1 Exemples introductifs

1.1 Auto-référencement et auto-similarité

Les sigles auto-référentiels sont des sigles qui font appels à l'auto-référence selon un procédé de mise en abyme littéraire. Quelques exemples :

- Bing : *Bing Is Not Google*,
- WINE : *Wine Is Not an Emulator*,
- FIJI : *FIJI Is Just ImageJ* (logiciel de traitement d'images),
- TikZ : *TikZ Ist Kein Zeichenprogramm* (TikZ n'est pas un logiciel de dessin).

Le chou Romanesco est un exemple de structure *auto-similaire* :

La fraction infinie ci-dessous est un exemple d'auto-référencement dans une expression mathématique [1] :

$$x = \cfrac{1}{1+\cfrac{1}{1+\ldots}}$$

1.2 Calcul d'une somme à l'aide d'une fonction qui s'auto-référencie

Considérons la fonction `somme(n)` définie ci-dessous :

```
def somme(n):
    if n==0 :
        return 0
    else :
         return n+somme(n-1)
```

Cette fonction renvoie la somme S_n des n premiers entiers naturels non nuls

$$S_n = \sum_{k=1}^{n} k$$

en exploitant la propriété

$$S_0 = 0 \quad \text{et} \quad \forall n \in \mathbb{N}^*, \quad S_n = n + S_{n-1}.$$

> Dans l'exemple précédent, l'instruction `else` n'est pas indispensable.

1.3 Suite de Fibonacci

La fonction `fibo(n)` renvoie le n-ième terme de la suite de Fibanocci définie par récurrence sur deux rangs, selon

$$\mathcal{F}_0 = 0, \quad \mathcal{F}_1 = 1, \quad \text{et} \quad \forall n \in \mathbb{N}^*, \quad \mathcal{F}_{n+1} = \mathcal{F}_n + \mathcal{F}_{n-1}.$$

```
def fibo(n): # fonction récursive
    if n==0:
        return 0
    if n==1:
        return 1
    return fibo(n-1)+fibo(n-2)
```

1. En mathématique, l'ensemble de Cantor est également un exemple de structure auto-similaire.

Pour bien comprendre l'algorithme, il peut être utile de reformuler la relation de récurrence en *décalant l'indice* n :

$$\text{pour tout entier } n > 1, \quad \mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2}.$$

On constate que cette fonction est structurée comme suit :

- Les cas $n = 0$ et $n = 1$ sont traités à part. Ils constituent la **condition d'arrêt**.
- Lorsque $n > 1$, l'exécution de la fonction engendre deux appels à la même fonction, avec les arguments $n - 1$ et $n - 2$ décroissants.

Pour évaluer une fonction récursive, le processeur utilise une **pile d'appel**.

Par exemple, pour calculer `fibo(3)`, le processeur crée un appel à la fonction `fibo` avec $n = 3$. Dans l'*espace de noms* relatif à cet appel, il rencontre les expressions `fibo(2)` et `fibo(1)`. Il traite d'abord l'appel à `fibo(2)` qu'il empile sur la pile d'exécution en ouvrant un *nouvel espace de noms* relatif à ce nouvel appel, cet espace de noms étant totalement indépendant de celui de la fonction appelante (`fibo` avec $n = 3$). De nouveau, dans le traitement de `fibo(2)`, il rencontre `fibo(1)` et `fibo(0)`. Il traite en premier `fibo(1)` en l'empilant encore sur la pile d'appel. Lors de l'exécution de `fibo(1)`, le processeur tombe sur la condition d'arrêt qui lui permet de *dépiler* `fibo(1)` de la pile d'appel. L'exécution se poursuit ainsi jusqu'à remonter à l'appelant initial. *In fine*, pour évaluer `fibo(3)`, les appels successifs à la fonction `fibo` se font dans l'ordre suivant pour l'argument n (cf fig. 1) :

$$n : 3 \to 2 \to 1 \to 0 \to 1$$

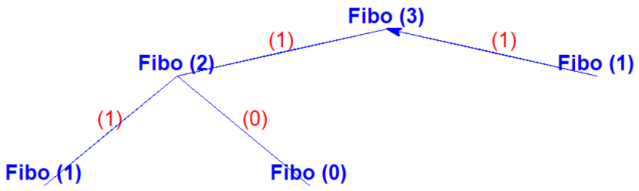

FIGURE 1 – Illustration des appels à `fibo(3)`, les valeurs renvoyées sont en rouge.

Exercice 1 : pile d'exécution

Déterminer la liste des valeurs de n empilées par le processeur pour évaluer `fibo(5)`.

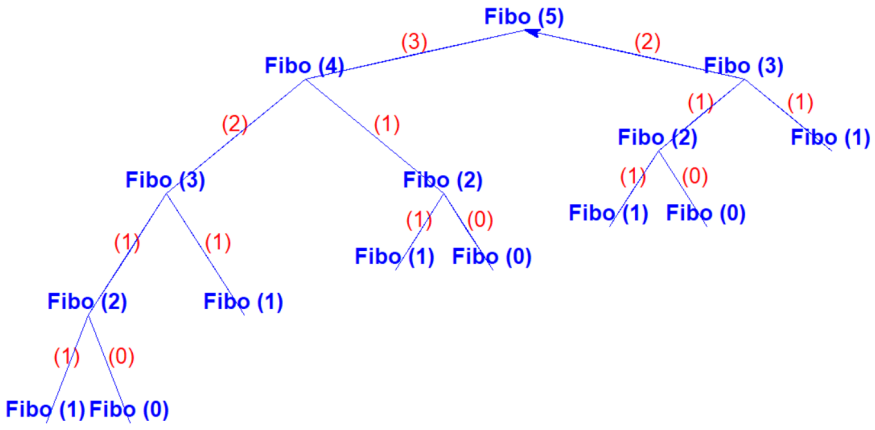

FIGURE 2 – Illustration de l'algorithme récursif pour la suite de Fibonacci.

