TP (S1) 3 Récursivité

Objectifs
- Introduction aux fonctions récursives.
- Exemples de problèmes résolus par récursivité.

1. PREMIER EXEMPLE ET GÉNÉRALITÉS

1.1. Le principe récursif

Pour présenter la notion de récursivité, commençons par l'exemple classique du calcul de la factorielle d'un entier naturel n. On peut définir l'entier $n!$ par

$$n! = 1 \times 2 \times \cdots \times n$$

en ajoutant que $0! = 1$ par convention. Par exemple $4! = 1 \times 2 \times 3 \times 4 = 24$. Cette définition conduit naturellement à concevoir le calcul de la factorielle en utilisant une boucle `for` :

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n. """
    f = 1
    for k in range(1,n+1) :
        f *= k
    return f
```

Notez que pour $n = 0$ le résultat est correct (puisque `range(1,n+1)` est vide, la valeur de f initialisée à 1 n'est donc pas modifiée).

Une autre manière de présenter la définition de la $n!$ est d'écrire :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n \in \mathbb{N}^* \end{cases}$$

Notons que cette définition de la factorielle fait appel à la factorielle elle-même (mais sur une valeur plus petite), ce qui est caractéristique du principe **récursif**. Le calcul de 4! se fait alors ainsi :

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 24 \end{aligned}$$

Notons que pour que le calcul ne boucle pas infiniment sur lui-même, il est indispensable qu'au bout d'un certain nombre d'étapes il aboutisse à une valeur de factorielle que l'on sait calculer directement. Une telle situation est appelée un **cas terminal**. Pour notre exemple de la factorielle, c'est la valeur $0! = 1$ qui joue le rôle de cas terminal. À noter aussi que si on lance le calcul sur une valeur strictement négative, celui-ci bouclera sans s'arrêter. Par exemple :

$$(-1)! = (-1) \times (-2)! = (-1) \times ((-2) \times (-3)!) = \ldots$$

Il se trouve que cette définition récursive peut se transposer telle qu'elle pour écrire une nouvelle version de la fonction `factorielle` :

```
def factorielle(n : int) -> int :
    """ renvoie la factorielle de l'entier naturel n """
    if n == 0:
        return 1
    else :
        return n*factorielle(n-1)
```

Tapez cette fonction et expérimentez-la sur plusieurs valeurs de n.

Il est probable que, si vous découvrez la notion de récursivité, vous soyez surpris qu'un tel programme fonctionne, sans que vous n'ayez écrit dans son code la

moindre boucle. Pour éclairer ce phénomène, il faut comprendre que lorsqu'un appel à une fonction se fait à l'intérieur d'une autre fonction, cette dernière interrompt son exécution (en sauvegardant tout le contexte qui lui permettra de reprendre le calcul là où il en était) pour permettre l'exécution de la fonction appelée. Le mécanisme est identique si la fonction appelée est la même que la fonction appelante. On peut présenter les appels successifs provoqués par `factorielle(3)` de la manière suivante :

- l'appel `factorielle(3)` commence et s'interrompt en déclenchant l'appel de `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` commence et s'interrompt en déclenchant l'appel de `factorielle(1)` ;
    - — l'appel `factorielle(1)` commence et s'interrompt en déclenchant l'appel de `factorielle(0)` ;
      - — l'appel `factorielle(0)` va jusqu'à son terme et renvoie la valeur 1 à la fonction `factorielle(1)` qui l'a appelée ;
    - — l'appel `factorielle(1)` reprend, termine son calcul $1 \times 1 = 1$, et renvoie cette valeur à l'appel `factorielle(2)` ;
  - ◇ l'appel `factorielle(2)` reprend, termine son calcul $2 \times 1 = 2$, et renvoie cette valeur à l'appel `factorielle(3)` ;
- l'appel `factorielle(3)` reprend, termine son calcul $3 \times 2 = 6$, et renvoie cette valeur, qui est l'unique valeur de retour de l'appel initial.

Le mécanisme consistant à stopper le déroulement d'un appel pour en permettre un autre oblige l'ordinateur à consommer de la mémoire pour stocker son contexte. Afin de contrôler cette dépense en mémoire, le langage Python limite par défaut le nombre d'appels récursifs à environ 1000. Au delà, une erreur est générée :

