La récursivité

TABLE DES MATIÈRES

I PRINCIPE

1 Définition

Une fonction récursive est une fonction qui peut s'appeler elle-même.
Cela permet de traduire informatiquement l'idée de récurrence en mathématiques.

Exemple

Par exemple, la définition par récurrence de la factorielle est :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Écrire une fonction récursive, c'est écrire :

- Une condition d'arrêt, très importante, correspondant à l'initialisation de la récurrence,
- Un ou plusieurs appels récursifs, correspondant à l'hérédité.

Exemple

Pour la factorielle, cela donne :

```
def fact(n):
    "Renvoie la factorielle de n"""
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

2 Pile de récursion

L'interpréteur gère une pile dite de récursion pour les appels successifs puis les calculs à effectuer.

Exemple

Pour schématiser, l'appel de `fact(4)` correspond à :

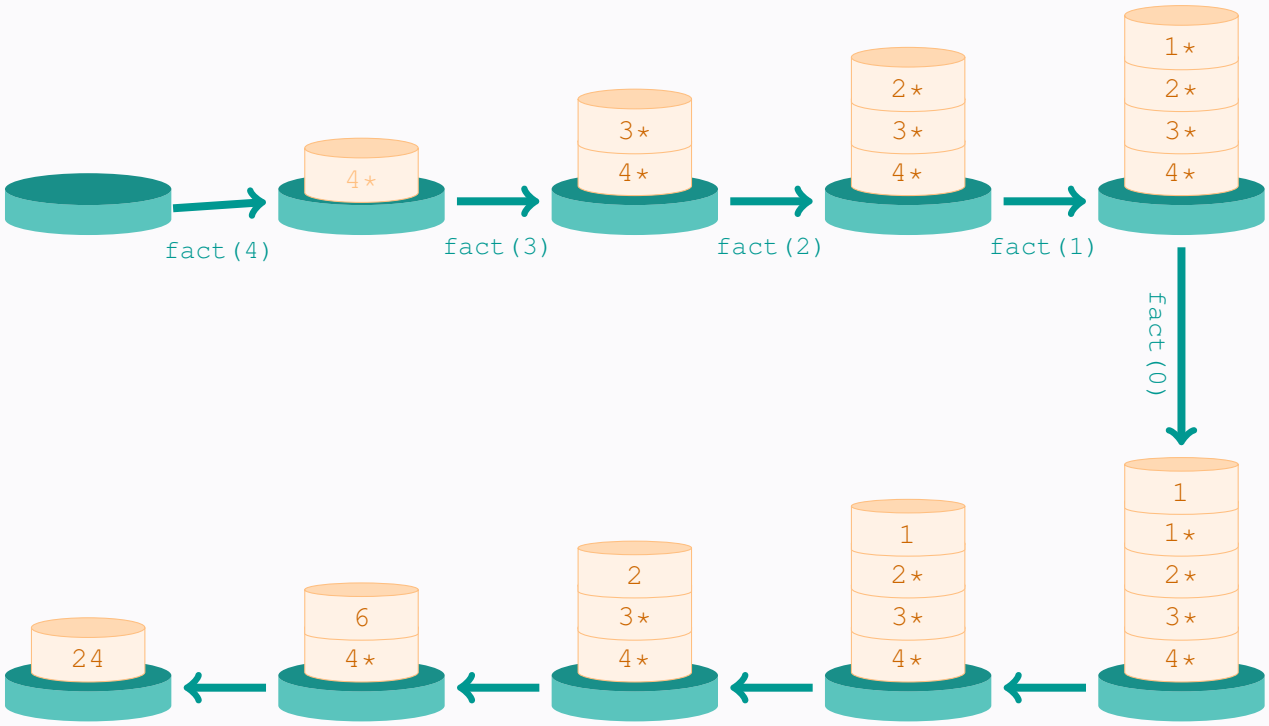


FIGURE 1 – Schéma d'exécution de `fact(4)`

En Python, la pile de récursion est limitée à 1 000 appels environ. Il est possible de modifier cette limite :