Solution : la suite des valeurs prise par l'argument n lors des appels successifs est $5 \to 4 \to 3 \to 2 \to 1 \to 0 \to 1 \to 2 \to 1 \to 0 \to 3 \to 2 \to 1 \to 0 \to 1$.

On remarque clairement que cet algorithme n'est pas optimal car les appels à `fibo(1)` et `fibo(0)` sont

effectués à de multiples reprises [a].

a. Cet inconvénient peut être contourné en utilisant la technique de *mémoïsation*, qui consiste à retenir les valeurs retournées par la fonction pour éviter de les calculer deux fois.

2 Théorie des fonctions récursives

La récursivité est un moyen de répéter un bloc d'instructions sans utiliser `for` ou `while`. Pour cela on utilise une fonction qui va s'appeler elle-même.

Définition 2.1 : fonction récursive

Une fonction est dite récursive si elle contient une instruction qui l'appelle elle-même.

La programmation récursive permet généralement d'écrire des algorithmes élégants, compacts et proches de l'écriture mathématique. La notation de fonction récursive est étroitement liée au **principe de récurrence**. Toutefois, les algorithmes récursifs ne sont pas généralement performants en termes de complexité en temps ou en espace comme nous allons le voir dans les exemples qui suivent.

Tout algorithme récursif peut être transformé en un algorithme itératif équivalent : c'est la *dérécursivation* [a].

a. Les techniques de dérécursivation sont hors-programme

2.1 Récursivité simple, multiple, mutuelle

La récursivité est dite :

- **simple**, lorsque la fonction comporte un appel récursif unique. Exemple :

```
def fac(n):
    if n==0 :
        return 1
    return n*fac(n-1)
```

- **multiple**, lorsque la fonction comporte plusieurs appels récursifs. Exemple : la programmation récursive de la suite de Fibonacci en utilisant directement la relation $u_n = u_{n-1} + u_{n-2}$ (cf `Fibo(n)`) possède deux appels récursifs.
- **mutuelle**, lorsque deux fonctions f et g comportent des appels croisés, c'est-à-dire que l'évaluation de f au rang n requiert d'évaluer la fonction g à un (ou des) rang(s) inférieur(s) à n et que, de même, l'évaluation de g au rang n requiert d'évaluer la fonction f à un (ou des) rang(s) inférieur(s).

Exercice 2 : Récursivité mutuelle

Écrire en python deux fonctions `suiteU(n)` et `suiteV(n)` qui permettent de calculer pour tout $n \in \mathbb{N}$ les termes des suites (u_n) et (v_n) définies par

$$u_0 = 1\,, v_0 = 0 \quad \text{et, pour tout } n \in \mathbb{N}, \begin{cases} u_{n+1} &= 3u_n - 2v_n + n \\ v_{n+1} &= -u_n + 3v_n + 1 \end{cases}$$

Solution :

```
def suiteU(n):
    if n==0 :
        return 1
    return 3*suiteU(n-1)-2*suiteV(n-1)+n-1 # <- attention n-1 !
def suiteV(n):
    if n==0 :
        return 0
    return -suiteU(n-1)+3*suiteV(n-1)+1
```

On prend garde à bien *décaler* la relation de récurrence pour l'écrire par rapport au terme u_n, soit $u_n = 3u_{n-1} - 2v_{n-1} + (n-1)$. De même pour v_n.

2.2 Preuve de terminaison

Pour se terminer, un algorithme récursif doit toujours revenir sur une **condition d'arrêt**.

Pour prouver la terminaison d'un algorithme récursif, on exhibe généralement une suite d'entiers naturels strictement décroissante. Il peut s'agir de l'argument n de la fonction récursive.

Preuve de terminaison pour la fonction `fac(n)`

A chaque appel de la fonction `fac(n)`, la valeur de l'argument n diminue de 1. Or, lorsque $n = 0$, la fonction renvoie 1 et se termine (c'est la condition d'arrêt). Ainsi, la suite des valeurs prises par l'argument n est donc strictement décroissante et minorée par zéro. Le nombre d'appels à la fonction `fac(n)` est donc fini, ainsi l'algorithme se termine.

2.3 Preuve de correction

Pour prouver la correction d'un algorithme récursif, on procède généralement par **récurrence** (simple, multiple ou forte [2]).

