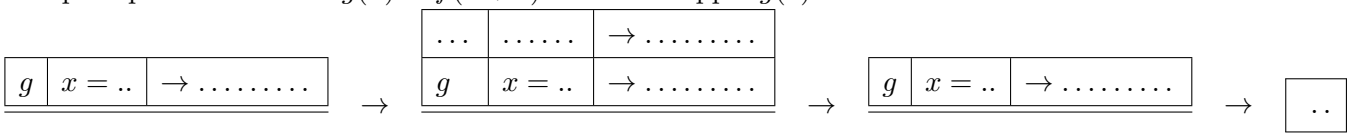
I État de la pile d'exécution

Comme pour tout appel de fonction, l'environnement de la fonction est mis dans la pile d'exécution qui sera vidée à la sortie de la fonction.

expl1 : pour la fonction $f(x) = x^2 + 1$ avec l'appel $f(2)$:

pile d'exécution

expl2 : pour la fonction $g(x) = f(x+1) - 1$ avec l'appel $g(2)$:



pile d'exécution pile d'exécution pile d'exécution

II Définition et exemples

Jusqu'à présent, nous avons vu la programmation itérative, d'autres paradigmes existent comme la programmation récursive. Depuis l'arrivée de langages comme LISP ou algol 60 vers 1 960, la plupart des langages de programmation acceptent la programmation récursive.

Définition 1

Une fonction récursive est une fonction qui ..... ou aura des appels de fonctions qui .....

Ainsi, lors d'appels récursifs, la pile d'exécution grandie et si les appels sont infinis la pile ..... de l'espace mémoire. Pous palier ce problème, le langage `python` instaure une limite pour le nombre de récursion que l'on peut connaître grâce au `getter` suivant :

```
import sys
sys.getrecursionlimit()    # renvoie le nombre d'appels récursifs autorisés
```

Une fonction récursive doit toujours contenir deux choses :

◇ ...

◇ ...

◇ ...

1 Exemple : la factorielle

Pour $n \in \mathbb{N}$, comment calculer la factorielle de n définie par $n! = 1 \times 2 \times \dots n$ et $0! = 1$.

◇ **Version itérative :**

```
def factoI(n : int) -> int :
    f = 1
    for i in range( ... , ... ) :
        f = ...
    return ...
```

◇ **Version récursive :**

Par exemple, $4! = \dots$

$6! = \dots$

a) dans quel cas, connait-on la valeur et la renvoye-t-on directement ?

b) lorsque l'on connaît $n!$, comment calcule-t-on $(n+1)!$?

On peut donc définir $facto(n) = \begin{cases} 1 & \text{si } \dots \\ \dots\dots\dots & \text{sinon} \end{cases}$

```
def factoR(n : int) -> int :
    if ......... :
        return 1
    return .........
```

◇ **État de la pile d'exécution :**

Comment évolue l'état de la pile pour $n = 3$?

$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

.....	...	$\rightarrow \ldots\ldots\ldots$
$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

.....	...	$\rightarrow \ldots\ldots\ldots$
.....	...	$\rightarrow \ldots\ldots\ldots$
$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

.....	...	$\rightarrow \ldots$
.....	...	$\rightarrow \ldots\ldots\ldots$
.....	...	$\rightarrow \ldots\ldots\ldots$
$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

.....	...	$\rightarrow \ldots\ldots\ldots$
.....	...	$\rightarrow \ldots\ldots\ldots$
$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

.....	...	$\rightarrow \ldots\ldots\ldots$
$factoR$	...	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

$factoR$	$n = \ldots$	$\rightarrow \ldots\ldots\ldots$

pile d'exécution

$\rightarrow$

...

2 Exemple : somme des valeurs d'un tableau

Comment faire la somme des valeurs d'un tableau d'entiers ?

