TP-Cours d'Informatique 06

Récursivité

ITC – PCSI^2 2024-2025 – Lycée Fabert (Metz) – L. Parise – S.Limal – D.Mengel

I. Principe de la récursivité en programmation. Premiers exemples

I.1. Principe général. Exemple de la fonction factorielle

Définition : On dit qu'une fonction est récursive si elle peut s'appeller elle-même une ou plusieurs fois lors de son exécution.

Lorsqu'une fonction f est récursive, on veillera, lors de l'analyse d'un programme par exemple, à distinguer

- l'**appel principal** de la fonction f qui a lieu lorsque f n'est pas encore en cours d'exécution, en général dans le « programme principal »,
- les **appels récursifs** de f qui sont les appels provoqués par l'exécution de f elle même.

Par opposition, lorsqu'une fonction contient de boucles **for** et/ou **while** et ne s'appelle par elle-même, on dit que **f est itérative** ou qu'**elle suit un algorithme itératif**.

En mathématiques, de nombreux objets peuvent avoir une définition itérative et/ou récursive (on dit qu'ils sont définis par récurrence), comme la fonction factorielle :

$$\underbrace{n! = 1 \times 2 \times 3 \times \cdots \times n = \prod_{k=1}^{n} k.}_{\text{définition itérative de } n!} \qquad \underbrace{n! = \left| \begin{array}{ll} 1 & \text{si } n = 0 \\ n(n-1)! & \text{sinon.} \end{array} \right.}_{\text{définition récursive de } n!}$$

On donne ci-dessous les programmes Python correspondants respectivement aux définitions précédentes :

```
1 def fact(n):
2     p = 1
3     for i in range (1,n+1):
4         p = p * i
5     return p
```

```
1 def factorielle(n):
2     if n==0:
3         return 1
4     else:
5         return n*factorielle(n-1)
```

De même, l'algorithme d'Euclide fournit une méthode récursive du calcul du `PGCD` de deux entiers a et b :

$$\texttt{PGCD}(a,b) = \left| \begin{array}{ll} a & \text{si } b = 0 \\ \texttt{PGCD}(b, a \bmod b) & \text{sinon ,} \end{array} \right.$$

$a \bmod b$ désignant ici le reste de la division euclidienne de a par b. On peut également concevoir une fonction dans laquelle interviennent plusieurs appels récursifs de celle-ci. C'est le cas pour le calcul du n^{e} terme de la suite de Fibonacci :

$$\left\{ \begin{array}{l} u_0 = 1\,,\ u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n\,, \quad \text{pour tout } n \in \mathbb{N}. \end{array} \right.$$

Dans chacun des exercices suivants, nous allons construire des fonctions récursives pour répondre aux problèmes qui nous sont posés. Certains d'entre eux pourront avoir une solution non récursive simple, mais d'autres verront leur résolution grandement simplifiée par l'utilisation d'un algorithme récursif.

Pour déterminer si un problème peut avoir un algorithme récursif de résolution et, le cas échéant, pour mettre en œuvre cet algorithme, nous nous proposons de suivre les étapes :

a. décomposer le problème en sous-problèmes, ce qui fixera l'algorithme et permettra de vérifier si le problème peut être résolu de manière récursive.

b. déterminer quelles sont les données nécessaires à la résolution du problème.

c. déterminer une ou plusieurs conditions d'arrêt de l'algorithme. Pour ce faire, on considérera des cas simples où la solution peut être donnée **sans** appel récursif. Dans tous les cas, on vérifiera qu'à chaque appel récursif, on se rapproche de la condition d'arrêt et qu'ainsi **l'algorithme se termine bien**.

I.2. Définition récursive d'une suite de courbes de limite une courbe fractale

Une application classique de la récursivité est la représentation de courbes fractales. Une courbe fractale est la limite d'une suite de courbes (ici des lignes brisées) définie par récurrence. Pour représenter la n^{e} courbe d'une telle suite (il est impossible de dessiner la limite de ces courbes), nous allons définir les différentes actions permettant son tracé.

Après avoir défini le point de départ de la courbe (polygonale) et une direction de départ, il nous faut pouvoir changer de direction et avancer ou reculer d'une certaine longueur dans la direction choisie : on pourra assimiler les déplacements successifs réalisant le tracé demandé à ceux d'une tortue.