```
import sys
sys.setrecursionlimit(nb)
```

3 Complexité, correction, terminaison

- La **complexité spatiale** est donc en général au moins linéaire (par rapport au nombre d'appels récursifs, mais attention à d'éventuelles copies de structures volumineuses à chaque appel).

> **Exemple**
>
> Pour la factorielle, elle est en $O(n)$.

- La **complexité temporelle** s'obtient sous forme de suite récurrente : il n'est pas toujours simple d'en obtenir un ordre de grandeur.

> **Exemple**
>
> $T(n) = \mathrm{O}(n)$ (et même $T(n) = n$.)

- La **correction** se prouve par récurrence.

> **Exemple**
>
> Pour la factorielle, une récurrence rapide donne que `fact(n)` renvoie bien $n!$.

- La **terminaison** peut se prouver soit dans la récurrence précédente, soit en invoquant une suite strictement décroissante d'entiers naturels. C'est là que les cas d'arrêt sont déterminant !

> **Exemple : fonction 91 de MacCarthy**
>
> La terminaison n'est pas toujours aussi évidente à démontrer que pour la factorielle. C'est le cas pour la fonction classique suivante dont les valeurs sont en fait fort simples :

```
def f_91(n):
    if n > 100:
        return n - 10
    else:
        return f_91(f_91(n + 11))
```

4 Avantages, inconvénients

En résumé, concernant la récursivité :

Avantages : facile à écrire (pas de boucle en général), on n'a pas à gérer la pile.

Inconvénients : demande plus de place en mémoire (complexité spatiale), peut retarder l'exécution. Peut aussi avoir des conséquences catastrophiques si mal utilisée (voir ci-après).

II EXEMPLES D'ALGORITHMES RÉCURSIFS, RÉCURSIVITÉ PERFORMANTE

Parfois, on peut éviter de redescendre la pile de récursion s'il n'y a pas de calcul à faire après avoir atteint le cas d'arrêt. Cela s'appelle de la récursivité terminale, et cela a besoin d'être reconnu par l'interpréteur-compilateur.

Malheureusement, ce n'est pas le cas de Python. D'ailleurs, Guido von Rossum, concepteur de Python, dit [1] :

> *I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists, especially those who love Scheme and like to teach programming by starting with a "cons" cell and recursion. But to me, seeing recursion as the basis of everything else is just a nice theoretical approach to fundamental mathematics (...) not a day-to-day tool.*

1 Premier exemple : algorithme d'Euclide

L'algorithme d'Euclide [2] permet de calculer le pgcd de deux entiers naturels :

$$a \wedge b = \begin{cases} a & \text{si } b = 0 \\ b \wedge (a \bmod b) & \text{sinon.} \end{cases}$$

(Remarquons que c'est valable même si $b < a$, car dans ce cas la première étape permet de se ramener à $b \wedge a$)

Cela donne la fonction récursive :

```
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

Ici, le résultat est obtenu pourrait être renvoyé directement, si Python savait le reconnaître (mais ce n'est pas le cas).

On peut se poser la question du nombre de divisions effectuées dans l'algorithme d'Euclide. Un théorème mathématique, le théorème de Lamé, nous apprend que ce nombre d'étapes est majoré par $\left\lfloor \frac{\ln b}{\ln \varphi} \right\rfloor$ si $b \leqslant a$ où $\varphi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or, et est exactement égal à ce nombre lorsque l'on

1. Voir http://neopythonic.blogspot.fr/2009/04/tail-recursion-elimination.html

2. 
Euclide d'Alexandrie (vers -325 - Alexandrie vers -265) est un mathématicien grec. Peu de choses sont connues sur la vie d'Euclide. Il est l'auteur des *Éléments*, traités de géométrie considérés comme l'un des textes fondateurs des mathématiques modernes. Les résultats y sont démontrés avec une rigueur remarquable. Euclide donne des postulats à la base de la géométrie dite euclidienne de nos jours dont le 5ème particulièrement célèbre : par un point passe une et une seule parralèle à une droite fixée. On a longtemps pensé que ce postulat était en fait une conséquence des autres axiomes, jusqu'à ce qu'on construise, au XIXème siècle, une géométrie ne vérifiant pas ce postulat.

caclule le pgcd (égal à 1) de deux termes successifs de la suite de Fibonacci [1] définie par $F_0 = 0$, $F_1 = 1$ et pour tout n, $F_n + F_{n+1} = F_{n+2}$.

