

Récurtivité et dictionnaires

February 17, 2022

ATTENTION il faut faire ce TP sur Firefox ou Chrome.

1 Fonctions récursives

Il est possible en Python qu'une fonction s'appelle elle-même. Il n'y a rien de particulier à faire, tout se passe naturellement :

```
[1]: def compte_rebours(n):  
    print(n, '...')  
    if n > 1:  
        compte_rebours(n-1)  
    else:  
        print('Partez !')  
compte_rebours(5)
```

```
5 ...  
4 ...  
3 ...  
2 ...  
1 ...  
Partez !
```

Ici, on remarque un point essentiel : on est obligé de s'arrêter et pour cela il ne faut pas faire d'appels récursifs (nom donné à un appel dans une fonction d'elle-même), dans tous les cas. Les valeurs des arguments qui ne produisent pas d'appels récursifs sont appelés les **cas de base**. Il est important d'être sûr qu'on va bien atteindre un cas de base.

Les fonctions **récursives** sont à l'informatique ce que les démonstrations par **réurrence** sont aux mathématiques: cas de base $\Leftrightarrow$ initialisation, appel récursif $\Leftrightarrow$ hérédité.

Prenons l'exemple d'une fonction pour calculer $n!$. Quand on appelle **fact(5)**, celle-ci va appeler **fact(4)** qui va appeler **fact(3)**, et ainsi de suite jusqu'à l'appel de **fact(0)** qui est le cas de base et va produire une valeur 1 **sans effectuer d'appels récursifs**. A partir de celle-ci, on va remonter jusqu'à la valeur de **fact(5)**.

```
[2]: def fact(n):  
    if n == 0:  
        return 1  
    else:
```

```
    return n * fact(n-1)

fact(5)
```

[2]: 120

Si jamais on omet le cas `n == 0`, on se retrouve avec la fonction suivante qui va mettre longtemps avant de produire une erreur `RecursionError` qui détecte qu'on a fait trop d'appels récursifs, et donc qu'il y a sûrement un problème. Pour comprendre pourquoi c'est le cas, on peut reprendre l'analyse précédente, sauf qu'au lieu de s'arrêter à l'appel de `fact(0)`, on continue avec `fact(-1)`, puis `fact(-2)`, ...

```
[3]: def fact(n):
      return n * fact(n-1)
      fact(5)
```

```

↳ -----
RecursionError                                Traceback (most recent call↳
↳ last)

<ipython-input-3-3dff359507f0> in <module>
      1 def fact(n):
      2     return n * fact(n-1)
----> 3 fact(5)

<ipython-input-3-3dff359507f0> in fact(n)
      1 def fact(n):
----> 2     return n * fact(n-1)
      3 fact(5)

... last 1 frames repeated, from the frame below ...

<ipython-input-3-3dff359507f0> in fact(n)
      1 def fact(n):
----> 2     return n * fact(n-1)
      3 fact(5)

RecursionError: maximum recursion depth exceeded
```

Pour les fonctions suivantes, on ne va pas utiliser `print` mais une fonction nommée `affiche` pour

l'affichage, ce qui permettra aux tests de pouvoir vérifier que tout se passe bien.

On vous demande d'écrire des fonctions récursives (**sans boucle** donc) `triangle_descendant` et `triangle_ascendant`. On rappelle que pour faire une chaîne de caractère qui contient 5 fois le caractère '*' il suffit d'écrire '*'*5.

`triangle_descendant(7)` doit afficher :

```
*****
*****
*****
****
***
**
*
```

