implémentation de la fonction factorielle

- Pour que les objets définis de manière récursive aient un intérêt, il faut qu'on puisse les calculer. Par exemple, la définition récursive des suites u, v et $(n!)$ n'a de sens que si on peut calculer tous les termes de ces suites. Pour mieux comprendre la difficulté qui découle d'une définition d'un objet à l'aide de cet objet lui-même, on propose l'implémentation suivante de la fonction factorielle.

```python
def factrec(n) :
    # La fonction fact prend en argument
    # un entier n et renvoie n!
    return n * factrec(n-1)
```

- ▶ Recopier ces lignes de codes sur votre ordinateur.
- ▶ On souhaite effectuer l'appel `factrec(3)` à l'aide de cette fonction. Détailler sur votre feuille les différentes étapes qui seront réalisées par la fonction `factrec` pour effectuer cet appel.

► Effectuer l'appel `factrec(3)`. Recopier le message d'erreur qui apparaît.

► Expliquer le problème survenu.

- Afin d'assurer la calculabilité, on est amené à préciser la définition récursive d'un objet.
  1) L'appel réalisé dans la définition de l'objet doit l'être sur une partie strictement plus petite de l'objet. La « taille » de l'objet sur lequel on effectue les appels diminue donc strictement au cours des appels successifs.
  (dans le cas de factorielle, le calcul de `factrec(n)` *sera effectué à l'aide de celui de* `factrec(n-1)` *et il y a bien une stricte décroissance de* 1 *de l'entier sur lequel on effectue l'appel)*
  2) La définition doit donner la valeur de l'objet de taille minimale. On parle alors du cas initial.
  (dans le cas de factorielle, il faut préciser que `factrec(0)` *vaut* 1*)*

  La taille de l'objet est souvent une valeur entière positive. La stricte décroissance de la taille de l'objet lors des appels assure que le cas initial sera forcément atteint ce qui met fin aux appels. En effet, il n'existe pas de suite strictement décroissante d'entiers positifs. Cela assure que les appels sucessifs nous mèneront forcément à un cas de base, ce qui permettra d'assurer la **terminaison** de la fonction.

I.2.b) Une implémentation correcte

► Calculer 5! à l'aide de la définition récursive de factorielle.

► Ajouter au code précédent les lignes nécessaires à la bonne implémentation de la fonction `factrec` et effectuer l'appel `factrec(5)`. On vérifiera le résultat précédent.

► Réaliser l'appel `factrec(-5)`. Noter le message d'erreur obtenu. Expliquer.

- Ajouter en première ligne de la fonction `assert(n >= 0 and type(n) == int)`. Effectuer de nouveau l'appel `factrec(-5)`. Qu'obtient-on ? Quel est l'avantage ?

- Effectuer l'appel `factrec(1500)` puis l'appel `factrec(10250)`. Recopier le message d'erreur et tenter d'apporter une explication.

Taille de la pile d'appels

- En **Python**, le module `sys` fournit un accès à certaines variables utilisées et maintenues par l'interpréteur, et à des fonctions interagissant fortement avec ce dernier.
- En particulier, ce module peut être utilisé pour connaître la taille maximale de la pile d'appels et pour modifier cette taille.
- Plus précisément :
  - × on peut obtenir la taille de la pile d'appels par défaut par la commande suivante :

    ```
    print(sys.getrecursionlimit())
    ```

    (ne pas oublier d'ajouter `import sys` en début de feuille)
  - × on peut fixer la taille maximale de la pile d'appels par la commande : `sys.setrecursionlimit(m)` (`m` représente alors le nombre maximum d'appels autorisés).

II. Des exemples de fonctions récursives

II.1. Des petits exemples sur les listes

Rappels sur les listes

- Afin de réaliser des fonctions récursives sur les listes, commençons par rappeler le principe du slicing (« tranchage ») d'une liste. Si `L` est une liste :
  - l'appel `L[i:j]` (où i et j sont deux entiers) permet d'obtenir la liste obtenue à partir de la liste `L` en ne conservant que les éléments situés entre l'indice `i` et l'indice `j-1`.
  - l'appel `L[i:]` (où i est un entier) permet d'obtenir la liste obtenue à partir de la liste `L` en ne conservant que les éléments situés entre l'indice `i` et l'élément final de la liste.
  - l'appel `L[:j]` (où j est un entier) permet d'obtenir la liste obtenue à partir de la liste `L` en ne conservant que les éléments situés entre le début de la liste et l'élément situé à l'indice `j-1`.
- On rappelle que si `L` est une liste et `x` un élément, alors la commande `L.append(x)` permet de modifier la liste `L` en lui ajoutant `x` comme dernier élément.
- On rappelle que si `L1` et `L2` sont deux listes, alors la commande `L1 + L2` crée la liste obtenue par concaténation des listes `L1` et `L2`.