Considérons l'exemple suivant qui implémente un nouvel algorithme pour le calcul des termes de la suite de Fibonacci en utilisant une fonction qui renvoie un couple de valeurs :

```
1 def fiboDouble(n): # algorithme récursif sur deux termes successifs
2     if n==0 :
3         return [0,1]
4     else :
5         prev = fiboDouble(n-1)
6          return [prev[1],prev[0]+prev[1]]
```

Commençons d'abord par prouver la terminaison de l'algorithme. La fonction `fiboDouble` ne comporte qu'un seul appel récursif, il s'agit donc de récursivité simple. Pour $n = 0$, la fonction renvoie `[0,1]`, c'est la condition d'arrêt. Tant que $n > 0$, à chaque appel, l'argument n diminue de 1. La suite des valeurs prises par l'argument n est donc une suite strictement décroissante d'entiers naturels. Elle ne prend donc qu'un nombre fini de valeurs. L'algorithme se termine.

Pour prouver la correction, on doit montrer que l'appel à `fiboDouble(n)` renvoie la paire de valeurs consécutives de la suite de Fibonacci, notée $(\mathcal{F}_n, \mathcal{F}_{n+1})$.

Preuve de correction de l'algorithme de `fiboDouble(n)`

Soit $\mathscr{P}(n)$ l'assertion « `fiboDouble(n)` renvoie $[\mathcal{F}_n, \mathcal{F}_{n+1}]$ ».

- **Initialisation**. `fiboDouble(0)` renvoie le couple [0,1], soit $[\mathcal{F}_0, \mathcal{F}_1]$. Ainsi $\mathscr{P}(0)$ est vraie.
- **Hérédité**. Soit $n \in \mathbb{N}$, on suppose $\mathscr{P}(n)$ vraie, c'est-à-dire que `fiboDouble(n)` renvoie $[\mathcal{F}_n, \mathcal{F}_{n+1}]$. Considérons l'appel à `fiboDouble` avec l'argument `n` $= n+1$. L'argument de `fiboDouble` étant `n` $= n+1$, il est supérieur à 0. Donc, comme l'instruction `prev = fiboDouble(n-1)` est exécutée avec `n` $= n+1$, la variable `prev` contient donc $[\mathcal{F}_n, \mathcal{F}_{n+1}]$. La fonction renvoie ensuite `[prev[1],prev[0]+prev[1]]` soit $[\mathcal{F}_{n+1}, \mathcal{F}_n + \mathcal{F}_{n+1}]$. Or, par définition de la suite de Fibonacci, $\mathcal{F}_n + \mathcal{F}_{n+1} = \mathcal{F}_{n+2}$. Ainsi, « la fonction `fiboDouble(n+1)` renvoie
$$[\mathcal{F}_{n+1}, \mathcal{F}_n + \mathcal{F}_{n+1}] = [\mathcal{F}_{n+1}, \mathcal{F}_{n+2}] \text{ »},$$
ce qui est précisément l'assertion $\mathscr{P}(n+1)$.
- **Conclusion.** On en déduit par récurrence que, pour tout $n \in \mathbb{N}$, `fiboDouble(n)` renvoie $[\mathcal{F}_n, \mathcal{F}_{n+1}]$.

Remarques :

- Bien que la suite $(\mathcal{F}_n)$ soit définie par récurrence sur deux rangs, la démonstration s'effectue en récurrence simple (et non multiple).
- Il aurait été possible de **regrouper les preuves** de terminaison et de correction en raisonnant par récurrence sur l'assertion « `fiboDouble(n)` se termine et renvoie $[\mathcal{F}_n, \mathcal{F}_{n+1}]$ ».

2. On parle de récurrence forte lorsque pour établir l'*hérédité* d'une propriété $\mathscr{P}(n)$ (*i.e.* passer de n à $(n+1)$), on doit supposer sa véracité à tous les rangs inférieurs (*i.e.* $((\forall\, 0 \leq k \leq n,\ \mathscr{P}(k)) \Rightarrow \mathscr{P}(n+1))$. On parle de récurrence multiple lorsque, pour établir $\mathscr{P}(n+1)$, on doit supposer non seulement $\mathscr{P}(n)$ mais aussi $\mathscr{P}(n-1)$ (il s'agit d'une récurrence d'ordre 2). Dans ce cas, il faut deux conditions d'initialisation d'indices consécutifs, c'est-à-dire avoir prouvé que $\mathscr{P}(0)$ est VRAIE et que $\mathscr{P}(1)$ est VRAIE. Les récurrences forte ou multiple se ramènent toujours à de la récurrence simple à condition de choisir convenablement la propriété $\mathscr{P}(n)$.

2.4 Complexité

La complexité temporelle est liée au temps d'exécution, elle dépend du rang d'appel n. On peut quantifier la complexité temporelle, par exemple, en comptant le nombre d'appels, en comptant le nombre d'opérations mathématiques, de comparaison, etc...

La complexité en mémoire comptabilise le nombre de cases mémoires utilisées pour l'algorithme (on compte généralement une case pour chaque variable de type élémentaire (entier, flottant, booléen ou caractère), et $n =$ `len(L)` cases pour une liste L de n variables élémentaires).

Le calcul de complexité $C(n)$ s'effectue généralement de manière récursive.