`triangle_ascendant(7)` doit afficher :

```
*
**
***
****
*****
*****
*****
```

```
[4]: def triangle_descendant(n):
      # BEGIN SOLUTION
      if n > 0:
          affiche('*'*n)
          triangle_descendant(n-1)
      else: # pas nécessaire mais permet de mettre
            pass # en évidence que le cas de base est ici
      # END SOLUTION
```

```
[5]: def triangle_ascendant(n):
      # BEGIN SOLUTION
      if n > 0:
          triangle_ascendant(n-1)
          affiche('*'*n)
      # END SOLUTION
```

```
[6]: from civ import *
      init()
      test_affiche(triangle_descendant, 5, ['*****', '****', '***', '**', '*'])
      test_affiche(triangle_ascendant, 5, ['*', '**', '***', '****', '*****'])
      valide()
```

```
*****
*****
```

```

***
**
*
OK!: triangle_descendant(5) affiche ce qui était attendu.
*
**
***
****
*****
OK!: triangle_ascendant(5) affiche ce qui était attendu.
Tous les tests sont valides

```

2 Apprivoiser une tortue

Dans cette partie du TP, on va réaliser des dessins en faisant se déplacer une petite tortue sur l'écran. Cette tortue démarre à la coordonnée (10,250) d'une grille de dimension 500x500 en regardant vers la droite, et on interagit avec elle à l'aide de trois fonctions :

- `avance(n)` fait avancer la tortue de `n` pixels dans la direction qui lui fait face
- `tourne_gauche(a)` fait tourner la direction de la tortue vers la gauche (donc sens trigo) de `a` degrés
- `tourne_droite(a)` idem

Important : on considère que les fonctions suivantes sont tout le temps définies. En fait, toutes vos fonctions seront évaluées par les tests automatiques qui feront en sorte de créer la tortue et de rendre disponible ces fonctions.

Voici un exemple où on a tracé un triangle équilatéral :

```

[7]: from civ import *
      init(turtle=True)
      for i in range(3):
          avance(100)
          tourne_gauche(120)

```

Turtle()

```

[8]: def carre(n):
      '''Trace un carre de coté n en partant
      de la direction de la tortue et en allant vers la gauche
      comme pour le triangle au dessus'''
      # BEGIN SOLUTION
      for i in range(4):
          avance(n)
          tourne_gauche(90)
      # END SOLUTION

```

```
[9]: from civ import *
init(turtle=True)
test_turtle(carre, 100, [(10, 250), (110, 250), (110, 150), (10, 150), (10,
↪250)])
valide()
```

Turtle()

OK!: carre(100) effectue le bon tracé
Tous les tests sont valides

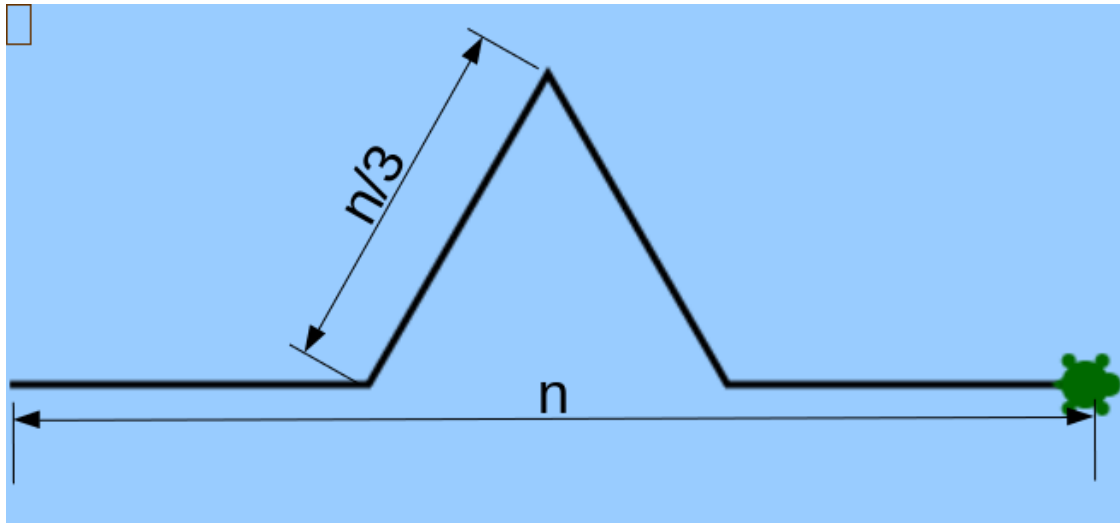
```
[10]: def polygone(k, n):
    '''Trace un polygone régulier à k cotés de côté n en partant
    de la direction de la tortue et en allant vers la gauche
    comme pour le triangle au dessus'''
    # BEGIN SOLUTION
    for i in range(k):
        avance(n)
        tourne_gauche(360/k)
    # END SOLUTION
```

```
[11]: from civ import *
init(turtle=True, speed=10)
test_turtle(polygone, 4, 100, [(10, 250), (110.0, 250.0), (110.0, 150.0), (10,
↪150.0), (10.0, 250.0)])
avance(100)
test_turtle(polygone, 5, 100, [(10, 250), (110.0, 250.0), (110.0, 150.0), (10.
↪0, 150.0), (10.0, 250.0), (110.0, 250.0), (210.0, 250.0), (240.9, 154.9),
↪(160.0, 96.10000000000001), (79.1, 154.9), (110.0, 250.0)])
avance(100)
test_turtle(polygone, 6, 100, [(10, 250), (110.0, 250.0), (110.0, 150.0), (10.
↪0, 150.0), (10.0, 250.0), (110.0, 250.0), (210.0, 250.0), (240.9, 154.9),
↪(160.0, 96.10000000000001), (79.1, 154.9), (110.0, 250.0), (210.0, 250.0),
↪(310.0, 250.0), (360.0, 163.4), (310.0, 76.80000000000001), (210.0, 76.
↪80000000000001), (160.0, 163.4), (210.0, 250.0)])
valide()
```