2 Deuxième exemple : retour sur la factorielle

Peut-on concevoir une version récursive terminale de la factorielle ? La réponse est oui, à l'aide d'un argument supplémentaire constituant un accumulateur pour y faire directement les calculs au fur et à mesure au lieu de les faire dans la pile de récursion. Cela donne :

```
def fact(n, accu=1):
    if n == 0:
        return accu
    return fact(n - 1, n * accu)
```

Par exemple, pour calculer la factorielle de 4 :

```
fact(4, 1) → fact(3, 4) → fact(2, 12) → fact(1, 24) → fact(0, 24) →  24
```

3 Troisième exemple : exponentiation rapide

L'exponentiation rapide est très facile à implémenter récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair} \\ x \cdot \left(x^{\lfloor n/2 \rfloor}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

```
def expo_rap(x, n):
    "renvoie x^n par exponentiation rapide"
    if n == 0:
        return 1
    y = expo_rap(x, n // 2)
    if n % 2 == 0:
        return y * y
    else:
        return x * y * y
```

1. 

Léonard de Pise, dit Fibonacci (Pise, 1170 - Pise, 1245) est un mathématicien italien. Il introduit en Europe le système décimal et l'écriture des nombres en chiffres arabes. On connaît surtout la suite qui porte son nom, définie par $u_0 = 0$, $u_1 = 1$ et $\forall\, n \in \mathbb{N}^*,\ u_{n+1} = u_n + u_{n-1}$, introduite initialement pour illustrer la reproduction de lapins, et qui est telle que $\frac{u_{n+1}}{u_n}$ converge vers le nombre d'or.

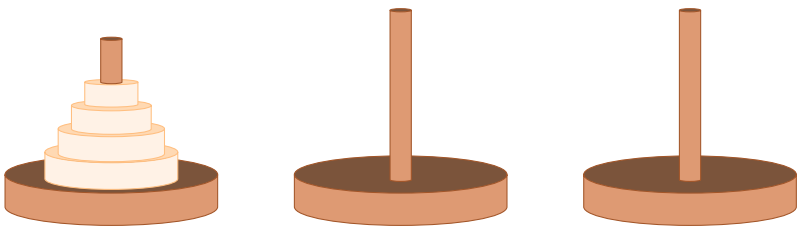
Le nombre d'appels récursifs est le nombre de chiffres en base 2 de n, on a donc une complexité temporelle et spatiale logarithmique : $O(\ln n)$.

Pour la forme, une version récursive terminale s'écrit facilement en changeant la formule (de laquelle on aurait aussi pu partir) :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(x^2\right)^{n/2} & \text{si } n \text{ est pair} \\ x \cdot \left(x^2\right)^{\lfloor n/2 \rfloor} & \text{si } n \text{ est impair} \end{cases}$$

```
def expo_rap(x, n, accu=1):
    "renvoie x^n par exponentiation rapide"
    if n == 0:
        return accu
    if n % 2 != 0:
        accu *= x
    return expo_rap(x * x, n // 2, accu)
```

4 Quatrième exemple : les tours de Hanoï



Le jeu des tours de Hanoï a été inventé par le mathématicien Edouard Lucas [1] au dix-neuvième siècle. Le principe est le suivant : il faut déplacer un pile de n disques d'un emplacement initial à un emplacement final en utilisant un emplacement intermédiaire, en respectant les règles :

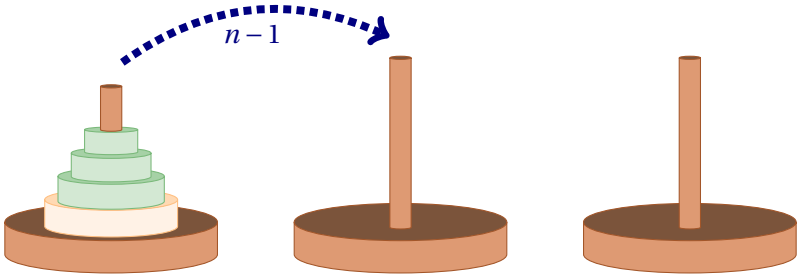
- on ne peut déplacer que le disque en haut d'une pile,
- on ne peut empiler un disque que sur un disque plus grand.