```
RecursionError: maximum recursion depth exceeded
```

Testez ce phénomène en appelant `factorielle(988)` puis `factorielle(989)`. Cette limite est généralement largement assez haute pour permettre l'exécution des exemples que nous rencontrerons, mais sachez qu'il est possible de la modifier par la commande suivante (qui positionne le nombre maximal d'appels ici à 2000) :

```
import sys
sys.setrecursionlimit(2000)
```

Relancez alors le calcul de `factorielle(989)`.

La notion de récursivité est généralement assez déstabilisante au départ, mais vous constaterez à l'usage que le principe devient assez rapidement familier, de très nombreux concepts pouvant être exprimés récursivement. Comme vous allez le constater, elle dépasse d'ailleurs très largement le champs des exemples mathématiques par lesquels nous commençons.

Exercice 1 [Sol 1]

1) Écrivez une fonction récursive `puissance(x:float, n:int) -> float` renvoyant x^n pour n entier naturel, en exprimant la définition sous la forme :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{si } n \in \mathbb{N}^*. \end{cases}$$

2) Dans l'exemple précédent, comme sur celui de la factorielle, l'appel récursif se fait en diminuant la variable n de 1. Il est parfois possible de la réduire d'avantage (pour minimiser le nombre d'appels nécessaires), et souvent de la diviser par 2, en utilisant un principe **dichotomique**. Reprendre la fonction précédente avec la définition suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^2)^{\frac{n}{2}} & \text{si } n \in \mathbb{N}^* \text{ est pair} \\ x \times (x^2)^{\frac{n-1}{2}} & \text{si } n \in \mathbb{N}^* \text{ est impair.} \end{cases}$$

On utilisera le quotient de la division euclidienne de n par 2 (qui s'obtient par `n//2`), et qui est égal à $\frac{n}{2}$ si n est pair et à $\frac{n-1}{2}$ si n est impair.
Afin de bien vous approprier leurs fonctionnements, listez à la main les appels successifs engendrés par `puissance(2,10)` pour chacune des deux versions.

3) Il arrive que l'on puisse réduire une variable autrement qu'en la diminuant de 1 ou en la divisant par 2.
Par exemple, écrivons une fonction `enBase(b : int, n : int) -> list` renvoyant, sous la forme d'une liste, l'écriture de l'entier naturel n en base b (l'entier b est supposé supérieur ou égal à 2). On rappelle qu'une écriture en base b s'exprime sous la forme de la liste

$$[c_k, c_{k-1}, \dots, c_2, c_1, c_0]$$

où les chiffres $c_0, \dots, c_k$ sont des entiers compris entre 0 et $b-1$, et représente l'entier

$$n = c_0 + c_1 \times b + c_2 \times b^2 + \cdots + c_{k-1} \times b^{k-1} + c_k \times b^k.$$

Par exemple la liste $[1,0,0,1,1]$ est l'écriture en base 2 de l'entier

$$1 + 1 \times 2 + 0 \times 4 + 0 \times 8 + 1 \times 16 = 19.$$

Pour obtenir l'écriture en base b d'un entier naturel n, on utilise le fait que le « chiffre des unités » c_0 est le reste de la division euclidienne de n par b, et que le quotient de cette division euclidienne a pour écriture $[c_k, c_{k-1}, \dots, c_2, c_1]$ en base b. On peut donc adopter la définition récursive suivante pour l'écriture en base b d'un entier naturel n (en distinguant le cas terminal correspondant à un nombre d'un seul chiffre en base b) :

$$\begin{cases} [n] & \text{si } n < b \\ \text{insérer } n\%b \text{ à la fin de la liste } \texttt{enBase(b,n//b)} & \text{si } n \geqslant b \end{cases}$$

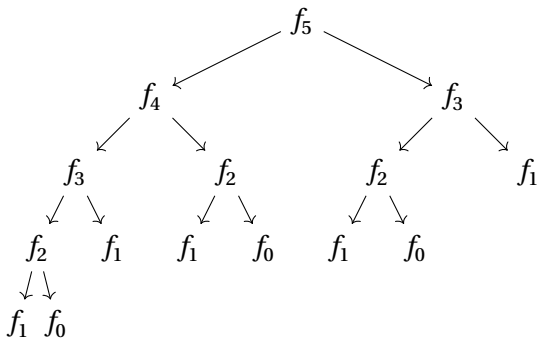
Écrivez le code de la fonction récursive `enBase`. Listez à la main les appels successifs engendrés par `enBase(2,19)`.

BILAN. Une fonction **récursive** est une fonction qui s'appelle elle-même dans sa définition, sur des valeurs « réduites » de ses paramètres, afin d'aboutir de proche en proche à un (ou plusieurs) **cas terminal** qui dont la valeur est renvoyée directement. Une erreur classique dans l'écriture des fonction récursive consiste à mal évaluer ce cas terminal et que les appels bouclent indéfiniment (en pratique, il seront limités en Python par le nombre d'appels imbriqués autorisés).

MISE EN GARDE. Si à l'usage vous trouverez les fonctions récursives de plus en plus naturelles et faciles à écrire, une approche trop naïve peut conduire à des programme extrêmement peu efficaces. L'exemple le plus classique est celui d'une fonction `fibonacci(n : int) -> int` renvoyant le terme d'indice n de la suite de fibonacci définie comme vous le savez par $f_0 = f_1 = 1$ et pour tout $n \geqslant 2$, $f_n = f_{n-1} + f_{n-2}$. Il est tentant de proposer la fonction récursive suivante :