1. Se familiariser avec l'usage du module turtle inspiré du langage LOGO de Seymour Papert en s'entraînant sur les exercices de la page `http://logo.pcsi2.net` .

On considère les trois suites de courbes définies ci-après :

Courbe de Von Koch $(K_n)_{n\in\mathbb{N}}$

K_0 est un segment de longueur a.
K_1 est ce même segment au milieu duquel est découpé un triangle équilatéral de côté $\frac{a}{3}$.
K_{n+1} est construit à partir de K_n en appliquant la même opération sur chacune de ses arêtes.

Figure 1 – *Courbe de Von Koch*

Courbe de Minkowski $(M_n)_{n\in\mathbb{N}}$

M_0 est un segment de longueur a.
M_1 est ce même segment sur lequel sont découpés en son milieu deux carrés de côtés $\frac{a}{4}$.
M_{n+1} est construit à partir de M_n en en appliquant la même opération sur chacune de ses arêtes.

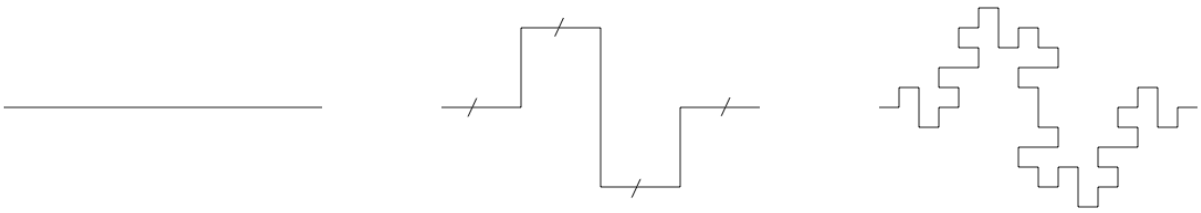


Figure 2 – *Courbe de Minkowski*

Courbe du dragon $(D_n)_{n\in\mathbb{N}}$

D_0 est un segment porté par le bipoint (A, B).

D_1 est la ligne brisée formée des points A, C et B tels que ABC soit isocèle et $(\overrightarrow{CA}, \overrightarrow{CB}) = \pi/2$.

D_{n+1} est construit sur le bipoint (A, B) en construisant D_n sur le bipoint (A, C) et le bipoint (B, C).

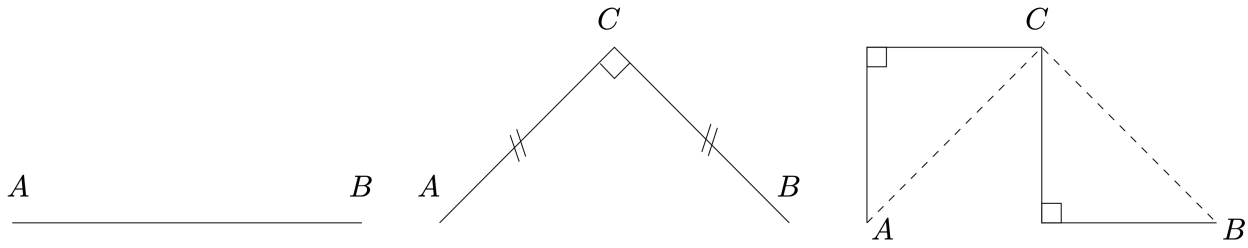


FIGURE 3 – *Courbe du dragon*

2. Définir les fonctions VonKoch(a:**float**,n:**int**)→None, Minkowski(a:**float**,n:**int**)→None ainsi que dragon(a:**float**,n:**int**)→None réalisant respectivement le tracé de K_n et M_n et D_n à partir d'un segment de longueur a.

I.3. Algorithme récursif des tours de Hanoï

Dans le célèbre Temple de Bénarès, sous le dôme marquant le centre du monde, trône un socle de bronze sur lequel sont fixées trois aiguilles de diamant, chacune d'elles haute d'une coudée et fine comme la taille d'une guêpe. Sur une de ces aiguilles, à la création, Dieu empila soixante quatre disques d'or pur, du plus grand au plus petit, le plus large reposant sur le socle de bronze. Il s'agit de la tour de Bramah. Jour et nuit, inlassablement, les moines déplacent les disques d'une aiguille vers l'autre tout en respectant les immuables lois de Bramah qui obligent les moines à ne déplacer qu'un disque à la fois et à ne jamais le déposer sur un disque plus petit. Quand les soixante quatre disques auront été déplacés de l'aiguille sur laquelle Dieu les déposa à la Création vers une des autres aiguilles, la tour et le temple seront réduits en poussière, et le monde disparaîtra dans un grondement de tonnerre.