Turtle()

OK!: polygone(4, 100) effectue le bon tracé
OK!: polygone(5, 100) effectue le bon tracé
OK!: polygone(6, 100) effectue le bon tracé
Tous les tests sont valides

On vous demande ici de faire le motif suivant qui, étant donné une valeur n , effectue un déplacement de $n/3$, puis parcourt un triangle équilatéral de $n/3$ et avance à nouveau de $n/3$.



```
[5]: def motif(n):
      '''Réalise le motif précédent'''
      # BEGIN SOLUTION
      avance(n//3)
      tourne_gauche(60)
      avance(n//3)
      tourne_droite(120)
      avance(n//3)
      tourne_gauche(60)
      avance(n//3)
      # END SOLUTION
```

```
[6]: from civ import *
      init(turtle=True, speed=10)
      test_turtle(motif, 450, [(10, 250), (160.0, 250.0), (235.0, 120.1), (310.0, 250.
      ↪0), (460.0, 250.0)])
      valide()
```

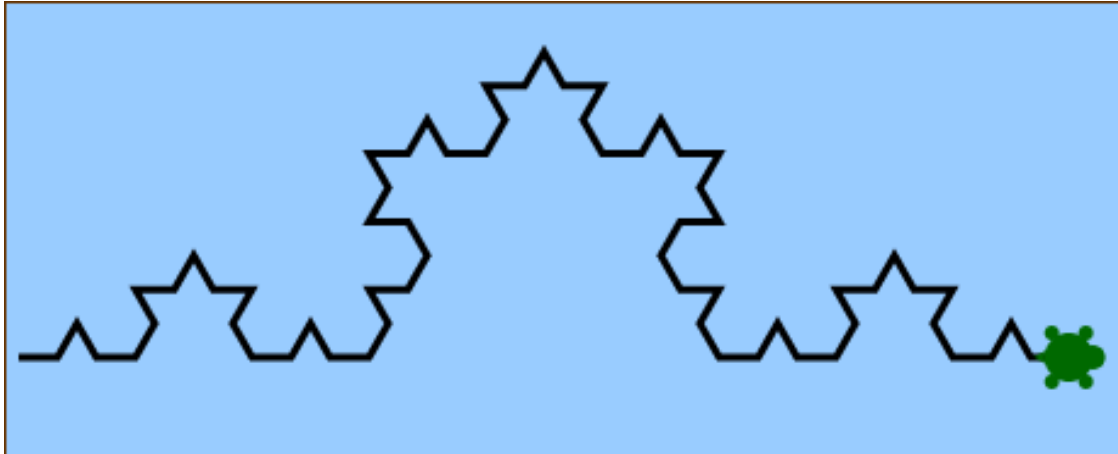
Turtle()