Terme général d'une suite à récurrence linéaire

Rappel. Comment déterminer le terme général d'une suite à récurrence linéaire de la forme [a] :

$$(u_1 \text{ et } u_0 \text{ données}) \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+2} = a\,u_{n+1} + b\,u_n + c$$

- On part de la relation homogène associée $u_{n+2} = a\,u_{n+1} + b\,u_n$ (obtenue en retirant le terme constant c). Pour cela, on cherche les racines r_1 et r_2 de l'équation caractéristique associée
$$r^2 - a\,r - b = 0.$$
- On cherche une solution particulière (v_n) de la relation de récurrence « complète » en prenant une suite constante ($\forall n,\ v_n = V$). V vérifie donc
$$V = a\,V + b\,V + c.$$
- Le terme général de (u_n) est donc de la forme
$$\forall n \in \mathbb{N}, \quad u_n = Ar_1^n + Br_2^n + V$$
où les constantes A et B sont obtenues grâce aux deux conditions initiales
$$u_0 = Ar_1^0 + Br_2^0 + V \quad \text{et} \quad u_1 = Ar_1^1 + Br_2^1 + V.$$

a. Si au lieu de la constante c, on a une expression polynomiale en n, de degré k, au lieu d'une solution particulière (v_n) constante, on cherche une suite polynomiale de degré k. Si au lieu de racines simples, l'équation caractéristique possède une racine double $r0$, le terme général de la relation homogène devient $(An + B)r_0^n$. A titre d'entraînement établir l'expression du terme général de la suite $(u_n)_{n\in\mathbb{N}}$ vérifiant $u_{n+2} = u_{n+1} + 2u_n + 3n - 1$, $u_0 = 0$ et $u_1 = 1$.

Réponse : $u_n = 2^n - \frac{3}{4}(-1)^n - 3/2n - 1/4$

Exercice 3 : complexité temporelle

Calculer la complexité temporelle de la fonction `fibo(n)` donnée au § 1.3 obtenue en comptant le nombre d'additions effectuées.

Solution : soit $C(n)$ le nombre d'additions effectuées pour évaluer `fibo(n)`. On a $C(0) = 0$ et $C(1) = 0$ car l'appel à la fonction avec l'argument $n = 0$ ou $n = 1$ renvoie le résultat sans aucune addition. Pour $n > 1$, pour évaluer la fonction `fibo(n)`, il faut effectuer une addition, plus toutes celles pour évaluer `fibo(n-1)` et toutes celles pour évaluer `fibo(n-2)`. Ainsi, pour n entier supérieur à 1, la complexité $C(n)$ vérifie la relation de récurrence suivante :

$$C(n) = C(n-1) + C(n-2) + 1.$$

Ou bien de manière totalement équivalente, pour $n \in \mathbb{N}$,

$$C(n+2) = C(n+1) + C(n) + 1.$$

- Le polynôme caractéristique est $r^2 - r - 1 = 0$. Ses racines sont
$$r_1 = \varphi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad r_2 = \frac{1-\sqrt{5}}{2} = -\varphi^{-1} - 1$$
- La suite constante de terme général V est solution si $V = V + V + 1$ soit $V = -1$.
- Le terme général de $(C(n))$ est donc
$$C(n) = Ar_1^n + Br_2^n - 1.$$
Comme $C(0) = C(1) = 0$, il vient $A + B - 1 = 0$ et $Ar_1 + Br_2 - 1 = 0$ d'où $A = r_1/\sqrt{5}$ et $B = -r_2/\sqrt{5}$. Ainsi, la complexité temporelle de la fonction `fibo(n)` est

$$\boxed{\forall n \in \mathbb{N}, \quad C(n) = \frac{1}{\sqrt{5}}\varphi^{n+1} - \frac{1}{\sqrt{5}}\left(-\varphi\right)^{-n-1} - 1}$$

> Pour n grand, $C(n) \underset{+\infty}{\sim} \frac{\varphi^n}{\sqrt{5}}$, complexité de cet algorithme est donc en $\mathcal{O}(\varphi^n)$: il s'agit d'une complexité exponentielle, c'est très mauvais.

Notons qu'il est possible de *suivre* durant l'exécution la complexité du programme en comptabilisant le nombre d'additions dans une variable globale `nbAdd` que l'on incrémente du nombre d'opérations effectuées :

```
import math # pour la racine carrée
def fibo(n): # fonction récursive
    global nbAdd
    if n==0:
        return 0
    if n==1:
        return 1
    nbAdd+=1 # la variable global est accessible en écriture
    return fibo(n-1)+fibo(n-2)
def C(n): # complexité théorique
    phi=(1+math.sqrt(5))/2
    return (phi**(k+1)-(-phi)**(-k-1))/math.sqrt(5)-1
# script d'appel
for k in range(9):
    nbAdd = 0 # initialisation du compteur
    print(k,fibo(k),nbAdd,int(C(k))) # int : convertir en entier
```

On obtient :

```
0 0 0 0
1 1 0 0
2 1 1 1
3 2 2 2
4 3 4 4
5 5 7 7
6 8 12 12
7 13 20 20
8 21 33 33
```

3 Optimisation d'un algorithme récursif

Nous allons voir quelques techniques pour améliorer l'efficacité d'un algorithme récursif diminuant sa complexité en temps et/ou en espace.