3. Essayer de résoudre ce problème avec 2, 3 puis 4 disques. Montrer comment le problème à $n+1$ disques peut être résolu si l'on sait résoudre le problème à n disques ($n \in \mathbb{N}^*$) .
4. Utiliser le principe précédent pour écrire une fonction récursive donnant la solution du problème en affichant tous les déplacements effectués sous la forme `"Déplacement d'un disque de l'aiguille A vers l'aiguille B"` lorsqu'on a nommé respectivement A, B et C les 3 aiguilles.

   Ainsi, lorsqu'il y a 3 disques sur l'aiguille A, l'appel de votre fonction devra produire l'affichage suivant :

```
"Déplacement d'un disque de l'aiguille A vers l'aiguille C"
"Déplacement d'un disque de l'aiguille A vers l'aiguille B"
"Déplacement d'un disque de l'aiguille C vers l'aiguille B"
"Déplacement d'un disque de l'aiguille A vers l'aiguille C"
"Déplacement d'un disque de l'aiguille B vers l'aiguille A"
"Déplacement d'un disque de l'aiguille B vers l'aiguille C"
"Déplacement d'un disque de l'aiguille A vers l'aiguille C"
```

   On notera qu'il y a eu 7 mouvements de disque pour réaliser le déplacement des 3 disques de l'aiguille A vers l'aiguille C.

5. Si les moines déplacent une pièce par seconde en suivant le procédé décrit au 1, sans commettre d'erreur, donner la date de la fin du monde.
6. Dans cette question, nous souhaitons simuler plus précisément l'état des aiguilles (les tours de Hanoï) au départ et après chaque déplacement de disque afin de visualiser toutes les étapes du processus décrit précédemment. Nous reprenons le **même** algorithme, mais en utilisant la liste tours contenant trois listes représentant les trois aiguilles et les disques se trouvant à un instant donné sur chaque aiguille.

   Par exemple, lorsqu'il y a 3 disques sur l'aiguille A, on définira la liste tours de la sorte :

```
tours=[[3,2,1], [], []]
```

   Ainsi, tours[0] représentera l'état de l'aiguille A (aiguille n° 0), tours[1] celui de l'aiguille B (aiguille n° 1) et tours[2] celui de l'aiguille C (aiguille n° 2). Pour simuler le déplacement d'un disque en cours de processus, on retirera le dernier élément d'une des trois listes pour l'ajouter à la fin d'une autre de ces listes. Pour ce faire, on utilisera les méthodes pop et append.

```
tours=[[3,2,1], [], []]
disque=tours[0].pop()
tours[2].append(disque)
print(tours)
disque=tours[0].pop()
tours[1].append(disque)
print(tours)
```

Listing 1 – Exemple d'utilisation de pop et append pour simuler le déplacement d'un disque

Écrire la fonction

Hanoi2(tours:**list**,depart:**int**,via:**int**,arrivee:**int**,n:**int**)→None

qui modifiera la liste tours pour simuler tous les déplacements de disque et affichera à chaque étape l'état des aiguilles.

Lorsqu'au départ il y a 3 disques sur l'aiguille A, la fonction Hanoi2 devra produire l'affichage qui suit la première ligne ci-contre :

```
[[3,2,1], [], []]
[[3,2], [], [1]]
[[3], [2], [1]]
[[3], [2,1], []]
[[], [2,1], [3]]
[[1], [2], [3]]
[[1], [], [3,2]]
[[], [], [3,2,1]]
```

II. Pile d'appels. Intérêt et limites de la programmation récursive

II.1. Pile d'appels récursifs

En informatique, la pile d'exécution/pile d'appels (en anglais, Call Stack) est une structure de données de type pile qui sert à enregistrer des informations au sujet des **fonctions actives** dans un programme.

Les fonctions actives sont celles qui ont été appelées, mais n'ont pas encore terminé leur exécution. La principale utilisation de la **pile d'appels** est de garder la trace de l'adresse où le flux d'exécution d'un programme doit se poursuivre lorsqu'une fonction active termine sa propre exécution.