1. 

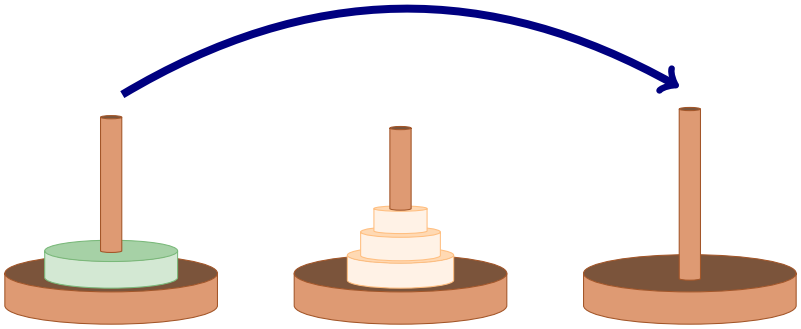
Edouard Lucas (1842 - 1891) est un arithméticien français également connu pour ses récréations mathématiques. Il entre à l'École Normale Supérieure puis devient professeur de Spéciales à Paris. Ses travaux mathématiques concernent la géométrie euclidienne non élémentaire, et surtout la théorie des nombres. Sa principale contribution est celle faite aux tests de primalité. Il a en particulier prouvé que le nombre de Mersenne $2^{127} - 1$ est premier, ce qui reste le plus grand nombre premier découvert sans l'aide d'un ordinateur. Lucas est aussi connu pour être l'inventeur de nombreuses récréations mathématiques. La plus répandue d'entre elles est le problème des tours de Hanoï. Lucas est mort au cours d'un banquet : une assiette portant un couteau est tombée et lui a transpercé la gorge.

C'est un problème facile à résoudre récursivement : il suffit de

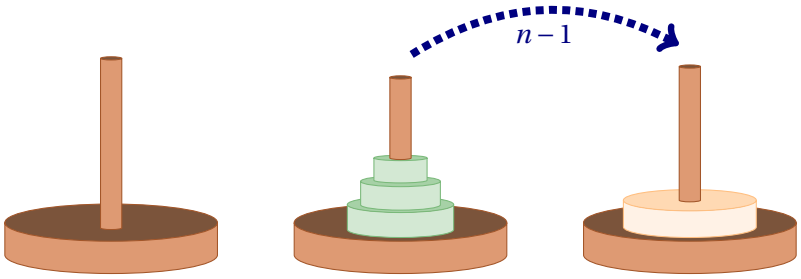
1. déplacer les $n-1$ premiers disques sur la tige intermédiaire,



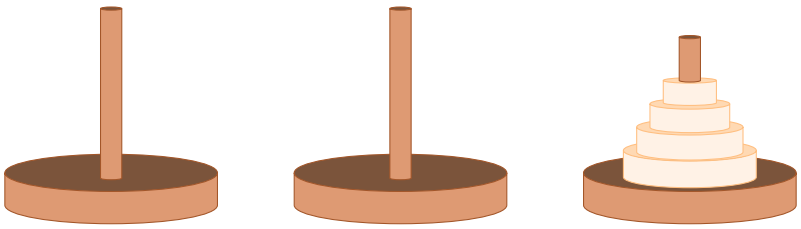
2. puis le plus grand disque sur la tige finale



3. et enfin de nouveau les $n-1$ premiers disques vers la tige finale.



