Récursivité et mémoïsation

I Rappels sur la récursivité

Une fonction est dite *récursive* si, au cours de son exécution, elle fait appel à elle même (avec des paramètres différents). Elle est en général constituée d'un cas de base auquel les différents appels récursifs permettent de se ramener.

```
def f(...) :
    if ... : # cas de base

    else : # la récursivité
```

Ce mode de programmation est particulièrement adaptée à certaines situations où le résultat de la fonction à un « rang n » se déduit des valeurs aux « rangs précédents ».
On peut illustrer cela par exemple par le calcul des termes de la suite de Fibonacci $(f_n)_{n\in\mathbb{N}}$:

$$f_0 = f_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, f_{n+2} = f_{n+1} + f_n$$

Le calcul peut se programmer de la façon suivante :

```
def f(n) :
    if n in [0,1] :
        return 1
    else :
        return f(n-1)+f(n-2)
```

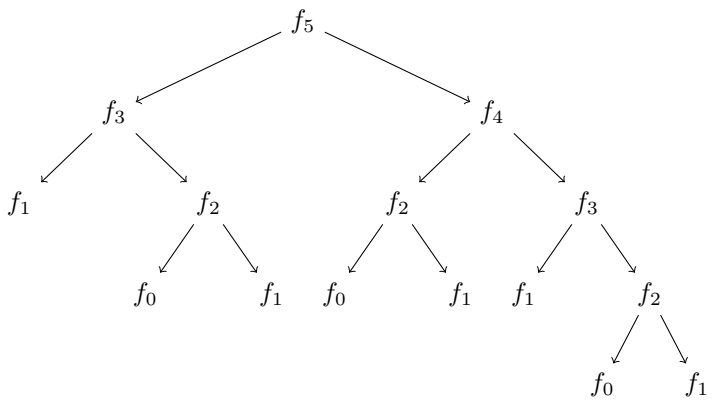
Ce code pose différents problèmes :

- l'utilisation de la récursivité limite les valeurs possibles de l'entier n car le nombre d'appels récursifs est limité par la taille de la pile d'exécution,
- les deux appels à la fonction `f` dans le calcul de `f(n)` (récursivité multiple) posent des problèmes de complexité.

Si on note C_n le nombre d'additions effectuées dans le calcul de f_n alors on a

$$C_0 = C_1 = 0 \quad \text{et} \quad \forall n \in \mathbb{N}, C_{n+2} = C_{n+1} + C_n + 1$$

On en déduit alors $C_n = f_n - 1 \sim \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$; on a donc une complexité exponentielle.
On peut comprendre le problème de complexité si on examine par exemple les différents appels récursifs effectués lors du calcul de f_5 (la phase de descente, qui ramène les calculs au cas de base)



On se rend compte que pour calculer f_5, on va calculer :
- f_4
- 2 fois f_3
- 3 fois f_2

Une solution pour contourner le problème, sur cet exemple, serait de calculer récursivement le couple (f_n, f_{n+1}), ie transformer la fonction en une récursivité simple) :

```
def fiboCouple(n) :
    if n == 0 :
        return (1,1)
    else :
        a,b = fiboCouple(n-1)
        return (b,a+b)
```

Pour obtenir la valeur de f_n, il suffit de demander

```
>>> fiboCouple(n)[0]
```

Enfin, pour éviter cette dernière syntaxe trop lourde, on peut définir la fonction récursive `fiboCouple` à l'intérieur de la fonction `fibo` (une sous-fonction) :

```
def fibo(n) :
    def fiboCouple(n) :
        if n == 0 :
            return (1,1)
        else :
            a,b = fiboCouple(n-1)
            return (b,a+b)
    return fiboCouple(n)[0]
```

Sur cet exemple simple, on peut aussi remplacer le calcul récursif (qui part de l'entier n pour descendre au cas de base avant d'effectuer le calcul) par un calcul itératif :

```
def Fibo(n) :
    a,b = 1,1
    for _ in range(n) :
        a,b = b,a+b
    return a
```

Une telle fonction effectue en fait les mêmes calculs que la fonction `fibo` mais en partant directement du cas de base (c'est la phase de remontée du calcul récursif) ; on parle de calcul de *bas en haut*.
En plus d'éviter les problèmes de complexité posées par la première forme de récursivité, cette programmation itérative évite aussi les problèmes de pile d'exécution et permet donc de calculer des valeurs de f_n pour des entiers n plus grands.

II Mémoïsation

Prenons comme deuxième exemple le calcul des coefficients binomiaux à partir de la formule de Pascal :

$$\binom{n}{p} + \binom{n}{p+1} = \binom{n+1}{p+1}$$

On peut le programmer récursivement de façon naïve :

```
def b(n,p) :
    if p == 0 or n == p :
        return 1
    else :
        return b(n-1,p-1) + b(n-1,p)
```