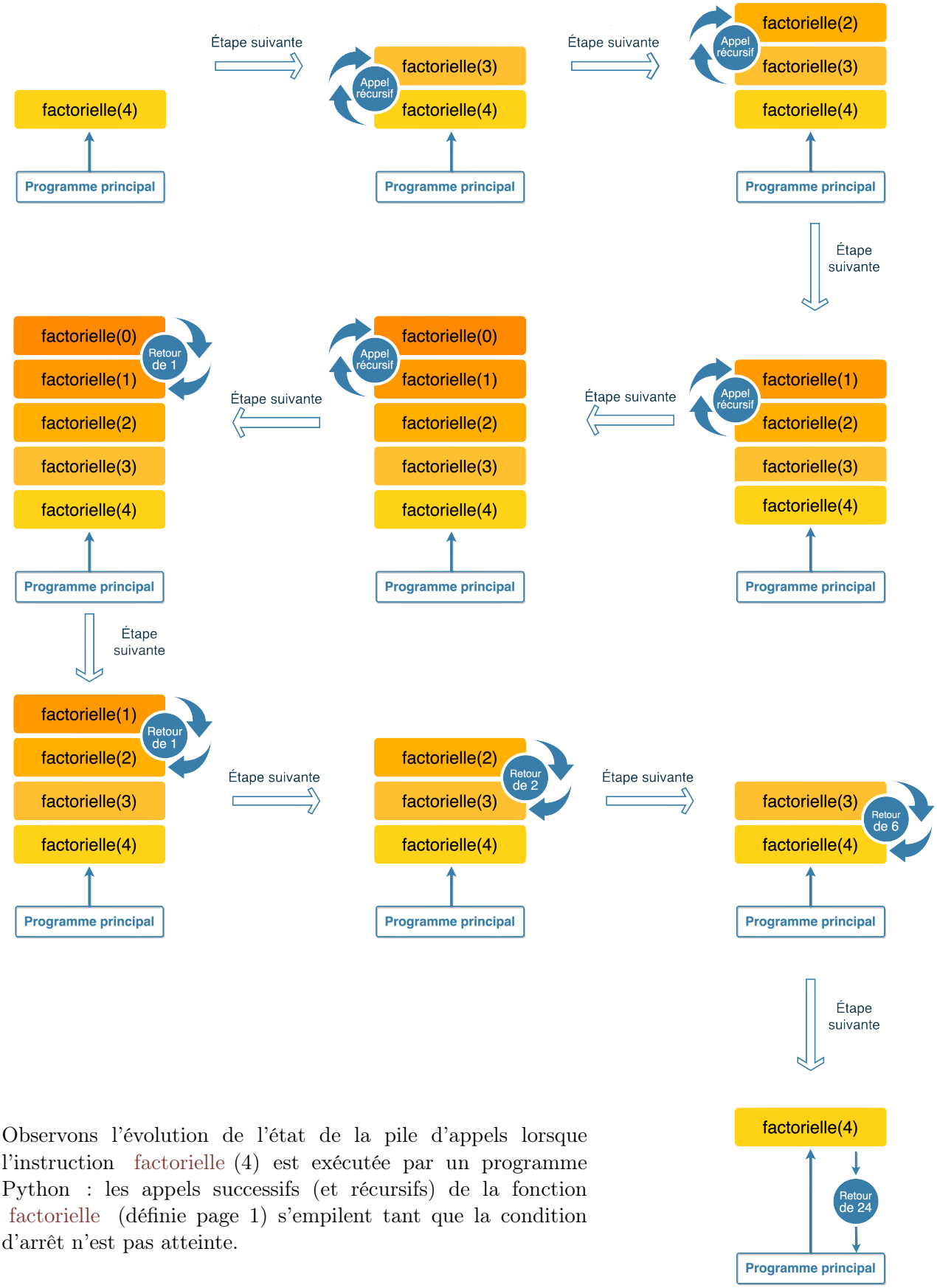
Toutefois, la pile d'appels emmagasine aussi d'autres valeurs associées à chaque fonction active comme ses **variables locales**, **ses paramètres**, etc. Dans le TP-Cours 06 (*Utilisation de modules*), la fonction ecart_type appelle la fonction moyenne et se faisant, pousse l'adresse de retour du flux d'exécution sur la pile, de sorte que la fonction moyenne, quand elle se termine, récupère l'adresse de retour au sommet de la pile d'appels et y transfère le contrôle du flux du programme.