Dans cette section, on présente quelques algorithmes consistant à effectuer une opération sur la liste par inspection de tous les éléments de cette liste. Les algorithmes présentés pourront se coder de manière récursive en exploitant l'idée que pour effectuer l'opération sur la liste en entier on :

- effectue tout d'abord l'opération sur le premier élément de la liste,
- effectue ensuite l'opération sur le reste de la liste par un appel récursif.

Évidemment, il faudra aussi définir un cas de base. Généralement, il s'agit de préciser ce qu'effectue l'opération sur une liste vide (ou une liste à un élément).

► Implémenter, de manière récursive, la fonction `nbElementListe` qui prend en paramètre une liste et renvoie le nombre d'éléments de la liste.

► Implémenter, de manière récursive, la fonction `sommeListe` qui prend en paramètre une liste et renvoie la somme de tous les éléments de cette liste.

► Implémenter, de manière récursive, la fonction `ajoutListe` qui prend en paramètre un élément et une liste et qui renvoie la liste obtenue en ajoutant l'élément en paramètre à tous les éléments de la liste.

► Implémenter, de manière récursive, la fonction `rechercheListe` qui prend en paramètre un élément et une liste et qui renvoie `True` si l'élément est dans la liste et `False` sinon.

(il est important de préciser que, du fait de l'implémentation du `or`*, les appels successifs sont arrêtés dès que* `L[0] == x` *est évalué à* `True` *- l'ordre dans lequel les conditions apparaissent est important)*

► Implémenter, de manière récursive, la fonction `positifListe` qui prend en paramètre une liste et qui renvoie `True` si tous les éléments de la liste sont positifs et `False` sinon.

II.2. Extension à des fonctions opérant sur des matrices

- Dans cette section, on se propose d'étendre les fonctions précédentes à des matrices qui seront présentées sous forme d'une liste de listes (chacune de ces listes représente alors une ligne de la matrice).
- Pour effectuer une opérations sur une telle matrice on :
  × effectue tout d'abord l'opération sur la première liste de la matrice,
  × effectue ensuite l'opération sur le reste de la matrice (les listes restantes) par un appel récursif.

► Implémenter, de manière récursive, la fonction `nbElementMatrice` qui prend en paramètre une liste et renvoie la longueur de la liste.

► Implémenter, de manière récursive, la fonction `sommeMatrice` qui prend en paramètre une liste et renvoie la somme de tous les éléments de cette liste.

► Implémenter, de manière récursive, la fonction `positifMatrice` qui prend en paramètre une matrice et qui renvoie `True` si tous les éléments de la matrice sont positifs et `False` sinon.

II.3. Exponentiation rapide

- Écrire une fonction récursive `expo_naive` qui prend en paramètres un réel `x` et un entier `n` et renvoie la valeur de $\mathtt{x}^{\mathtt{n}}$. On utilisera la relation de récurrence suivante :

$$\forall n \in \mathbb{N}, \quad x^n \; = \; \begin{cases} 1 & \text{si } n = 0 \\ x^{n-1} \times x & \text{sinon} \end{cases}$$

- Démontrer la terminaison de cette fonction.

- Que se passe-t-il si l'on effectue l'appel `expo_naive(1, 4000)` ?

- On peut remarquer que, pour tout $n \in \mathbb{N}^*$:

$$x^n \; = \; \begin{cases} (x^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ (x^2)^{\frac{n-1}{2}} \times x & \text{si } n \text{ est impair} \end{cases}$$

À l'aide de cette relation, proposer une fonction récursive `expo_rapide` qui prend en paramètres un réel `x` et un entier `n` et renvoie la valeur de `x`$^{\mathtt{n}}$.

- Démontrer la terminaison de cet algorithme.