OK!: motif(450) effectue le bon tracé

Tous les tests sont valides

On peut maintenant tirer partie de la récursivité pour définir une fonction `motif_rec` prenant deux arguments. `motif_rec(k,n)` va réaliser le motif précédent mais en utilisant, si $k > 0$, `motif_rec(k-1,n//3)` à la place de `avance(n//3)`. Le cas de base sera pour $k = 0$ où on demande juste d'avancer de n pixels. L'appel de `motif_rec(1,n)` doit donc se comporter comme la fonction `motif(n)` précédente.

Voilà ce que devrait afficher `motif_rec(3,450)` :



```
[13]: def motif_rec(k, n):
      # BEGIN SOLUTION
      if k == 0:
          avance(n)
      else:
          motif_rec(k-1, n//3)
          tourne_gauche(60)
          motif_rec(k-1, n//3)
          tourne_droite(120)
          motif_rec(k-1, n//3)
          tourne_gauche(60)
          motif_rec(k-1, n//3)
      # END SOLUTION
```

```
[14]: from civ import *
      init(turtle=True, speed=10)
      test_turtle(motif_rec, 1, 450, [(10, 250), (160.0, 250.0), (235.0, 120.1), (310.
      ↪ 0, 250.0), (460.0, 250.0)])
      valide()
```

Turtle()

OK!: motif_rec(1, 450) effectue le bon tracé
Tous les tests sont valides

```
[16]: from civ import *
      init(turtle=True, speed=20)
      test_turtle(motif_rec, 2, 450, [(10, 250), (60.0, 250.0), (85.0, 206.7), (110.
      ↪ 0, 250.0), (160.0, 250.0), (185.0, 206.7), (160.0, 163.39999999999998), (210.
      ↪ 0, 163.39999999999998), (235.0, 120.09999999999998), (260.0, 163.
      ↪ 3999999999999998), (310.0, 163.39999999999998), (285.0, 206.7), (310.0, 250.
      ↪ 0), (360.0, 250.0), (385.0, 206.7), (410.0, 250.0), (460.0, 250.0)])
      valide()
```

Turtle()