3.1 Récursivité terminale

Reprenons l'exemple de la fonction `somme(n)` du § 1.2, dans laquelle la dernière instruction est `return n+somme(n-1)`. Pour exécuter cette instruction, le processeur a besoin de conserver la valeur de la variable `n` tout en lançant l'appel récursif à la fonction `somme(n-1)`. Ainsi, la pile d'exécution conserve la valeur de l'argument n tant que le résultat de la fonction `somme(n-1)` n'est pas remontée. On peut schématiser la pile d'exécution pour `n=4` de la manière suivante :

```
somme(4) = 4 + somme(3)
   somme(3) = 3 + somme(2)
      somme(2) = 2 + somme(1)
         somme(1) = 1 + somme(0)
            somme(0) = 0
         somme(1) = 1
      somme(2) = 3
   somme(3) = 6
somme(4) = 10
```

Ainsi, par rapport à une programmation itérative avec une boucle `for`, l'algorithme récursif n'est efficace, ni en terme d'espace mémoire, ni en terme de temps d'exécution (l'empilement des données étant également coûteux en temps de calcul). Il est possible de contourner cet inconvénient en évitant que le processeur n'ait à stocker des données temporaires. Il faut pour cela que la dernière instruction à exécuter soit l'appel récursif (et non l'addition comme dans cet exemple).

Définition 3.1 : récursivité terminale (ou finale ou *tail-recursive*)

Une fonction est dite à récursivité terminale si, dans cette fonction, l'appel récursif est la **dernière** instruction avant le `return`.

Dans un tel algorithme, il n'y a donc qu'un seul appel récursif qui est effectué à la fin. Voici une version en récursivité terminale de la fonction `somme` :

```
def sommeTerm(n,res=0): # version terminale
    if n==0 :
        return res
    else:
     return sommeTerm(n-1,res+n)
```

Lors de l'appel à la fonction `sommeTerm(4)`, l'argument `res`, qui est optionnel, est initialisé à zéro. Le processeur termine l'exécution de la fonction par l'appel récursif. Ainsi, la pile d'exécution n'est parcourue qu'une fois en descendant :

```
sommeTerm(4, res=0) = sommeTerm(3, res = 4)
sommeTerm(3, 4) = sommeTerm(2, res = 7)
sommeTerm(2, 7) = sommeTerm(1, res = 9)
sommeTerm(1, 9) = sommeTerm(0, res = 10)
sommeTerm(0, 10) = res = 10
```

Le second argument `res` joue le rôle d'une *variable tampon* qui sert à conserver le résultat des opérations d'addition (tout comme dans une version itérative). Cette variable n'est pas dupliquée, mais plutôt transférée à chaque appel récursif à la fonction du niveau inférieur. Lorsque l'appel récursif est effectué, la fonction appelante est terminée et sa mémoire libérée. Les différents appels se font successivement **à la manière d'une itération simple**.

Exercice 4 : algorithme itératif

Écrire en Python la version itérative `sommeIter(n)` de la fonction `sommeTerm(n)` en utilisant une boucle `for`.

L'exécution de l'algorithme itératif est très similaire à la version en récursivité terminale.

```
def sommeIter(n): # version itérative
    res=0
    for k in range(n+1): # attention au n+1 !
        res+=k
     return res
```

3.2 Récurrence simple vs récurrence multiple

La récurrence double est à éviter tant que possible. Voyons comment on peut s'en affranchir.

Exercice 5 : fonction mystère

Décrire le rôle de la fonction suivante : donner, sous forme mathématique, la définition de la suite (u_n) dont la fonction `u(n)` permet le calcul. S'agit-il de récursivité simple, multiple, mutuelle ? S'agit-il d'une complexité logarithmique, linéaire, quadratique, exponentielle, ... ? Justifier brièvement. Proposer une amélioration notable de ce programme tout en conservant une version récursive. Calculer sa complexité $C(n)$ en ne comptant que la division par u_n (opération la plus coûteuse en temps d'exécution).

```
def u(n,A=2) :
    if n==0 :
        return 1
    return 0.5 * (u(n-1,A)+A/u(n-1,A))
```

Solution : la suite (u_n) définie par l'algorithme `u(n)` est mathématiquement définie ainsi

$$u_0 = 1 \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{A}{u_n}\right).$$

- L'algorithme est programmé en récursivité multiple (ici en récursivité double) car chaque exécution de la fonction requiert, pour $n > 0$, exactement deux appels récursifs.
- Lorsque n augmente de 1, le nombre d'appels double. La complexité est donc exponentielle [3], de l'ordre de $\mathcal{O}(2^n)$.
- Pour améliorer ce programme, il suffit de n'effectuer qu'un seul appel récursif. Cela est possible en utilisant une variable « intermédiaire ». Voici le programme ainsi modifié.

```
def u(n,A=2) :
    if n==0 :
        return 1
        x=u(n-1,A) # variable intermédiaire
    return 0.5 * (x+A/x)
```

- Soit $C(n)$ la complexité en nombre de divisions. On a $C(0) = 0$, et $\forall n \in \mathbb{N}$, $C(n+1) = 1 + C(n)$. On a donc immédiatement $\boxed{C(n) = n}$, la complexité est linéaire.

Dans la mesure du possible il faut éviter les récurrences multiples.

3.3 Optimisation : compromis temps *vs* espace

Prenons l'exemple de l'algorithme `fibo(n)` qui utilise naïvement la relation de récurrence et dont la complexité est exponentielle (en $\mathcal{O}(\varphi^n)$). Le problème de cet algorithme, tel que nous l'avons décrit sur la figure 2 qui présente le schéma de calcul de `fibo(5)`, est que les évaluations intermédiaires ($Fibo(3)$ ou $Fibo(2)$ par exemple) sont effectuées à de multiples reprises et les résultats de ces calculs ne sont pas réutilisés (cf fig. 2).