- On cherche maintenant à évaluer la complexité (ou coût) de cet algorithme. Fixons x et notons $C(n)$ le nombre total de multiplications effectuées lors du calcul de x^n par `expo_rapide`.
On peut démontrer :

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C(n) = C\left(\dfrac{n}{2}\right) + 1 & \textit{si } n > 0 \textit{ pair} \\ \forall n \in \mathbb{N}^*,\ C(n) = C\left(\dfrac{n-1}{2}\right) + 2 & \textit{si } n \textit{ impair} \end{cases}$$

- Expliquer ce résultat.

► Pour déterminer $C(n)$, on se propose tout d'abord de réaliser l'étude de la suite $(U(n))_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} U(0) = 0 \\ \forall n \in \mathbb{N}^*,\ U(n) = U\left(\left\lfloor \dfrac{n}{2} \right\rfloor\right) + 1 \end{cases}$$

Démontrer : $U(n) \underset{n\to+\infty}{=} O\big(\lfloor \log_2(n) \rfloor\big)$.

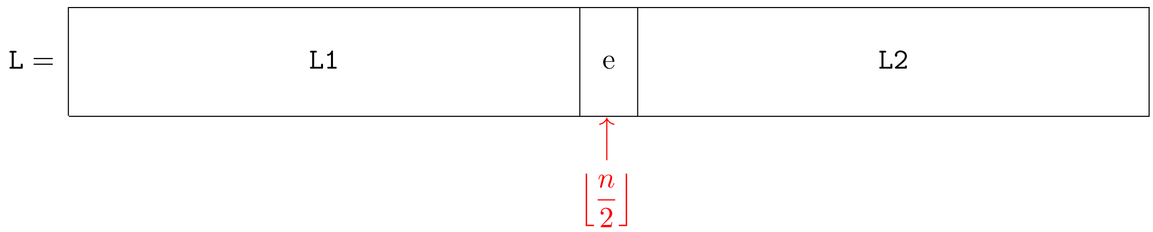
► Par récurrence forte, démontrer :

$$\forall n \in \mathbb{N},\ U(n) \ \leqslant\ C(n) \ \leqslant\ 2\,U(n) \tag{*}$$

► Conclure alors : $C(n) \underset{n\to+\infty}{=} O\big(\ln(n)\big)$.

II.4. Recherche dichotomique d'un élément dans une liste triée

- Dans le TP précédent, on a vu comment réaliser la recherche d'un élément dans une liste triée de manière impérative. Dans cette partie, on reprend cet algorithme et on l'implémente de manière récursive.
- Détaillons le procédé. On considère une liste `L` de longueur `n` triée dans l'ordre croissant. On note `e` l'élément qui se situe « au milieu » de cette liste. On obtient alors le découpage suivant :



La recherche dichotomique d'un élément `x` dans `L` se base sur le principe suivant :

- si `x` vaut `e` alors l'élément `x` est bien dans `L`,
- si `x` est strictement plus grand que `e` alors on effectue de manière récursive la recherche de `x` dans la liste `L2`.
- si `x` est plus petit que `e` alors on effectue de manière récursive la recherche de `x` dans la liste `L1`.

▶ Étant donnée une liste `L`, quelles commandes permettent d'obtenir les listes `L1` et `L2` définies par le schéma ci-dessus ?

▶ Implémenter la fonction `rechercheDichotoListe` qui réalise la recherche dichotomique d'un élément dans une liste par la méthode précisée plus haut.

II.5. Étude d'une suite récurrente d'ordre 2

On considère la suite (u_n) suivante : $\begin{cases} u_0 = 1 \\ u_1 = 0 \\ \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + 2u_n \end{cases}$

- Écrire une fonction `suiteU` qui prend en paramètre un entier `n` et renvoie le terme d'indice `n` de la suite (u_n). On effectuera une implémentation récursive en prenant soin de mettre en place les procédés présentés en partie ***I***.

- Calculer u_5 puis u_{15}, u_{30} et u_{50} à l'aide de la fonction précédente.
  Que dire du temps d'exécution nécessaire à ce calcul ?

- Combien d'appels sont nécessaires au calcul de u_5 ? Tracer l'arbre d'appels.

► Combien d'appels, environ, sont nécessaires au calcul de $n!$?