Ainsi, les adresses de retour s'accumulent sur la pile d'appels et sont récupérées une à une lors de la fin de l'exécution de chaque fonction.

Lors de l'appel principal d'une fonction récursive, les appels récursifs induits sont donc stockés dans la pile d'appels jusqu'à atteindre la condition d'arrêt : c'est la **phase de descente**. Une fois la condition

d'arrêt (des appels récursifs) obtenue, ces appels récursifs (ou pluttôt les adresses de retour) sont désempilés : c'est la **phase de remontée**.

Le nombre d'appels récursifs en Python est limité à 1000 (taille maximale de la pile d'exécution). Si l'appel de fact(1000) ne pose aucun problème, l'appel de factorielle (1000), quant à lui, provoque le message d'erreur : RecursionError : maximum recursion depth exceeded in comparison



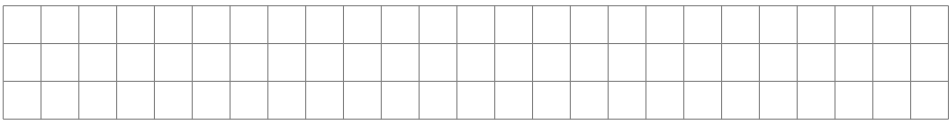
Observons l'évolution de l'état de la pile d'appels lorsque l'instruction factorielle (4) est exécutée par un programme Python : les appels successifs (et récursifs) de la fonction factorielle (définie page 1) s'empilent tant que la condition d'arrêt n'est pas atteinte.

II.2. Redondance d'appels, mémoïsation

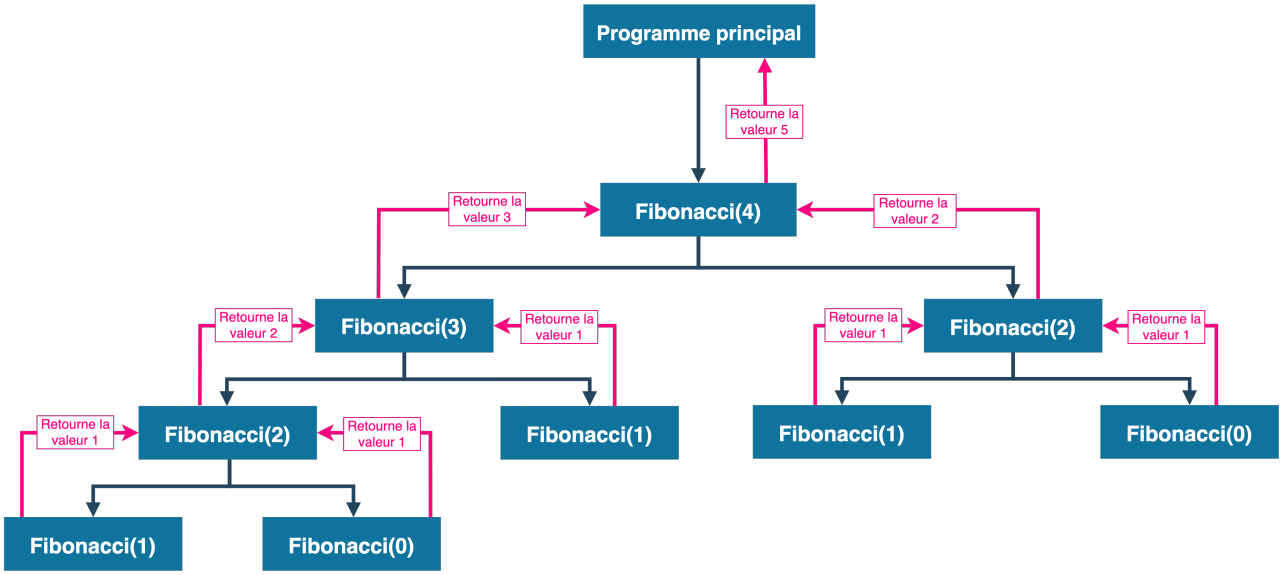
Dans cette partie, on s'intéresse à la suite de Fibonacci pour illustrer avantages et inconvénients des fonctions récursives. La suite de Fibonacci notée ici $(u_n)_{n\in\mathbb{N}}$ est définie par

$$\begin{cases} u_0 = 1\,,\ u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n\,, \quad \text{pour tout } n \in \mathbb{N}. \end{cases}.$$

7. À partir de la définition précédente, écrire une fonction récursive Fibonacci(n:**int**)→**int** qui renvoie le n$^{\text{e}}$ terme de la suite u c'est-à-dire u_n. Calculer u_{15} et u_{30}.