Pour améliorer cet algorithme, une idée naturelle est de conserver les valeurs $\mathcal{F}_k$ de la suite $(\mathcal{F}_n)$ dans une liste afin **d'éviter d'avoir à les recalculer**. Ainsi, on peut envisager de créer une liste `fiboList` fonctionnant sur le principe d'une variable globale dans laquelle on stocke les valeurs. Cette liste est initialisée avec la valeur `-1` qui code le fait que la valeur n'est pas connue. Voici le code Python correspondant :

```
# SOLUTION 1 : Fibonacci récursif avec mémorisation dans une var. globale
NMAX = 100
fiboList=[-1]*NMAX # initialisation à -1 => valeur inconnue
fiboList[0]=0 # valeur connue
fiboList[1]=1 # valeur connue
nbAppels=0
def fibo(n): # il faut n<NMAX!
    global nbAppels
    nbAppels+=1
    if fiboList[n]==-1 : # la valeur est inconnue?
        fiboList[n]=fibo(n-1)+fibo(n-2)# si oui, on la calcule
    return fiboList[n] # à ce stade, la valeur est connue
```

L'inconvénient de cette méthode est qu'il revient au script appelant de gérer l'existence de la variable globale (sa déclaration et son initialisation). Une autre méthode consiste à « encapsuler » la fonction récursive à l'intérieur d'une fonction qui jouera le conteneur pour la variable globale, de manière à ce que cette variable globale n'ait pas à être gérée au niveau du script d'appel.

```
# SOLUTION 2: Fibonacci avec mémorisation et encapsulation
def fibo(n):
    fiboList=[-1]*(n+2) # il faut (n+2) cases mémoire (cf cas n=0)
    fiboList[0]=0 # valeur connue
    fiboList[1]=1 # valeur connue
    def fiboRec(p): # fonction recursive non accessible de l'extérieur
        if fiboList[p]==-1 :
            fiboList[p]=fiboRec(p-1)+fiboRec(p-2) # fiboRec !
        return fiboList[p] # même principe
    return fiboRec(n) # appel à la sous-fonction
```

Pourquoi n'y a-t-il pas de condition d'arrêt testant `p==0` et `p==1` dans `fiboRec` ?

- La seconde solution est à privilégier dans le cas d'un appel unique avec un grand nombre n.

3. Soit $C(n)$ le nombre d'opérations élémentaires (additions, multiplications ou divisions). On a $C(0) = 0$ et pour $n > 0$, $C(n+1) = 2C(n) + 3$. On montre donc aisément que $C(n) = 3(2^n - 1) \underset{n\to\infty}{\sim} 3 \times 2^n = \mathcal{O}(2^n)$

Cette méthode illustre le principe du compromis temps *vs* mémoire : il est possible de diminuer le coût d'exécution à condition d'augmenter la consommation de mémoire [a].

a. La technique utilisée s'appelle *mémoïsation*, sa connaissance n'est pas exigible.

4 Quelques exemples

4.1 Dichotomie

Soit f une fonction continue monotone strictement sur un intervalle $I = [a \quad b]$ telle que $f(a).f(b) < 0$ dont on souhaite trouver le zéro, c'est-à-dire trouver la solution $x \in I$ de l'équation $f(x) = 0$ à ε près.

```
def dicho(f,a,b,eps): # zero d'une fonction monotone par dichotomie
    m = (a+b)/2
    if (b-a) < eps : # précision souhaité atteinte?
        return m
    if f(m) * f(a) >0 :
        return dicho(f,m,b,eps)
    else:
        return dicho(f,a,m,eps)
```

Si l'intervalle est suffisamment petit, on renvoie la valeur centrale, sinon on recommence la recherche dans le sous-intervalle $[a \quad m]$ ou $[m \quad b]$. Application pour trouver le zéro de la fonction $g : x \to x^2 - 2$ sur l'intervalle $[1 \quad 2]$.

```
g =lambda x:x**2-2 # lambda function
print(dicho(g,1,2,1e-8)) # annulation de g sur [1\quad2]
```

Le script affiche la valeur `1.414213564246893`.

> La *lambda function* `lambda x:x**2-2` permet une écriture compacte.

Analysons l'algorithme. Soit a_0 et b_0 les valeurs initiales des paramètres `a` et `b`.

- **Terminaison**. A chaque appel, la longueur de l'intervalle est divisée par deux. Ainsi, au bout de n récursions, sa longueur est $(b_0 - a_0)/2^n$. Il est évident qu'elle devient inférieure à toute valeur $\epsilon > 0$ lorsque n est suffisant grand. Donc l'algorithme se termine.
- **Correction**. L'assertion $\mathscr{P}(n)$ « la largeur de l'intervalle au rang n est $(b_0 - a_0)/2^n$ et$f(a).f(b) < 0$ » est vraie pour le premier appel. Elle est héréditaire. Lorsque l'algorithme se termine, le zéro appartient à l'intervalle $[a \quad b]$ dont la largeur est inférieure à ε, ce qui prouve la correction.
- **Complexité**. Le nombre n d'appels à la fonction est tel que
$$(b_0 - a_0)/2^n < \varepsilon.$$
Soit $2^n > \frac{b-a}{\varepsilon}$ donc $\boxed{n = \left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil}$. A chaque fois que l'on souhaite ajouter un chiffre significatif à la précision du zéro (on divise ε par 10), on augmente le nombre moyen de récursions de $\log_2(10) = \ln(10)/\ln(2) \approx 3,32$. La complexité du calcul augmente donc de **manière linéaire avec le nombre de chiffres significatifs** souhaités.