Cela ne viole aucune règle du jeu



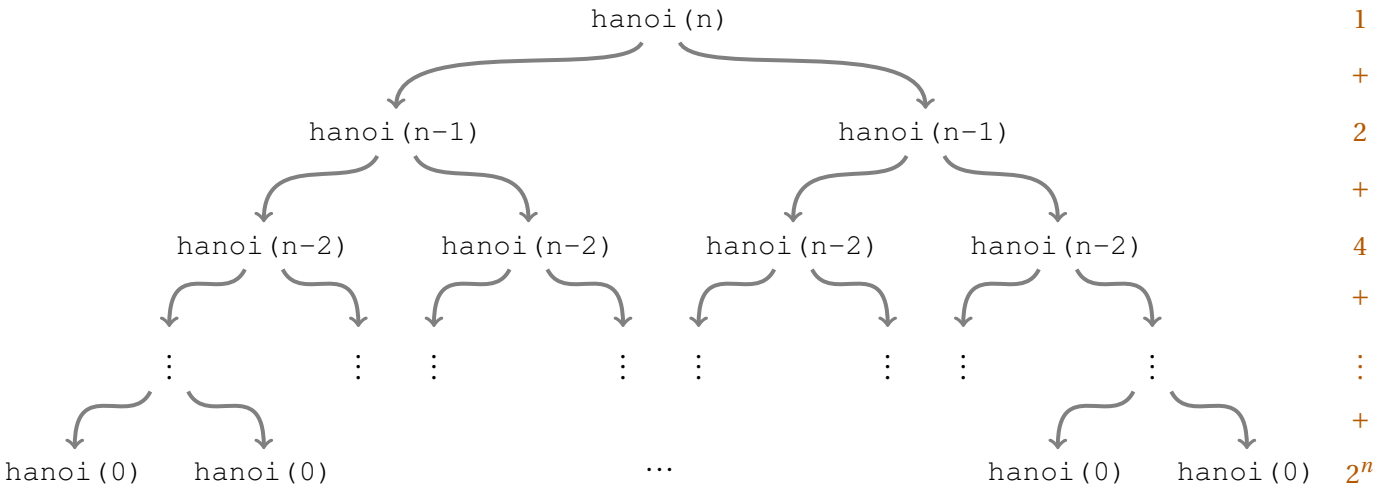
```python
def hanoi(n, start=0, end=2):
    """Affiche les déplacements à effectuer pour résoudre le problème
des tours de Hanoï avec n disques à déplacer depuis l'emplacement
start jusqu'à l'emplacement end. Ces deux emplacements étant
différents parmi 0, 1 et 2."""
    if n != 0: # n = 0 est le cas d'arrêt : rien à faire.

        middle = 3 - start - end # position intermédiaire

        hanoi(n - 1, start, middle)

        print("Déplacer (le disque {}) de {} à {}".format(n, start, end))
        hanoi(n - 1, middle, end)
```

Voilà ce que donne l'exécution de `hanoi(4)` :

```
>>> hanoi(4)
Déplacer (le disque 1) de 0 à 1
Déplacer (le disque 2) de 0 à 2
Déplacer (le disque 1) de 1 à 2
Déplacer (le disque 3) de 0 à 1
Déplacer (le disque 1) de 2 à 0
Déplacer (le disque 2) de 2 à 1
Déplacer (le disque 1) de 0 à 1
Déplacer (le disque 4) de 0 à 2
Déplacer (le disque 1) de 1 à 2
Déplacer (le disque 2) de 1 à 0
Déplacer (le disque 1) de 2 à 0
Déplacer (le disque 3) de 1 à 2
Déplacer (le disque 1) de 0 à 1
Déplacer (le disque 2) de 0 à 2
Déplacer (le disque 1) de 1 à 2
```

Une petite manipulation de piles permettrait même de gérer les trois tiges et de faire un affichage des déplacements (voir le fichier `hanoi.py`).