◇ **Version itérative :**

```
def somI(tab : list) -> int :
    som = ...
    for ......................
        .........
    return ...
```

◇ **Version récursive :**

Prenons un exemple, `tab=[1, 3, 5, 7]`,
on peut dire que la somme du tableau complet `[1, 3, 5, 7]` est égal à la somme du 1^er élément avec la somme du sous-tableau ..... ;
Puis que la somme du sous-tableau `[3, 5, 7]` est égal à la somme du 1^er élément avec la somme du sous-tableau ..... ;
et ainsi de suite jusqu'à atteindre le tableau vide qui possède une somme .....
On pourra alors remonter les calculs pour connaître la somme finale.

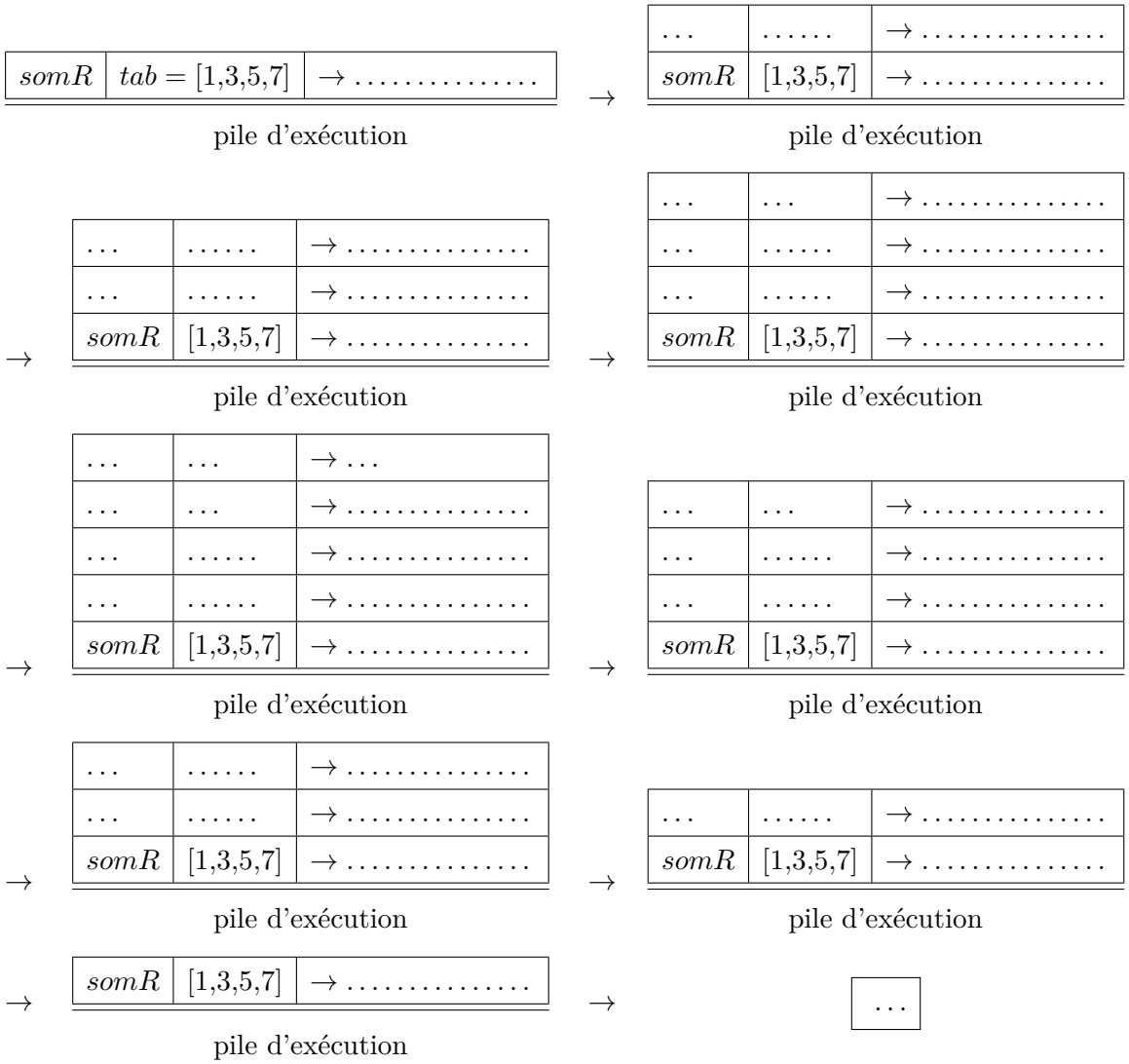
On peut définir $somR(tab) = \begin{cases} \ldots & \text{si le tableau est vide} \\ \ldots\ldots\ldots\ldots\ldots & \text{sinon} \end{cases}$

```
def somR(tab : list) -> int :
    if ...... == 0 :
        return ...
```

```
    else :
        return ..................
```