```
def fibonacci(n : int) -> int :
    """ renvoie le terme d'indice n de la suite de fibonacci """
    if 0 <= n <= 1 :
        return(1)
    else :
        return(fibonacci(n-1)+fibonacci(n-2))
```

Si ce code renvoie bien la valeur de f_n, il le fait de manière extrêmement maladroite puisque les deux appels récursifs générés à chaque étape conduisent à recalculer plusieurs fois de nombreux termes de la suite, comme le montre l'exemple de l'appel `fibonacci(5)` (on dispose à chaque fois sur la ligne d'en-dessous les appels générés par ceux de l'étape précédente, et pour plus de concision f_n est écrit au lieu de `fibonacci(n)`) :



Vous constatez que f_1 par exemple a été évalué 5 fois !

Notez qu'il est possible de construire une version (toujours récursive) plus efficace de la fonction `fibonacci` contournant ce problème en utilisant un principe dit de **programmation dynamique** que nous présenterons au second semestre.

1.2. D'autres exemples moins mathématiques

Exercice 2 [Sol 2]

1) Que va produire l'appel de `etoiles1(5)` pour la fonction récursive suivante ? (Essayez de le deviner d'abord sans taper la fonction, puis faites-le pour contrôler.)

```
def etoiles1(n : int) -> None :
    if n > 0:
        print('*'*n)
        etoiles1(n-1)
```

2) Même question pour l'appel de `etoiles2(5)` :