OK!: motif_rec(2, 450) effectue le bon tracé

Tous les tests sont valides

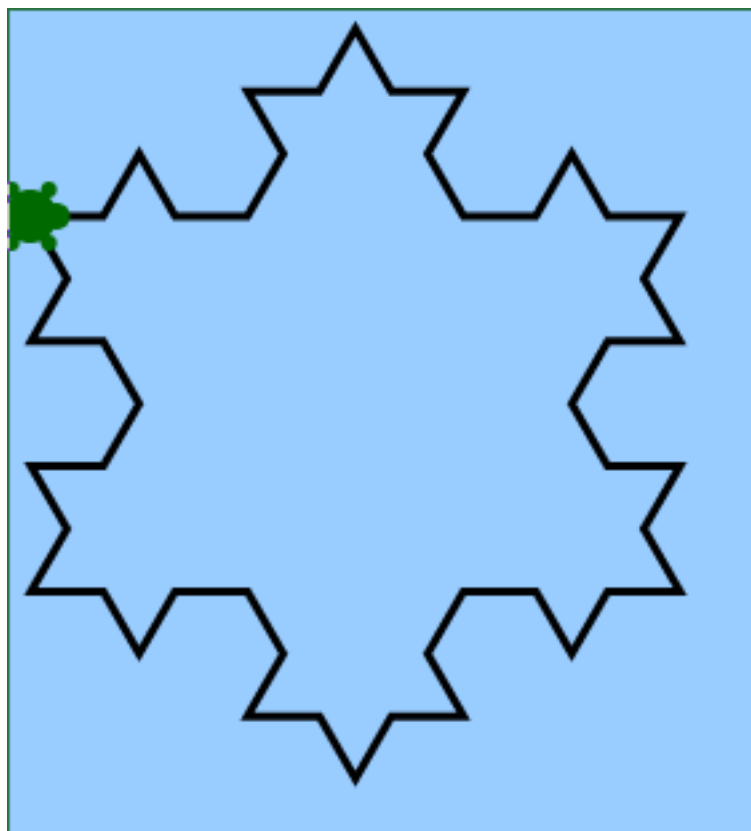
```
[15]: from civ import *
init(turtle=True,speed=100)
test_turtle(motif_rec, 3, 450, [(10, 250), (26.0, 250.0), (34.0, 236.1), (42.0, 250.0), (58.0, 250.0), (66.0, 236.1), (58.0, 222.2), (74.0, 222.2), (82.0, 208.29999999999998), (90.0, 222.2), (106.0, 222.2), (98.0, 236.1), (106.0, 250.0), (122.0, 250.0), (130.0, 236.1), (138.0, 250.0), (154.0, 250.0), (162.0, 236.1), (154.0, 222.2), (170.0, 222.2), (178.0, 208.29999999999998), (170.0, 194.39999999999998), (154.0, 194.39999999999998), (162.0, 180.49999999999997), (154.0, 166.59999999999997), (170.0, 166.59999999999997), (178.0, 152.69999999999996), (186.0, 166.59999999999997), (202.0, 166.59999999999997), (210.0, 152.69999999999996), (202.0, 138.79999999999995), (218.0, 138.79999999999995), (226.0, 124.89999999999995), (234.0, 138.79999999999995), (250.0, 138.79999999999995), (242.0, 152.69999999999996), (250.0, 166.59999999999997), (266.0, 166.59999999999997), (274.0, 152.69999999999996), (282.0, 166.59999999999997), (298.0, 166.59999999999997), (290.0, 180.49999999999997), (298.0, 194.39999999999998), (282.0, 194.39999999999998), (274.0, 208.29999999999998), (282.0, 222.2), (298.0, 222.2), (290.0, 236.1), (298.0, 250.0), (314.0, 250.0), (322.0, 236.1), (330.0, 250.0), (346.0, 250.0), (354.0, 236.1), (346.0, 222.2), (362.0, 222.2), (370.0, 208.29999999999998), (378.0, 222.2), (394.0, 222.2), (386.0, 236.1), (394.0, 250.0), (410.0, 250.0), (418.0, 236.1), (426.0, 250.0), (442.0, 250.0)])
valide()
```

Turtle()