◇ **État de la pile d'exécution :**

Déroulons les appels pour voir l'état de la pile sur `tab=[1, 3, 5, 7]`.



3 Exemple : suite (u_n)

Considérons la suite (u_n) définie sur $\mathbb{N}$ par $\begin{cases} u_0 = 3 \\ u_{n+1} = 2u_n + 1 \end{cases}$ est définie en fonction de sa valeur précédente.

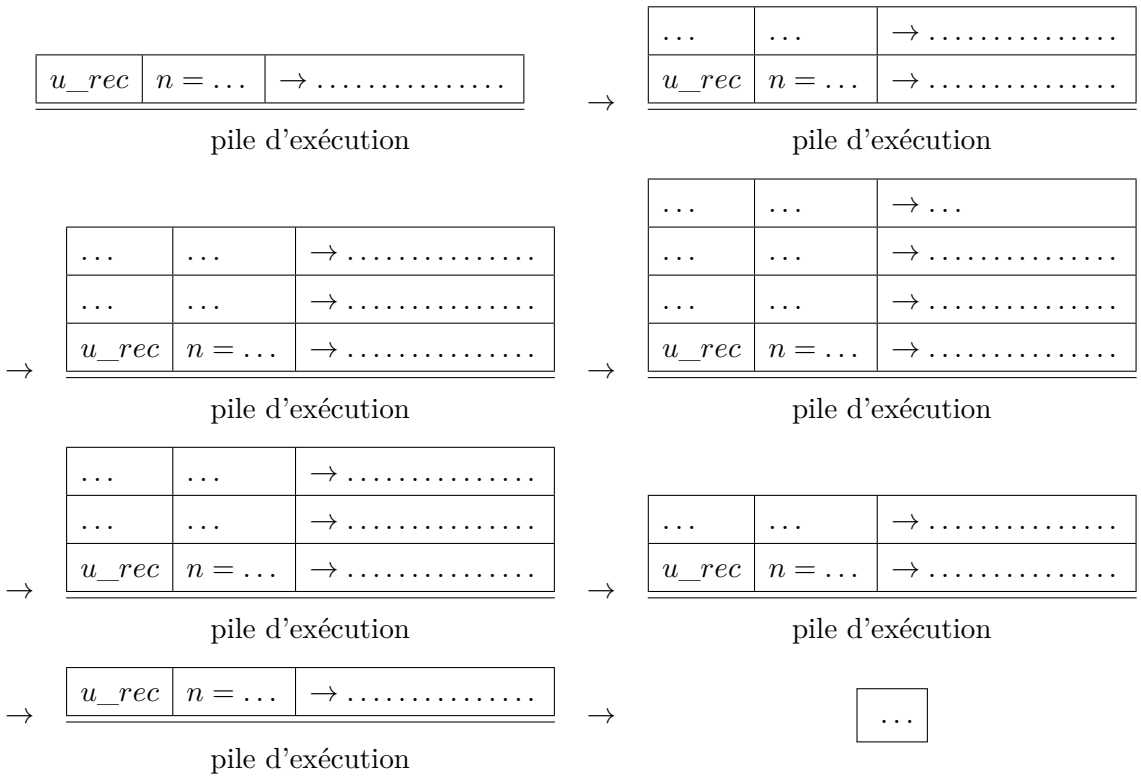
◇ **Version itérative :**

```
def suiteU(n : int) -> int :
    u = ...
    for i in range( ... ) :
        ...
    return ...
```

◇ **Version récursive :**

```
def u_rec(n : int) -> int :
    if ... :
        return ...
    return ..............
```