Ici, il n'y a pas de calculs à faire après la descente, mais il y a un autre problème : l'exécution de `hanoi(n)` peut se schématiser par l'arbre suivant.



avec la relation de récurrence facile : si $T(n)$ est le nombre d'appels à la fonction `hanoi` pour résoudre le problème, alors $T(0) = 1$ et pour tout $n \geqslant 1$, $T(n) = 1 + 2T(n-1)$, ce qui donne

$$T(n) = 1 + 2 + 4 + 8 + \cdots + 2^n = 2^{n+1} - 1.$$

Le nombre de déplacements à effectuer est alors donné par $D(1) = 1$ et pour tout $n \geqslant 2$, $D(n) = 1 + 2D(n-1)$, soit $D(n) = T(n-1) = 2^n - 1$.

On peut vérifier qu'il n'est pas possible de faire mieux : la complexité est exponentielle.

III ATTENTION AUX APPELS MULTIPLES

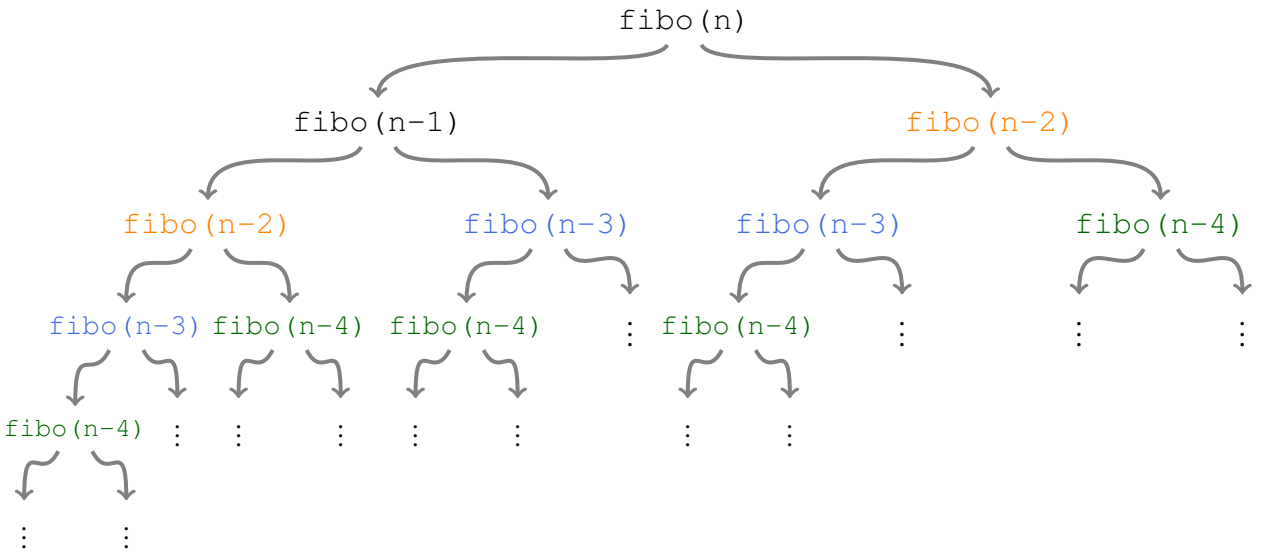
Comme on l'a vu dans l'exemple précédent, la complexité peut vite exploser si on ne prête pas attention à ce qu'on fait lorsque l'on écrit des fonctions récursives.

1 La suite de Fibonacci

Prenons l'exemple de la suite de Fibonacci, définie pr $F_0 = 0$, $F_1 = 1$ et pour tout $n \geqslant 2$, $F_n = F_{n-1} + F_{n-2}$. Si on écrit rapidement une fonction récursive calculant F_n, cela peut donner :

```
def fibo(n):
    if n < 2:
        return n
    return fibo(n - 1) + fibo(n - 2)
```

C'est une très mauvaise idée !



C'est moins volumineux que dans l'arbre précédent, mais un certain nombre de termes vont être calculés de nombreuses fois.

On a cette fois, en comptant les additions $T(0) = T(1) = 0$ et pour tout $n \geqslant 1$, $T(n) = T(n-1) + T(n-2) + 1$. Récurrence du même type que celle qui définit la suite elle-même.

On obtient facilement que pour tout n, $T(n) = F_{n+1} - 1$.