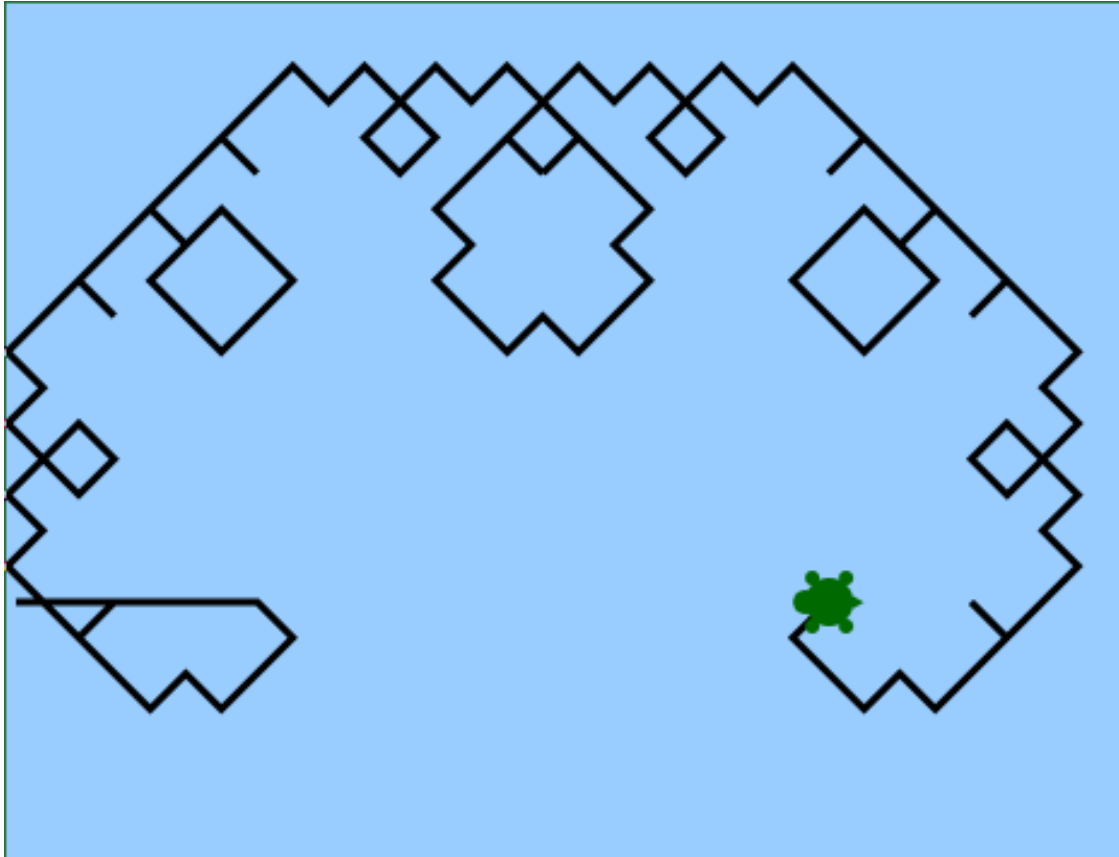
OK!: motif_rec(3, 450) effectue le bon tracé

Tous les tests sont valides

Expression libre : vous pouvez maintenant essayer de faire un flocon de



Von Koch : ou en-
core trouver le bon motif pour réaliser le dessin (7^{eme} génération) suivant



3 Dictionnaires

3.1 Présentation

Les dictionnaires (type `dict` en Python) permettent de représenter des associations entre des **clés** et des **valeurs**. Par exemple le programme suivant :

```
d = { 'nom' : 'Évariste', 'age' : 23, 42 : True }
```

définit un dictionnaire associé à la variable `d` et qui contient trois clés : `'nom'`, `'age'` et `42`. La valeur associée à la clé `'age'` est l'entier `23`. On remarque donc :

- les clés peuvent être des valeurs de différents types **immuables** : qu'on ne peut pas modifier, donc des **tuple** mais pas des **list**
- les valeurs peuvent être des valeurs de types quelconques.

Pour accéder à la valeur associée à la clé `'nom'` on écrit `d['nom']`. Comme pour un tableau, on peut y accéder en lecture ou en écriture :

```
x = d['nom'] # x contient la valeur 'Évariste'
d['nom'] = 'Ada' # d associe maintenant 'Ada' à la clé 'nom'
```

- Le dictionnaire vide est `{}`
- On peut tester si une clé `c` est dans le dictionnaire `d` avec la valeur `c in d` qui est de type `bool`

- On peut supprimer l'association de la clé `c` dans `d` avec l'instruction `del d[c]` (rarement utile)
- En utilisant une boucle `for` on peut parcourir toutes les clés d'un dictionnaire : `for clé in d: ...`
- **Avancé** : on peut obtenir les valeurs dans `d` à l'aide de `d.values()` ainsi que les couples (clé,valeur) avec `d.items()`.

On suppose que chacune de ces opérations sur un dictionnaire s'effectue en temps constant, c'est-à-dire avec un nombre d'instructions borné et notamment indépendant du nombre de clés.

```
[18]: d = { 'nom' : 'Évariste', 'age' : 23, 42 : True }

for clé in d:
    print('Clé', clé)

for valeur in d.values():
    print('Valeur', valeur)

for clé, valeur in d.items():
    print('Association', clé, '->', valeur)
```

```
Clé nom
Clé age
Clé 42
Valeur Évariste
Valeur 23
Valeur True
Association nom -> Évariste
Association age -> 23
Association 42 -> True
```

3.2 Comptage

Dans la question suivante on va utiliser un dictionnaire pour compter les occurrences dans un tableau, c'est-à-dire le nombre de fois que chaque valeur est présente. Pour cela, on va créer un dictionnaire `occ` tel que si `x` est dans le tableau, alors `occ[x]` contient le nombre de fois qu'il est présente.

