TP Informatique n° 3
Récursivité et programmation dynamique

I. 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 retrouve aussi ce schéma lors de la définition de la fonction factorielle :

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

La définition récursive d'un objet se base sur les principes suivants.

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 diminue donc strictement au cours des appels successifs. *(dans le cas de factorielle, la définition de $n!$ est donnée en fonction de $(n-1)!$)*

2) La définition doit donner la valeur de l'objet de taille minimale. On parle alors du cas initial. *(dans le cas de factorielle, $0! = 1$)*

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.

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

$5! = 5 \times 4! = 5 \times 4 \times 3! = 5 \times 4 \times 3 \times 2! = 5 \times 4 \times 3 \times 2 \times 1! = 5 \times 4 \times 3 \times 2 \times 1 \times 0!$
Ainsi : $5! = 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 120$.

I.2. Mise en place en informatique

I.2.a) Illustration avec la fonction factorielle

La définition récursive de fonctions est possible en **Python**.
Illustrons ce procédé avec la fonction factorielle.

```
1  def factrec(n):
2      """
3      La fonction fact prend en argument
4      un entier n et renvoie n!
5      """
6      assert(n >= 0 and type(n) == int)
7      if n == 0:
8          return 1
9      else:
10         return n*factrec(n-1)
```

▶ Que se passe-t-il si l'on réalise l'appel `facto(5)` ? Quel est le rôle de l'instruction `assert` ?

- Si la condition en paramètre de l'instruction `assert` est vérifiée, le reste du code est exécuté.
- Si ce n'est pas le cas, un message d'erreur est levé.

Dans le cas de la fonction factorielle, l'instruction `assert` permet de vérifier que le paramètre est toujours un entier positif. Cette condition permet d'assurer la terminaison des appels à la fonction `facto`.

▶ Combien d'appels sont nécessaires pour le calcul de 5! ? De n! ?

- D'après la question précédente, 5! nécessite 6 appels à la fonction factorielle.
- De manière générale, le nombre d'appels pour calculer $n!$ est donné par la valeur $D(n)$ qui vérifie la relation : $\begin{cases} D(0) = 1 \\ \forall n \in \mathbb{N},\ D(n+1) = 1 + D(n) \end{cases}$
  On en déduit que : $\forall n \in \mathbb{N},\ D(n) = n + 1$.

I.2.b) 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 la valeur de

```
1  def suiteU(n):
2      """
3      La fonction suiteU prend en argument
4      un entier n et renvoie u-n
5      """
6      assert(n >= 0 and type(n) == int)
7      if n == 0:
8          return 1
9      elif n == 1:
10         return 0
11     else:
12         return suiteU(n-1) + 2 * suiteU(n-2)
```

▶ 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 ?

- On obtient : $u_5 = 10$, $u_{15} = 10922$ et $u_{30} = 357913942$.
- Le calcul de u_{15} nécessite plusieurs secondes (cela dépend évidemment de la machine sur laquelle est faite ce calcul). On stoppe le calcul de u_{50}, beaucoup trop long.

▶ Combien d'appels sont nécessaires au calcul de u_5 ? Tracer l'arbre d'appels.
Combien d'appels, environ, sont nécessaires au calcul de $n!$?

- Pour calculer u_5, on demande le calcul de u_3 et u_4. Le calcul de u_3 demande le calcul de u_1 et u_2 ; le calcul de u_4 demande le calcul de u_2 et u_3.
  Le caclul de u_3 nécessite encore la calcul de u_2 ...
- Cet exemple simple permet de comprendre pourquoi le temps de calcul est si long : de nombreux calculs sont réalisés plusieurs fois.
- Chaque niveau de l'arbre d'appels contient deux fois plus d'appels que le niveau précédent (deux derniers niveaux mis à part).

  Pour le calcul de $n!$, on réalise donc environ $\sum_{k=0}^{n-1} 2^k = 2^n - 1$ appels.

II. Programmation dynamique

L'approche récursive est inadapté dans le cas précédent du fait de la multiplicité des résolutions du même sous-problème. Il serait plus judicieux de calculer le $n^{\text{ème}}$ terme de la suite à l'aide du paradigme de **programmation dynamique**. Il se base sur les principes suivants.

1) Les résultats intermédiaires sont gardés en mémoire. Toute solution optimale est alors calculé comme combinaison de sous-problème résoluts localement de façon optimale.
(la programmation dynamique est souvent utilisée dans des problèmes d'optimisation)

2) L'approche est ascendante (*bottom-up* en anglais) : on part de la solution des sous-problèmes les plus petits et on remonte jusqu'à obtenir la solution du problème initial.
(cela s'oppose à l'approche descendante - top-down - précédente)

► Écrire une fonction `suiteProgD` qui prend en paramètre un entier `n` et renvoie u_n à l'aide de l'approche de programmation dynamique. On utilisera deux variables `rangP` et `rangPP` permettant de garder en mémoire les calculs aux rangs précédents ainsi qu'une variable `rangA` permettant de connaître la valeur de la suite au rang actuel.

```
def suiteProgD(n):
    """
    La fonction suiteProgD prend en argument un entier n et
    renvoie u-n via une approche de programmation dynamique
    """
    assert(n >= 0 and type(n) == int)
    rangA = 1
    rangPP = 1
    rangP = 0
    for i in range(3,n+1):
        rangA = rangP + 2 * rangPP
        rangPP = rangP
        rangP = rangA
    return rangPP
```

► Calculer u_5, u_{15}, u_{30} et u_{50} à l'aide de ce programme. Calculer u_{10000} à l'aide de ce programme. Commenter. Donner le nombre d'affectations effectuées pour calculer u_n.

- On obtient évidemment le même résultat que précédemment.
- Le temps d'exécution du programme est largement meilleur.
  Le calcul de u_{10000} est non seulement réalisable mais presque immédiat.
- À chaque tour de boucle, 3 affectations sont réalisés.
  Comme on effectue n tours de boucles, on réalise en tout $3n$ affectations.
- La complexité en temps est donc linéaire en la taille du paramètre. Elle est exponentielle pour le codage récursif. Cette différence explique l'énorme gain observé en terme de temps de calcul.