Or l'étude mathématique classique donne $F_n = \dfrac{1}{\sqrt{5}}\varphi^n - \dfrac{1}{\sqrt{5}}\left(\dfrac{-1}{\varphi}\right)^n$ avec $\varphi = \dfrac{1+\sqrt{5}}{2} > 1$ donc $F_n \sim \dfrac{\varphi^n}{\sqrt{5}}$ et $T(n) = O\left(\varphi^n\right)$. La complexité est de nouveau exponentielle (même si $\varphi < 2$).

Pour résoudre ce problème :

- une solution peut être de calculer plutôt les paires (F_n, F_{n-1}) (avec $F_{-1} = 0$).

```python
def fibo2(n):
    "Renvoie (F_n, F_{n-1}) avec F_{-1}=0"
    if n == 0:
        return (0, 0)
    Fnm2, Fnm1 = fibo2(n - 1)
    return (Fnm2 + Fnm1, Fnm1)
```

Cette fois, le coût est linéaire : $T(n) = \mathrm{O}(n)$.

- On peut aussi faire de la mémoïsation : c'est-à-dire garder en mémoire les valeurs calculées successivement pour les réutiliser.

```python
F = [0] # On va avoir F[n] = F_n

def fibo_mem(n):
    "Remplit la liste F des nombres de Fibonacci et renvoie F_n"
    if n == 1:
        F.append(1)
    else:
        fibo_mem(n-1) # On remplit la liste jusqu'à F[n-1]
        F.append(F[-1] + F[-2])
    return F[-1]
```

- On peut faire mieux ! On a aussi que $\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$ et donc pour tout $n \in \mathbb{N}$, $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$. La puissance de la matrice pouvant se calculer par exponentiation rapide, cela donne une complexité logarithmique !

Pour d'autres implémentations, voir le fichier `fibonacci.py`.

2 Cœfficients binomiaux

On peut vouloir calculer $\binom{n}{k}$ en utilisant la formule du triangle de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Cela donne la fonction naïve :

```python
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    return binom(k - 1, n - 1) + binom(k, n - 1)
```

On a alors $T(k,n)=0$ si $k\leqslant 0$ ou $k\geqslant n$ et $T(k,n)=T(k-1,n-1)+T(k,n-1)+1$ sinon.

C'est une récurrence de la même forme que celle de $\binom{n}{k}$, on cherche a et b tels que $T(k,n)=a\binom{n}{k}+b$ et on trouve que $T(k,n)=\binom{n}{k}-1$ pour $0\leqslant k\leqslant n$.

Pour un n donné, la valeur maximale est $T\left(\left\lceil\frac{n}{2}\right\rceil,n\right)$ et la formule de Stirling nous donne du $O\left(\frac{2^n}{\sqrt{n}}\right)$. Pas très réjouissant !

Une meilleure idée serait d'utiliser les formules $\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1}$ et $\binom{n}{k}=\binom{n}{n-k}$:

```
def binom(k, n):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    if n - k < k:
        return binom(n - k, n)
    return (n * binom(k - 1, n - 1)) // k
```

avec une complexité linéaire cette fois.

On peut aussi faire de la mémoïsation et calculer les lignes du triangle de Pascal les unes après les autres (c'est de la programmation dynamique).

IV EXEMPLE DE DÉRÉCURSIFICATION

Notons enfin que toute fonction récursive peut s'écrire de manière itérative en gérant la pile de récursion manuellement. C'est en général facile lorsque la récursivité est terminale, mais ça peut l'être beaucoup moins lorsqu'elle ne l'est pas.

Exemple : Liste de permutations

On souhaite créer une liste des $n!$ permutations des éléments d'une liste à n élément. Notons que la complexité sera nécessairement mauvaise car il faut trouver les $n!$ permutations !