Les valeurs qui ne sont pas dans le tableau ne doivent pas apparaître comme clé dans le dictionnaire.

```
[19]: def comptage(tableau):
    '''renvoie un dictionnaire associant à chaque valeur dans tableau le nombre_
    entier de fois où
    cette valeur est présente'''
    ## BEGIN SOLUTION
    occ = {}
    for valeur in tableau:
        if valeur not in occ:
            occ[valeur] = 0
```

```

    occ[valeur] = 1 + occ[valeur]
return occ
## END SOLUTION

```

```

[20]: from civ import *
init()
test(comptage, ['a','b','a'], { 'a': 2, 'b' : 1 })
test(comptage, [1,1,1], { 1: 3 })
test(comptage, [], {})
test(comptage, [1,2,1], { 2: 1, 1: 2 })
valide()

```

OK!: `comptage(['a', 'b', 'a'])` vaut `{'a': 2, 'b': 1}`
 OK!: `comptage([1, 1, 1])` vaut `{1: 3}`
 OK!: `comptage([])` vaut `{}`
 OK!: `comptage([1, 2, 1])` vaut `{2: 1, 1: 2}`
 Tous les tests sont valides

Avancé : il est possible d'utiliser `d.get(clé, valeur_si_absente)` pour obtenir la valeur associée à une clé et, si cette clé n'est pas associée, renvoyer la valeur `valeur_si_absente`. Ainsi, on peut écrire

```
occ[valeur] = 1 + occ.get(valeur, 0)
```

3.3 Tri par comptage

Si on connaît à l'avance l'ensemble **ordonné** de valeurs présentes dans un tableau, il est possible d'utiliser le comptage pour en déduire un tri. Par exemple, si on sait que le tableau ne contient que des valeurs entières entre 1 et 4 et qu'on sait qu'il contient 4 fois 1, 3 fois 2 et 1 fois 4, alors trié par l'ordre croissant, c'est forcément le tableau `[1,1,1,1,2,2,2,4]`.

On peut donc en déduire un algorithme de tri appelé **tri par comptage** et qui est en temps linéaire sur le nombre de données et sur le nombre de valeurs possibles. Si le tableau contient n valeur et qu'il y a p valeurs possibles, on dira que sa complexité temporelle en pire cas est en $O(n + p)$.

```

[21]: def tri_comptage(atrier, possibles):
    '''Le tableau atriier contient uniquement des valeurs présentes, dans
    ↳ l'ordre croissant, dans le tableau
    possibles. On demande de renvoyer le tableau correspondant à atriier avec
    ↳ les valeurs triées dans l'ordre
    croissant déduit de possibles. On pourra utiliser la fonction comptage
    ↳ précédemment définie.'''
    ## BEGIN SOLUTION
    occ = comptage(atrier)
    trie = [ None ] * len(atrier)
    i = 0
    for possible in possibles:
        if possible in occ:
            for j in range(occ[possible]):

```

```

        trie[i] = possible
        i += 1
    return trie
## END SOLUTION

```

```

[22]: from civ import *
      init()
      test(tri_comptage, [4,3,2,1], [1,2,3,4], [1,2,3,4])
      test(tri_comptage, [1,2,1,3,2], [1,2,3], [1,1,2,2,3])
      test(tri_comptage, [], [1], [])
      valide()