4.2 Recherche linéaire dans une liste

Pour savoir si une liste de n éléments contient un élément nul, on peut utiliser une recherche récursive basée sur le principe suivant :

- ou bien la liste est vide et la réponse est non,
- ou bien le dernier élément est l'élément nul et on a trouvé,
- ou bien il suffit de rechercher dans la sous-liste des $n-1$ premiers éléments.

Voici une première implémentation :

```
def contient0(L):
    if len(L)==0:
        return False
    elif L[0]==0:
        return True
    else:
        return contient0(L[1:]) # slicing de la liste
```

Voici une seconde implémentation :

```
def contient0bis(L):
    if len(L)==0:
        return False
    elif L[0]==0:
        return True
    else:
        del L[-1] # suppression du dernier élément
        return contient0bis(L)
```

Exercice 6

Comparer les deux algorithmes : évaluer leur complexité. Quel est l'état de la liste à la fin de l'appel à la fonction ?

Quel algorithme est-il le plus efficace en temps d'exécution compte-tenu du caractère *mutable* de l'objet `list` en python ?

Solution : Les deux algorithmes sont écrits en récursivité terminale. Ils semblent tous les deux de complexité linéaire (la liste n'étant *a priori* pas triée, on ne peut pas faire mieux). Toutefois avec le premier algorithme le *sciling* ; la liste est recopiée avant d'être passée comme argument à la fonction de rang inférieure. Ainsi, la liste originelle est conservée mais la copie engendre un surcoût de complexité (la copie de liste est de coût linéaire) : la complexité de ce premier algorithme est donc quadratique. Ceci n'est plus le cas pour le second algorithme : l'instruction `del L[-1]` supprime le dernier élément de la liste sans que celle-ci ne soit copiée (c'est l'opération symétrique que `L.append(elem)`). Ainsi, la liste est successivement vidée de ses derniers éléments jusqu'à que l'élément zéro soit trouvée (si aucun élément zéro n'est présent, alors la liste est entièrement vidée). En terme de temps d'exécution, le second algorithme est plus efficace car il évite l'opération de copie (sa complexité est effectivement linéaire alors qu'elle est quadratique pour le premier).

4.3 Recherche dichotomique dans une liste triée

Soit `L` une liste d'éléments que l'on suppose triés par ordre croissant. On souhaite savoir si l'élément x est dans la liste.

Exercice 7

Écrire en Python une fonction `contientDicho(L,x)` qui recherche l'élément x par dichotomie dans une liste précédemment triée par ordre croissant et qui renvoie True ou False selon que l'élément a été trouvé ou non.

Solution :

```
def contientDicho(L,x):
    if (len(L)==0) or (x<L[0]) or (x>L[-1]):
        return False
    c=len(L)//2
    lc=L[c] # lire la valeur du milieu de liste
    if lc==x:
        return True
    elif lc<x: #x est trop grand
        return contientDicho(L[c+1:],x) # chercher à droite
    else:
        return contientDicho(L[:c],x) # chercher à gauche
```

Les instructions de *slicing* `L[c+1:]` et `L[:c]` effectuent la recopie intégrale de la liste et possèdent donc un coût en $\mathcal{O}(n)$, rendant l'algorithme peu efficace.

On peut modifier la fonction précédente de manière à **renvoyer l'indice de l'élément trouvé**. Pour cela, il ne faut pas découper la liste en sous-listes sans précaution car on perdrait l'information de la position. On procède donc comme pour la recherche du zéro d'une fonction sur un intervalle : on va faire passer en argument les bornes inférieure et supérieure de l'intervalle d'indices dans lequel la recherche doit s'étendre. L'entête de la fonction est ainsi : `dichoRang(L,a,b,x)`. Elle renvoie le rang de l'élément x entre les indices $i = a$ et $i = b - 1$ de la liste L. Elle renvoie -1 si l'élément n'est pas trouvé.

Voici le code Python d'une telle fonction :

```
def dichoRang(L,a,b,x): # recherche dichotomique dans la liste L[a:b]
    if a>=b :
        return -1 # élément non trouvé
    c=(a+b)//2 # indice du milieu arrondi à l'entier inférieur
    lc=L[c] # lecture de la valeur du milieu
    if lc==x:
        return c # l'indice est renvoyé, élément trouvé
    elif lc<x:
        return dichoRang(L,c+1,b,x)#recherche à droite
    else:
        return dichoRang(L,a,c,x)#recherche à gauche
```

Exercice 8 : recherche dichotomique en récursivité terminale

En vous inspirant de la fonction précédente, écrire en Python une fonction `dichoRangTerm(L,x)` en récursivité terminale qui renvoie l'indice d'un élément recherché x dans une liste supposée triée par ordre croissant. Si l'élément n'est pas trouvé, la fonction doit renvoyer -1 [a].

a. Indication : on effectuera un *slicing* de la liste initiale pour chaque appel récursif. On ajoutera un argument à la fonction pour repérer la valeur de l'indice de liste avant le *slicing*.