◇ **État de la pile d'exécution :** pour $n = 3$



4 Exemple : suite de Fibonacci

La suite de Fibonacci (mathématicien italien du XIII^èS), notée f_n, est définie par

$$\begin{cases} f_0 = f_1 = 1 \\ f_{n+2} = f_{n+1} + f_n \quad \forall n \in \mathbb{N} \end{cases}$$

Calculons des valeurs de f_n : $f_2 = \ldots$; $f_3 = \ldots$; $f_4 = \ldots$; $f_5 = \ldots$; $f_6 = \ldots$; $f_7 = \ldots$; $f_8 = \ldots$; etc.

◇ **Version itérative :**

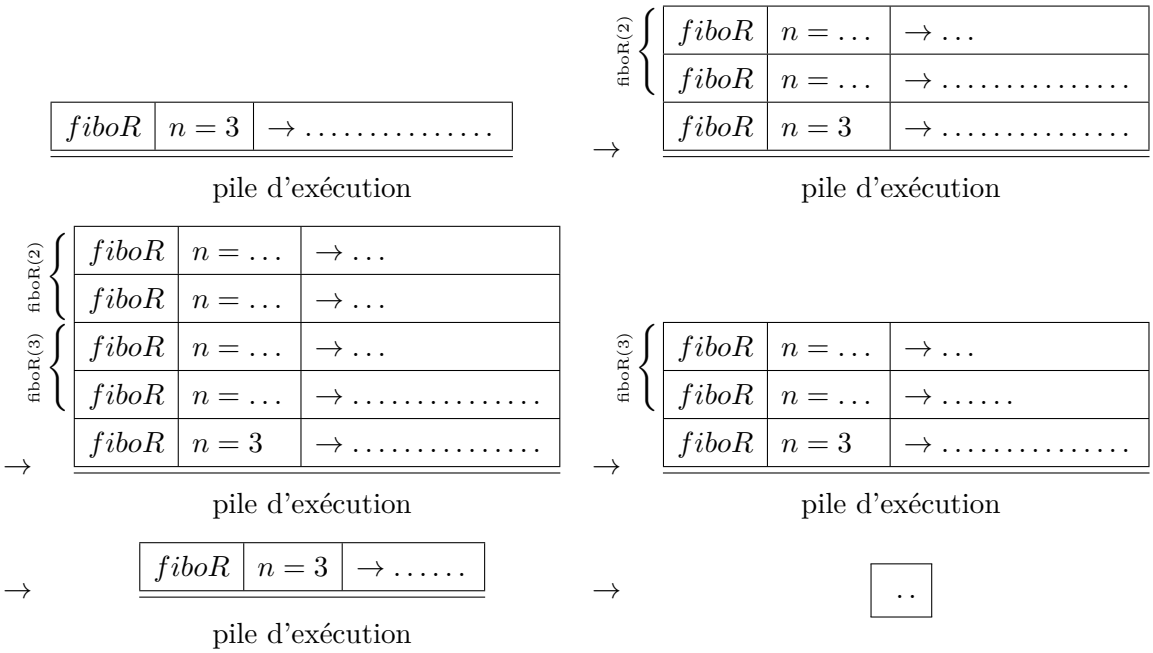
```
def fiboI(n : int) -> int :
    if n == 0 :
        return 1
    f, ff = 1, 1
    for i in range( ..., ... ) :
        tmp = ...
        f = ...
        ff = .........
    return ...
```

◇ **Version récursive :**

```
def fiboR(n : int) -> int :
    if n < 2 :
        return ...
    return ...............
```

◇ **État de la pile d'exécution :** pour $n = 3$

Dans un paradigme de programmation séquentielle, les appels sont généralement évalués de gauche à droite.



La pile semble grandir deux fois plus vite que dans les autres exemples. En vérité, la pile augmentera de l'ordre de φ^n où $\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618$ est le nombre d'or.

Rq : pour ne pas devoir faire un double appel récursif à chaque fois, lorsque $n > 2$, on peut calculer les termes f_{n-2} et f_{n-1} dans les paramètres. On calcule donc la valeur de f_n à la montée de la pile et pas à la descente. On appelle cela un appel récursif avec mémoïsation.