```
def etoiles2(n : int) -> None :
    if n > 0:
        etoiles2(n-1)
        print('*'*n)
```

Exercice 3 [Sol 3] Cet exercice est plus délicat.

1) Si `L` est une liste, on appelle *sous-liste* de `L` toute liste obtenue en enlevant un nombre quelconque d'éléments de `L` (et en conservant les éléments restants dans le même ordre). Par exemple, les listes suivantes sont toutes des sous-listes de

la liste L=[1,2,3,4,5,6] : [1,3,5,6], [2,6], [], [4,5]. Écrivez une fonction récursive `sous_listes(L : list) -> list` renvoyant la liste de toutes les sous-listes de L. (Notez que le résultat sera donc une liste de listes.) Par exemple l'appel `sous-listes([1,2,3])` doit renvoyer (à l'ordre des sous-listes près) : [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]].

Indication : On pourra appeler récursivement la fonction sur la liste amputée de son dernier élément, puis conserver ces sous-listes et leur ajouter toutes celles obtenues en leur ajoutant le dernier élément de la liste initiale. Quel est le cas d'arrêt ?

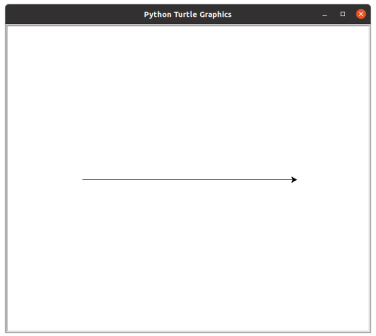
2) Si L est une liste, on appelle *permutation* de L toute liste obtenue en modifiant l'ordre des éléments de L. Par exemple, les listes suivantes sont toutes les permutations possibles de la liste L = [1,2,3] : [1,2,3][1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1].

Écrivez une fonction récursive `permutations(L : list) -> list` renvoyant la liste de toutes les permutations de L.

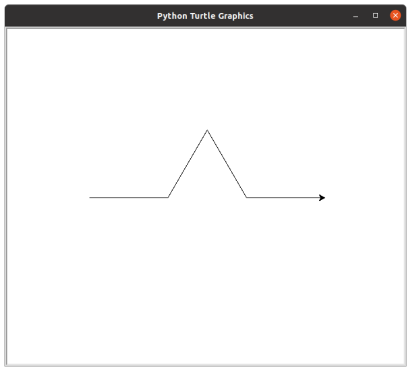
DESSIN DE FRACTALE. Une initiation à la récursivité ne saurait être complète sans un tracé de courbe *fractale*. Intéressons-nous à la célébrissime courbe de KOCH.

Elle consiste :

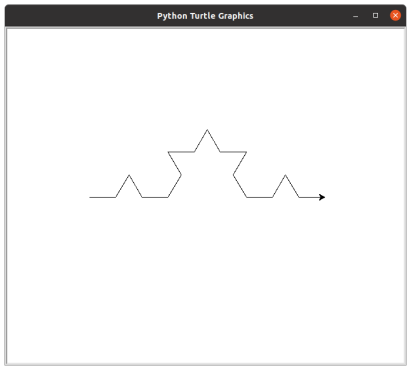
- à la profondeur 0 à tracer un simple segment d'une longueur ℓ donnée (notez que le petit triangle au bout du segment est dû au module `turtle` que nous allons utiliser pour effectuer le tracé) :



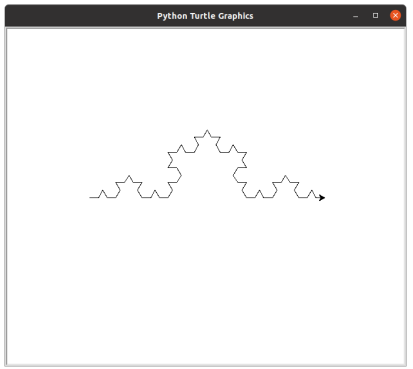
- à la profondeur 1, on remplace le segment précédent par la ligne brisée suivante (chacun des segments de cette ligne a pour longueur le tiers de celui du segment initial) :



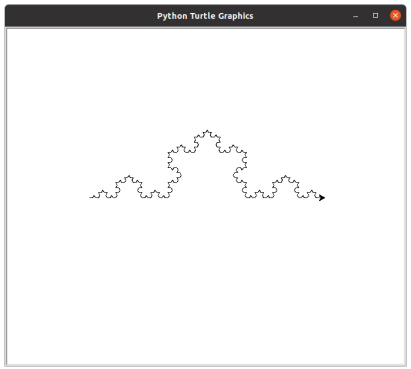
- à la profondeur 2, chacun des segments de la figure précédente est elle-même remplacée par une figure similaire à l'échelle correspondante et orientée dans sa direction :



- et ainsi de suite, à la profondeur 3 :



- puis 4 :



Exercice 4 [Sol 4] Vous trouverez sur l'espace partagé un fichier **koch.py** contenant les instructions utilisant le module `turtle` pour tracer le figure en profondeur 1 pour une longueur ℓ positionnée ici à 400.

Écrire une fonction récursive `koch(l : int, p : int) -> None` effectuant le tracé pour une longueur ℓ et à une profondeur p.

2. DEUX GRANDS CLASSIQUES

2.1. L'algorithme de HORNER

Soit $n \in \mathbb{N}$ et soit p une fonction polynômiale de degré n, $p : x \mapsto a_n x^n + \ldots + a_1 x + a_0$. Un telle fonction sera représentée en Python par la liste P de taille $n+1$ de coefficients rangés par degrés décroissants : P = $[a_n, a_1, a_0]$. Par exemple :

- $x \mapsto 2x^2 + x - 1$ sera représenté par la liste `[2,1,-1]`,
- $x \mapsto 2x^3 + x^2 - x$ sera représenté par la liste `[2,1,-1,0]`.

Le but de l'exercice suivant et de comparer le nombre de multiplications effectuées par différentes fonctions permettant l'évaluation de p en un réel x (c'est-À-dire. le calcul de $p(x)$).

Exercice 5 [Sol 5]

1. On propose les deux fonctions suivantes :