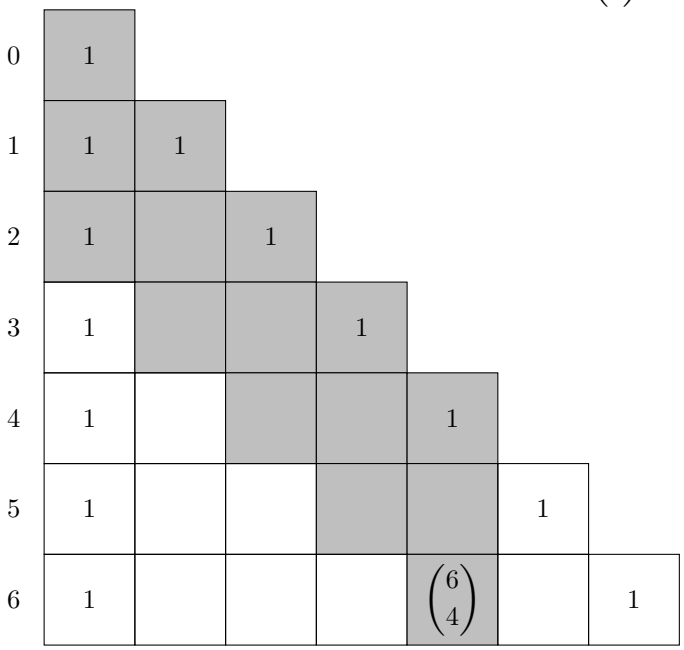
Ce code va poser exactement les mêmes problèmes que la première programmation de la suite de Fibonacci : l'appel récursif multiple va engendrer une complexité exponentielle due à des calculs de coefficients refaits plusieurs fois.

Cette fois ci, pour calculer $\binom{n}{p}$, on se rend compte qu'il suffit de connaître les coefficients de la ligne précédente (pour l'entier $n-1$) ; on pourrait donc calculer récursivement les coefficients de la ligne n avant d'extraire celui qui nous intéresse :

```
def B(n,p) :
    def binomeLigne(n) :
        if n == 0 :
            return [1]
        else :
            L = binomeLigne(n-1) # la ligne n-1
            l = [1] * (n+1)      # construction de la ligne n
            for k in range(1,n) :
                l[k] = L[k-1]+L[k]
            return l
    return binomeLigne(n)[p]     # extraction du coeff souhaité
```

Cette solution n'est pas optimale non plus car elle calcule, pour déterminer $\binom{n}{p}$, tous les coefficients de toutes les lignes précédentes alors qu'ils ne sont pas tous nécessaires (on pourrait se contenter de calculer ceux de la $n-1^{\text{ème}}$ ligne avant de déterminer $\binom{n}{p}$, et pas toute la $n^{\text{ème}}$ ligne, mais le défaut subsiste sur les lignes précédentes).

Coefficients binomiaux nécessaires au calcul de $\binom{6}{4}$



Afin de ne mémoriser que les valeurs utiles, on introduit un dictionnaire dans lequel seuls les résultats des appels récursifs seront stockés :

```
binoDico = {}

def binom(n,p) :
    if (n,p) not in binoDico :
        if p == 0 or n == p :
            b = 1
        else :
            b = binom(n-1,p-1) + binom(n-1,p)
        binoDico[(n,p)] = b
    return binoDico[(n, p)]
```

Si, à la suite du calcul de `binom(6,4)`, on fait afficher le dictionnaire, on obtient

```
{(2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (2, 2): 1, (3, 2): 3,
 (4, 2): 6, (3, 3): 1, (4, 3): 4, (5, 3): 10, (4, 4): 1, (5, 4): 5, (6, 4):
  15}
```

qui correspond bien aux cases grisées sur le schéma précédent : seules les valeurs indispensables ont été calculées et mémorisées.
Le dictionnaire a été défini comme une variable globale, ce qui peut présenter un avantage dans le cas de plusieurs calculs successifs : après chaque calcul le dictionnaire est enrichi par les nouvelles valeurs qui n'auront donc pas besoin d'être recalculée pour une utilisation suivante de la fonction `binom`.

Cette technique consistant à introduire un dictionnaire lors d'une programmation récursive afin de mémoriser les résultats intermédiaires utiles s'appelle *mémoïsation*.

III Résumé des étapes de la mise en place de la mémoïsation avec un dictionnaire

Version « naïve »

```
def f(n) :
    if n in [0,1] :
        r = 1
    else :
        r = f(n-1)+f(n-2)
    return r
```

Mémoïsation :
on introduit un dictionnaire en variable globale

```
d = {}

def g(n) :
    if n not in d :
        if n in [0,1] :
            r = 1
        else :
            r = g(n-1)+g(n-2)
        d[n] = r
    return d[n]
```

La version « définitive » :
on « enferme » le dictionnaire et la sous-fonction dans la fonction finale
la valeur renvoyée par la fonction finale fait appel à la sous-fonction récursive

```
def F(n) :
    d = {}
    def g(n) :
        if n not in d :
            if n in [0,1] :
                r = 1
            else :
                r = g(n-1)+g(n-2)
            d[n] = r
        return d[n]
    return g(n)
```

Exemple(s) :

III.1 Écrire une fonction itérative permettant de calculer $\binom{n}{p}$, toujours à partir de la formule de Pascal (on pourra introduire un tableau de taille $n \times p$ que l'on remplira au fur et à mesure).
Quel est l'intérêt de la programmation récursive ?

III.2 On considère la suite $(\alpha_n)_{n\in\mathbb{N}}$ définie par

$$\alpha_0 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}, \alpha_{n+1} = \frac{1}{n+1}\sum_{k=0}^{n} \alpha_k$$

Suite très intéressante puisqu'en fait (α_n) est la suite constante égale à 1, mais faisons comme si on ne le savait pas....

a) Écrire un code récursif « naïf » `alpha(n:int)->int` qui calcule α_n

b) Faire de même avec un calcul de bas en haut.

c) Écrire une fonction `Alpha(n:int)->int` qui utilise le principe de mémoïsation.

d) Comparer l'efficacité des trois fonctions en calculant α_{30}.

e) Expliquer les différences en évaluant la complexité des trois fonctions