Solution :

```
def dichoRangTerm(L,x,a=0): # recherche par dichotomie terminale
    if len(L)==0:
        return -1 # pas trouvé
    c=len(L)//2 # milieu de liste
    lc=L[c] # lire la valeur du milieu de liste
    if lc==x:
        return c+a
    elif lc<x:
        return dichoRangTerm(L[c+1:],x,a+c+1) # chercher à droite
    else:
        return dichoRangTerm(L[0:c],x,a) # chercher à gauche
```

4.4 Exponentiation rapide récursive

On souhaite calculer a^n avec a réel et n entier naturel à l'aide de la propriété suivante :

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a^{n/2} \times a^{n/2} & \text{si } n \text{ est pair} \\ a \times a^{(n-1)/2} \times a^{(n-1)/2} & \text{si } n \text{ est impair} \end{cases}$$

Voici l'algorithme correspondant.

```
def puissanceRapide(a,n):
    if n==0:
        return 1
    r=puissanceRapide( a , n//2)
    if n%2 == 0:
        return r*r
    else:
        return a*r*r
```

Cet algorithme peut également s'écrire en récursivité terminale ou de façon itérative.

4.5 Pour terminer avec Fibonacci

Exercice 9 : Fibonacci en récursivité terminale

Proposer un algorithme pour la fonction `fiboTerm(n,a=0,b=1)` renvoyant le n-ième de la suite de Fibonacci utilisant une récursivité terminale [a].

Prouver sa terminaison, sa correction et calculer sa complexité en comptant le nombre total d'additions et de tests.

a. Indication : pour la terminaison, on pourra utiliser un test sur la parité de `n`.

Solution : les variables `a` et `b` stockent deux termes successifs de la suite. La récurrence est simple car on calcule les nouvelles valeurs de ces termes à chaque appel.

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

La terminaison est évidente, dans la mesure où l'argument `n` décroît exactement de 1 à chaque appel. La condition d'arrêt `n==1` est donc rencontrée de manière certaine (si $n = 0$ lors de l'appel, la condition d'arrêt est satisfaite). Il reste à prouver la correction. Il reste donc à trouver la bonne *assertion* pour bâtir notre raisonnement.

> La démonstration de la correction est **nettement plus délicate** ici car la fonction `fiboTerm` dépend non seulement de `n` mais aussi de `a` et `b`. Ainsi, une assertion du genre « `fiboTerm(n)` renvoie $\mathcal{F}_n$ » n'est d'aucun secours car l'appel récursif au rang $n-1$ à `fiboTerm(n-1,b,b+a)` ne se fait pas avec les mêmes valeurs de `a` et de `b` que lorsque l'appel à `fiboTerm(n-1)` est fait directement depuis l'extérieur de la fonction.

A mesure que `n` diminue, les variables `a` et `b` prennent les valeurs successives de la suite $(\mathcal{F}_n)$ dans l'ordre croissant. Au rang n initial, on a l'égalité $[a, b] = [\mathcal{F}_0, \mathcal{F}_1]$. Notons n_0 la valeur initial de l'argument `n` lors du premier appel à la fonction `fiboTerm` et notons a_n et b_n les valeurs prises par les arguments `a` et `b` lorsque l'argument `n` varie de n_0 en décroissant. Lorsque $\texttt{n} = n_0$, on a $[a_{n_0}, b_{n_0}] = [\mathcal{F}_0, \mathcal{F}_1]$. Montrons que,

$$\forall n \in \mathbb{N} \text{ tel que } 1 \leq n < n_0, \quad [a_n, b_n] = [\mathcal{F}_{n_0-n}, \mathcal{F}_{n_0-n+1}]$$

Soit $\mathscr{P}(n)$ l'assertion « $[a_n, b_n] = [\mathcal{F}_{n_0-n}, \mathcal{F}_{n_0-n+1}]$ »,

- **Initialisation**. $\mathscr{P}(n_0)$ est vraie.
- **Hérédité**. Supposons $\mathscr{P}(n)$ vraie, montrons qu'alors $\mathscr{P}(n-1)$ est vraie. Au rang d'appel $n-1$, la fonction `fiboTerm` est appelée avec l'argument a_{n-1} qui vaut
$$a_{n-1} = b_n = \mathcal{F}_{n_0-n+1} = \mathcal{F}_{n_0-(n-1)}$$
et l'argument b_{n-1} qui vaut
$$b_{n-1} = a_n + b_n = \mathcal{F}_{n_0-n} + \mathcal{F}_{n_0-n+1} = \mathcal{F}_{n_0-n+2} = \mathcal{F}_{n_0-(n-1)+1}$$
Où l'on a utilisé la relation de récurrence définissant la suite de Fibonacci,$\mathcal{F}_k + \mathcal{F}_{k+1} = \mathcal{F}_{k+2}$ avec $k = n_0 - n$. Cela prouve donc $\mathscr{P}(n-1)$.
- **Conclusion**. L'itération se poursuit jusqu'à ce que n atteigne la valeur 1. Dans ce cas, la fonction `fiboTerm` s'arrête et renvoie la valeur de b_1. Or, la relation $b_n = \mathcal{F}_{n_0-n+1}$ étant vraie, pour $n = 1$, elle donne $b_1 = \mathcal{F}_{n_0-1+1} = \mathcal{F}_{n_0}$. Ce qui prouve la correction de l'algorithme.