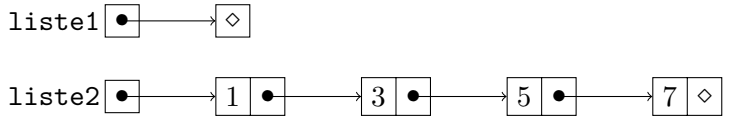
```
def fiboM(n : int, a : int = 1, b : int = 1) -> int :
    if n < 2 :
        return ...
    return fiboM( n-1, ..., ..... )
```

5 Exemple : liste chaînée

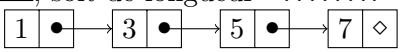
Une liste chaînée est par essence une structure récursive.

Voyons le calcul de la longueur d'une instance `listeChainee`.

Par exemple :



La liste 1 est vide et sa longueur est la longueur de la liste vide ◇, soit de longueur .....

La liste 2 n'est pas vide, et sa longeur est la longueur de la liste 

Cette liste a pour longueur : 1 + la longueur de la sous-liste ...

Cette sous-liste a pour longueur : 1+ la longueur de la 2è sous-liste ...

etc.

On peut définir `_lenR(m : 'Maillon') -> int`, par $\_lenR(m) = \begin{cases} \ldots & \text{si } m \text{ vaut None} \\ \ldots\ldots\ldots\ldots\ldots\ldots & \text{sinon} \end{cases}$

Ce qui donne :

```
def lenR(self) -> int : # appel principal
    return ..............

def _lenR(m : 'Maillon') -> int : # appel secondaire
    if ......... :
        return ...
    else :
        return ............
```

Rq : on peut le voir, certains exemples se prêtent naturellement à des appels récursifs.

III Analyse de fonctions récursives

Lorsque l'on écrit une fonction récursive, on vérifie généralement trois choses :

a) la terminaison : est-ce que la fonction s'arrêtera un jour ? (on utilisera un raisonnement par récurrence avec le .....)

b) la correction : est-ce que la fonction renvoie le bon résultat ? (on utilisera un raisonnement par récurrence sur .....)

c) la complexité spatiale et/ou temporelle : quel est le coût spatial et/ou temporel ?

Par la suite, nous n'étudierons que la complexité temporelle en étudiant une suite (ct_n) qui donne le coût temporel en fonction de n et en considérant qu'un appel de fonction, un test condtionnel, une affectation, un calcul coûtent 1.

1 Exemple : la factorielle

Pour rappel :

```
def factoR(n : int) -> int :
    if n == 0 :
        return 1
    return n * factoR( n-1 )
```

a) le variant est le paramètre n entier naturel qui décroit strictement à chaque appel. Soit l'hypothèse de récurrence $\mathscr{H}_n$ définie $\forall n \in \mathbb{N}$ par $\mathscr{H}_n$: "$factoR(n)$ s'arrête".

◇ Puisque la fonction $factoR(0)$ arrête les appels et renvoie la valeur 0, $\mathscr{H}_0$ est ...

◇ Supposons que $\mathscr{H}_n$ soit vraie pour un certain $n \geqslant 0$,
$factoR(n+1)$ appelle $factoR(\dots)$ puisque ... Par l'hypothèse de récurrence, $factoR(\dots)$ .....
donc $factoR(\dots)$ .....
L'hypothèse $\mathscr{H}_{n+1}$ est donc .....

◇ donc $\mathscr{H}_n$ est ...

b) l'invariant est l'hypothèse de récurrence $\mathscr{H}_n$ définie $\forall n \in \mathbb{N}$ par $\mathscr{H}_n$: "$factoR(n)$ renvoie $n!$".

◇ Puisque la fonction $factoR(0)$ renvoie la valeur ....., $\mathscr{H}_0$ est ...

◇ Supposons que $\mathscr{H}_n$ soit vraie pour $n \geqslant 0$,
$factoR(n+1)$ renvoie ..... puisque .....
..... selon .....
.....
L'hypothèse $\mathscr{H}_{...}$ est donc vérifiée.