C'est plutôt facile à écrire récursivement :

```
def liste_permut(L):
    """renvoie une liste de listes représentant les permutations
possibles des éléments de L"""
    Lperm = []
    def remplir_permut(perm=[]):
        """construit les permutations débutant par perm et les
stocke dans Lperm"""
        if len(perm) == len(L): # on a bien tous les éléments de L
            Lperm.append(perm) # on stocke la permutation
        for elem in L:
            if elem not in perm:
                # pour chaque élément non utilisé, on crée
                # les permutations débutant par perm + [elem]
                remplir_permut(perm + [elem])
    remplir_permut()
    return Lperm
```

On peut en donner une version dérécursifiée à l'aide d'une pile :

```
def liste_permut(L):
    """renvoie une liste de listes représentant les permutations
possibles des éléments de L"""
    Lperm = []
    pile = [[]] # va contenir les permutations incomplètes
    while pile != []:
        perm = pile.pop()
        if len(perm) == len(L): # permutation complète
            Lperm.append(perm)
        else:
            for elem in L:
                if elem not in perm:
                    pile.append(perm + [elem])
    return Lperm
```

(On pourrait optimiser en évitant le coût linéaire de la recherche dans `perm` en maintenant des tableaux de booléen indiquant si oui ou non un élément a déjà été pris (deuxième argument de la fonction `remplir_permut`.)

V RÉCURSIVITÉ CROISÉE

Il est aussi possible que des fonctions s'appellent récursivement mutuellement.

Exemple

```
def pair(n):
    "teste si n est pair"
    if n == 0:
        return True
    else:
        return impair(n - 1)

def impair(n):
    "teste si n est impair"
    if n == 0:
        return False
    else:
        return pair(n - 1)
```

VI EXERCICES

Pour chacune des fonctions, réfléchir aux complexités, à la terminaison et à la correction.

Exercices

Ex 1 – Écrire une fonction récursive renvoyant le maximum d'une liste.

Ex 2 – Écrire une fonction récursive renvoyant le n^{e} terme de la suite de Syracuse définie par $u_0 \in \mathbb{N}^*$ et pour tout $n \geqslant 0$, $u_{n+1} = \dfrac{u_n}{2}$ si u_n est pair et $3u_n + 1$ sinon.

Ex 3 – Écrire une fonction renvoyant le n^{e} terme de la suite de Héron : $u_0 = 1$ et $u_{n+1} = \dfrac{u_n^2 + 2}{2u_n}$.

Ex 4 – Écrire une fonction récursive `euclide_etendu(a,b)` prenant en entrée deux entiers et renvoyant un triplet (d, u, v) tel que $d = a \wedge b = au + bv$.

Ex 5 – Écrire une version récursive de l'algorithme de Newton.

Ex 6 – Écrire une version récursive de la recherche dichotomique d'un zéro d'une fonction f croissante entre deux bornes a et b.

Ex 7 – Écrire une version récursive de la recherche dichotomique dans un tableau trié (on renvoie la position si l'élément est dedans et -1 sinon).

Ex 8 – Écrire une fonction récursive renvoyant le numéro d'un couple $(x, y) \in \mathbb{N}^2$ avec la numérotation classique par diagonale (voir démonstration de la dénombrabilité de $\mathbb{N}^2$).

Ex 9 – Écrire la version logarithmique du calcul des termes de la suite de Fibonacci (matrices & exponentiation rapide).

Ex 10 – Écrire une fonction récursive qui prend un entier n en entrée et renvoie la liste des permutations de $[\![0, n-1]\!]$. En donner une version dérécursifiée.

On pourra implémenter l'amélioration proposée en V ou utiliser le fait que si σ est une permutation de $[\![1, n+1]\!]$,

- soit $\sigma(n+1) = n+1$ et σ permute les entiers jusqu'à n,
- soit $\sigma(n+1) = j$ et alors $\tau = \sigma \circ (j \;\; n+1)$ vérifie l'hypothèse ci-dessus, et $\sigma = \tau \circ (j \;\; n+1)$.

Les deux cas se résument par $\sigma = \tau \circ (j \;\; n+1)$ pour $0 \leqslant j \leqslant n+1$ et τ permutation des entiers entre 1 et n.

De manière empirique, c'est cette version qui donne les meilleurs résultats (en termes de temps de calcul).