```
def eval_pol1(P:list,x:float)->float:
    n, res = len(P)-1, 0
    for k in range(n+1):
        res = res + P[n-k]*x**k
    return res

def eval_pol2(P:list,x:float)->float:
    n, res = len(P)-1, 0
    X = 1
    for k in range(n+1):
        res = res + P[n-k]*X
        X = X*x
    return res
```

Vérifiez que ces deux fonctions renvoient le résultat attendu puis explicitez u_n, *resp.* v_n, le nombre de multiplications effectuée par `eval_pol1`, *resp.* `eval_pol2`, pour évaluer une fonction polynomiale de degré n.

2. L'algorithme de HORNER repose sur le principe suivant :
   - si p est constante, le calcul de $p(x)$ est immédiat,
   - sinon, il suffit de remarquer que :

$$p(x) = \left(\left(a_n x^{n-1} + \ldots + a_2\right) \times x + a_1\right) \times x + a_0.$$

2.1) Écrire une fonction récursive `eval_polH_rec(P:list,x:float)->float` permettant le calcul de $p(x)$ via l'algorithme de HORNER.

2.2) Explicitez le nombre w_n de multiplications effectuées par `eval_polH_rec` pour évaluer une fonction polynomiale de degré n et comparez aux fonctions précédentes.

2.3) Proposez une version non récursive de l'algorithme de HORNER.

2.2. Les tours de HANOÏ

Il y a des problèmes pour lesquels il peut s'avérer utile de raisonner récursivement pour leur résolution. En voici un, connu sous le nom des *tours de HANOÏ*.

On dispose de trois tours et d'une pile de n disques placée par exemple sur la Tour 1. Le but du jeu est de déplacer cette pile sur une des deux autres tours en respectant les règles suivantes :

- *On ne peut déplacer qu'un disque à la fois.*
- *On peut poser un disque sur une tour vide, ou sur un disque de diamètre supérieur uniquement.*

L'image animée *Hanoi.gif* dans le dossier partagé permet de visualiser l'animation.

On voudrait écrire une fonction :