◇ donc $\mathscr{H}_n$ est ...

c) On définit pour $n \in \mathbb{N}$, $ct_n = \begin{cases} \dots & \text{si n=0} \\ \dots\dots & \text{sinon} \end{cases}$

Il s'agit d'une suite arithmétique de raison 4, donc $ct_n =$ ..... Le coût temporel est ...

2 Exemple : somme des valeurs d'un tableau

Pour rappel :

```
def somR(tab : list) -> int :
    if len(tab) <= 0 :
        return 0
    else :
        return tab[0] + somR( tab[1:] )
```

Ici, on définit $n = len(tab)$ qui est un entier naturel.

a) le variant est le paramètre $n = len(tab)$ entier naturel qui ..... à chaque appel.
Soit l'hypothèse de récurrence $\mathscr{H}_n$ définie $\forall n = len(tab) \in \mathbb{N}$ par $\mathscr{H}_n$: "$somR(tab)$ s'arrête".

◇ Puisque la fonction $somR([])$ arrête les appels et renvoie la valeur 0, $\mathscr{H}_0$ est ...

◇ Supposons que $\mathscr{H}_n$ soit vraie pour $n = len(tab) \geqslant 0$,
$somR([v] + tab)$ appelle $somR(.....)$ puisque $len([v] + tab) = ...$
Par l'hypothèse de récurrence, $somR(...)$ s'arrête donc $somR(.....)$ s'arrête.
L'hypothèse $\mathscr{H}_{n+1}$ est donc .....

◇ donc $\mathscr{H}_n$ est ...

b) l'invariant est l'hypothèse de récurrence $\mathscr{H}_n$ définie $\forall n = len(tab) \in \mathbb{N}$ par $\mathscr{H}_n$: "$somR(tab)$ renvoie la somme des valeurs de tab".

◇ Puisque la fonction $somR([])$ renvoie la valeur ..., $\mathscr{H}_0$ est ...

◇ Supposons que $\mathscr{H}_n$ soit vraie pour $n = len(tab) \geqslant 0$,

$somR([v] + tab)$ renvoie ..... puisque ...
v +..... selon ...
la somme des valeurs du tableau .....

L'hypothèse $\mathscr{H}_{...}$ est donc vérifiée.

◇ donc $\mathscr{H}_n$ est ...

c) On définit pour $n \in \mathbb{N}$, $ct_n = \begin{cases} ... & \text{si n=0} \\ ..... & \text{sinon} \end{cases}$

Il s'agit d'une suite arithmétique de raison 7, donc $ct_n = ...$ Le coût temporel est ...
Rq : on a supposé, à tort en `python`, que `tab[1:]` ne faisait pas de copie de tableau.

3 Récapitulatif de la complexité temporelle

structure récursive	**relation de récurrence**	**coût temporel**	**durée (n=100)**	**durée (n=10 000)**
def f(n) **:** **return** f(n//2)	$ct_n = 1 + ct_{\lfloor \frac{n}{2} \rfloor}$	$\mathcal{O}(\ln(n))$	$\sim$ 66 ns	$\sim$ 133 ns
def f(n) **:** **return** f(n-1)	$ct_n = ...$	...	$\sim$ 1 μs	$\sim$ 100 μs

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n//2)
	$ct_n = ...$			

def f(n) **:** **return** f(n//2)+f(n//2)	$ct_n = ...$			

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n//2)+f(n//2)
	$ct_n = ...$	...	$\sim$ 6,6 μs	$\sim$ 1 329 μs

def f(n) **:**
  for i **in range(n) :**
  a=i
  return f(n-1)
	$ct_n = ...$	...	$\sim$ 100 μs	$\sim$ 1 s

def f(n) **:** **return** f(n-1)+f(n-1)	$ct_n = ...$	...	$\sim 10^{15}$ans (!)	...

Rq : pour les ordres de grandeurs des durées, on a considéré qu'une opération prenait 10 ns.
Rq : l'univers serait âgé de $\sim 10^{10}$ années et il contiendrait un total de $\sim 10^{80}$ atomes.