On s'aperçoit très rapidement du manque d'efficacité de cette fonction récursive, due aux multiples appels de la fonction Fibonacci avec le **même** paramètre n ! Ainsi, sur la figure ci-dessous, Fibonacci(0) est exécutée 2 fois, Fibonacci(1) est exécutée 3 fois et Fibonacci(2) 2 fois.



8. Pour se faire une idée de l'explosion combinatoire du nombre d'appels récursifs et donc du nombre d'additions conduisant à la valeur de u_n lors de l'utilisation de la fonction Fibonacci, nous allons modifier cette fonction pour écrire la fonction

   Fibonacci2(n:**int**, compteurs:**list**)→**int**

   qui renvoie le n$^{\text{e}}$ terme de la suite u et stocke le nombre d'appels récursifs et le nombre d'additions effectués pour réaliser ce calcul dans la liste compteurs.

   Indication : On initialisera la variable compteurs à [0,0] dans le programme principal. Pour calculer u_{15}, il faut 1973 appels de la fonction Fibonacci2 et pour calculer u_{30}, il en faut 2 692 537.

Pour remédier à ce problème de la **redondance des appels récursifs**, nous allons utiliser la technique de la **mémoïsation**.

En informatique, la mémoïsation est la mise en cache des valeurs de retour d'une fonction selon ses valeurs de paramètres. Une fonction mémoïsée stocke les valeurs renvoyées par ses appels précédents dans une structure de données adaptée (ici, un dictionnaire) et, lorsqu'elle est appelée à nouveau avec les mêmes paramètres, renvoie la valeur stockée au lieu de la recalculer. La mémoïsation est une façon de diminuer le temps de calcul d'une fonction, au prix d'une occupation mémoire plus importante. La mémoïsation modifie donc la complexité d'une fonction, en temps comme en espace.

Pour écrire la prochaine fonction, nous allons garder l'idée des compteurs (pour pouvoir effectuer une comparaison avec la fonction Fibonacci2) et utiliser la possibilité en langage Python de définir des paramètres de fonctions ayant des valeurs par défaut lorsque ladite fonction est appelée sans préciser ces paramètres.

```
>>> def ajouter(a,b=5):
        return a+b

>>> ajouter(14,15)
29
>>> ajouter(14)
19
```

Listing 2 – Exemple d'utilisation de paramètres de fonction avec valeur par défaut

9. Écrire la fonction Fibonacci3(n:**int**, compteurs:**list**, u={})→**int** où u est un dictionnaire vide lors de l'appel principal de cette fonction et qui, par un algorithme récursif, renvoie le n^{e} terme de la suite u.

   Lors d'un appel récursif de Fibonacci3 avec le paramètre entier n, la fonction vérifiera si la valeur de u_n se trouve déjà dans le dictionnaire u. Le cas échéant, c'est cette valeur qui sera retournée par la fonction. Dans le cas contraire, la fonction calculera $u_{n-1} + u_{n-2}$ obtenu par appels récursifs, ajoutera cette valeur au dictionnaire u et la retournera.

10. Afin de comparer les fonctions Fibonacci2 et Fibonacci3, nous allons importer et utiliser le module time à l'instar de l'exemple ci-dessous qui mesure le temps d'exécution de la fonction factorielle :

```
debut = time.time()
f = factorielle(20)
fin = time.time()
print(fin-debut,"ms")
```

Listing 3 – Exemple d'utilisation de la fonction time() du module time

Faites de même pour comparer le temps d'exécution des fonctions Fibonacci2 et Fibonacci3 appelées avec un paramère n compris entre 20 et 35.

11. Une autre façon d'éviter les problèmes d'appels redondants d'un fonction récursive consiste à essayer de trouver un algorithme itératif réalisant le même travail. Dans le cas de la suite de Fibonacci, c'est relativement simple : on utilisera deux variables u et v qui vont contenir sucessivement les valeurs 1 et 1 puis 1et 2 puis 2 et 3 puis 3 et 5, ... etc.

    Écrire la fonction Fibonacci4(n:**int**)→**int** qui renvoie le n^{e} terme de la suite u en suivant un algorithme itératif.