```
Hanoi(n:int, Tdepart:int, Tarrivee:int, Taux:int)->None
```

qui affiche à l'écran les déplacements à effectuer sous la forme `i -> j` où i et j sont deux numéros de tours, la pile de n disques étant initialement sur `Tdepart`, et devant se retrouver sur `Tarrivee` à la fin.

Exercice 6 Tours de HANOÏ : penser récursif! [Sol 6]

1) Compléter l'analyse suivante en s'assurant à chaque étape que les règles sont bien respectées :

- Si $n = 1$: il y a juste à afficher ....
- Si $n > 1$:
  1. pour pouvoir déplacer le plus grand disque, celui-ci doit être libéré, pour cela il faut déplacer la pile des $n-1$ ... de ... vers ...
  2. on peut alors déplacer le plus grand disque de ... vers ... (les règles sont respectées car ...),
  3. il faut déplacer la pile des $n-1$ ... de ... vers ... (en respectant les règles, ce qui est possible car ...).

2) Écrire la fonction récursive `Hanoi(n, Tdepart, Tarrivee, Taux)` et la tester avec de petites valeurs de n. On peut vérifier ses solutions à cette adresse http://championmath.free.fr/tourhanoi.htm.

Nous allons maintenant compter combien de déplacements de disques sont effectués au total dans cette fonction.

Exercice 7 Complexité de la fonction HANOÏ [Sol 7]

On note d_n le nombre de déplacements de disques effectués lorsque la pile comporte n disques.

1) Déterminer d_1.

2) Montrer que pour $n > 1$, $d_n = 2d_{n-1} + 1$.

3) En déduire que la suite $(1 + d_n)_{n \in \mathbb{N}^*}$ est géométrique et déterminer l'expression de d_n.

À retenir

- On appelle fonction récursive toute fonction qui fait appel à elle-même mais avec des paramètres « plus simples » (comme la récurrence en mathématiques).
- L'exécution d'une fonction récursive entraîne une succession d'empilements et dépilements de données dans la mémoire (dans une pile). Il peut y avoir un débordement en cas d'appels récursifs trop nombreux.
- Avant d'écrire une fonction sous forme récursive il faut veiller à ce qu'un même calcul ne soit pas fait plusieurs fois, sinon on perd en efficacité. Par exemple, programmer la suite de Fibonacci sous forme récursive est inefficace.
- Lors de l'écriture d'une fonction récursive, il faut s'assurer de l'existence d'un cas terminal (qui ne donne aucun appel récursif), sinon on a une boucle infinie.

SOLUTIONS DES EXERCICES

1)

```
def puissance(x : float, n : int) -> float :
    """ renvoie x^n pour n entier naturel. """
    if n==0 :
        return(1)
    else :
        return(x*puissance(x,n-1))
```

`puissance(2,10)` engendre les appels :

Solution 1

`puissance(2,9)`→`puissance(2,8)`→...→`puissance(2,1)`→`puissance(2,0)`

2)
```
def puissance(x : float, n : int) -> float :
    """ renvoie x^n pour n entier naturel. """
    if n == 0:
        return(1)
    elif n%2 == 0:
        return(puissance(x**2,n//2))
    else :
        return(x*puissance(x**2,n//2))
```
puissance(2,10) engendre les appels :

puissance(2,5)→puissance(2,2)→puissance(2,1)→puissance(2,0)

3)
```
def enBase(b : int, n : int) -> list :
    """ renvoie l'écriture de n en base b """
    if n < b:
        return([n])
    else :
        L = enBase(b,n//b)
        L.append(n%b)
        return(L)
```
enBase(2,19) engendre les appels : enBase(2,9)→ enBase(2,4) → enBase(2,2) → enBase(2,1).

Solution 2

1)
```
>>> etoiles1(5)
*****
****
***
**
*
```

2)
```
>>> etoiles2(5)
*
**
***
****
*****
```

Solution 3

1)
```
from copy import deepcopy
def sous_listes(L : list) -> list :
    """ renvoie la liste des sous-listes de L """
    if L == []:
        return([[]])
    else :
        d = L[len(L)-1]
        L1 = sous_listes(L[:len(L)-1])
```

```
        L2 = deepcopy(L1)
        for l in L2 :
            l.append(d)
        return(L1+L2)
```

2) L'idée mise en œuvre est d'appeler récursivement la fonction sur la liste amputée de son élément d'indice i et d'ajouter à chaque permutation cet élément, et de regrouper ces listes obtenues pour tous les indices i de la liste initiale.

```
def permutations(L : list) -> list :
    """ renvoie la liste des permutations de L """
    if L == []:
        return([[]])
    else :
        P = []
        for i in range(len(L)) :
            Temp = permutations(L[:i]+L[i+1:])
            for l in Temp :
                l.append(L[i])
            P += Temp
        return(P)
```

Solution 4

```
def koch(l : int, p : int) -> None :
    """ Trace une courbe de \textsc{Koch} de longeur l
    et de profondeur p """
    if p == 0:
        forward(l)
    else :
        koch(l/3,p-1)
        left(60)
        koch(l/3,p-1)
        right(120)
        koch(l/3,p-1)
        left(60)
        koch(l/3,p-1)
resetscreen()
up()
setposition(-200,0)
down()
koch(400,4)
```

Solution 5

1. On a immédiatement :

$$u_n = \sum_{k=0}^{n} (k-1+1) = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

$$v_n = \sum_{k=0}^{n} 2 = 2(n+1)$$

La deuxième fonction est donc bien plus rapide que la première.

2. **2.1)**

```
def eval_polH_rec(P:list,x:float)->float:
    n=len(P)-1
    if n==0:
        return P[0]
    else:
        return eval_polH_rec(P[:n],x)*x+P[n]
```

2.2) La suite (w_n) vérifie $w_0 = 0$ et $\forall n \in \mathbb{N}, w_{n+1} = w_n + 1$; on en déduit donc que $w_n = n$ ce qui est plus efficace que les fonctions précédentes.

2.3)

```
def eval_polH(P:list,x:float)->float:
    n, res = len(P)-1, 0
    for k in range(n+1):
        res = x*res + P[k]
    return res
```

Solution 6

1)
- Si $n = 1$: il y a juste un disque à déplacer de `Tdepart` vers `Tarrivee`.
- Si $n > 1$:
  1. pour libérer le plus grand disque, il faut déplacer la pile des $n-1$ premiers disques de `Tdepart` vers `Taux` (en respectant les règles),
  2. on peut alors déplacer le plus grand disque de `Tdepart` vers `Tarrivee` (les règles sont respectées car `Tarrivee` est vide),
  3. il faut déplacer la pile des $n-1$ premiers disques de `Taux` vers `Tarrivee` (en respectant les règles, ce qui est possible car `Tdepart` est vide et tous les disques de cette pile ont un diamètre inférieur à celui qui est sur `Tarrivee`).

2) La fonction récursive (elle ne renvoie pas de résultat) :

```python
def Hanoi(n:int, Tdepart:int, Tarrivee:int, Taux:int) -> None:
    if n == 1:
        print(Tdepart, "->", Tarrivee)  # un seul disque à déplacer
    else :
        Hanoi(n-1, Tdepart, Taux, Tarrivee)  # on déplace les n-1 premiers
        print(Tdepart, "->", Tarrivee)  # on déplace le plus grand disque
        Hanoi(n-1, Taux, Tarrivee, Tdepart)  # on met en place les n-1 premiers
```

Solution 7

1) d_n est la complexité de `Hanoi(n, Tdepart, Tarrivee, Taux)` en nombre d'affichages, on a évidemment $d_1 = 1$.

2) Lorsqu'on exécute `Hanoi(n, Tdepart, Tarrivee, Taux)` il y a deux appels à la fonction `Hanoi(n-1, ..., ..., ...)` car $n > 1$, plus un affichage, ce qui donne bien $d_n = 2d_{n-1} + 1$.

3) On a $d_n + 1 = 2(d_{n-1} + 1)$, on a donc une suite géométrique de raison 2, d'où $d_n + 1 = 2^{n-1}(d_1 + 1)$ et donc $d_n = 2^n - 1$.