```

```

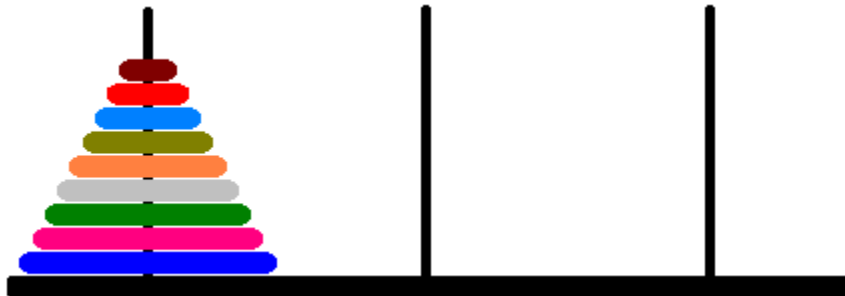
OK!: tri_comptage([4, 3, 2, 1], [1, 2, 3, 4]) vaut [1, 2, 3, 4]
OK!: tri_comptage([1, 2, 1, 3, 2], [1, 2, 3]) vaut [1, 1, 2, 2, 3]
OK!: tri_comptage([], [1]) vaut []
Tous les tests sont valides

```

4 Tours de Hanoi

Les tours de hanoï sont un casse-tête très ancien. Trois piquets sont plantés côte à côte sur le sol. Sur le premier piquet sont enfilés des disques de bois de différentes tailles, du plus grand au plus petit.

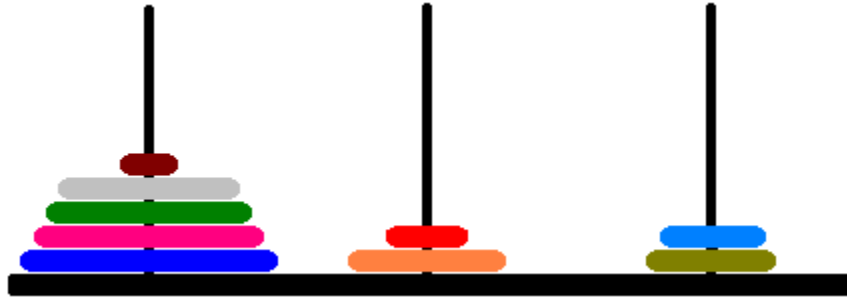
Voici une illustration d'un jeu de tours de Hanoï composé de 9 disques :



Le but du casse-tête est de déplacer l'ensemble de la pile de disques vers le troisième piquet. Un coup consiste à déplacer un disque du sommet d'un piquet vers le sommet d'un autre piquet, en respectant la règle suivante :

Un disque ne peut jamais être placé sur un disque plus petit que lui.

Illustration de l'état du jeu après quelques coups :



Vous devez écrire un programme qui, étant donné le nombre de disques se trouvant sur le premier piquet, décrit comment atteindre la position finale en effectuant un minimum de coups.

Les piquets sont représentés par un numéro 1, 2 ou 3 de gauche à droite.

Les déplacements sont représentés par des couples (i, j) indiquant un déplacement d'un disque du piquet i au piquet j . Une solution est donc une liste de couples. On rappelle que le symbole $+$ permet de concaténer (coller) deux listes.

Pour résoudre le programme, demandez-vous comment vous pourriez résoudre le problème avec n disques **quand vous savez le résoudre avec $n-1$** . On tirera notamment partie du fait qu'on peut changer les rôles des piquets dans des appels récursifs.

```
[23]: def hanoi(n, src, dest, temp):
    '''Renvoie la liste des déplacements pour déplacer n disques du piquet src_
    ↪ vers
    le piquet dest en utilisant le piquet temp comme piquet de travail.'''
    # BEGIN SOLUTION
    if n == 0:
        return []
    l1 = hanoi(n-1, src, temp, dest)
    l2 = hanoi(n-1, temp, dest, src)
    return l1 + [ (src, dest) ] + l2
    # END SOLUTION
```

```
[24]: # Pour vous permettre de **voir** votre solution
# A NE PAS LIRE (enfin, sauf si vous voulez comprendre comment on fait, ce qui_
    ↪ est bien aussi)
def piquet_vers_str(n,l):
    s = []
    t = 2*(n-1)+1+2
    for di in l:
        d = 2*(di-1)+1
        ds = '#' * d
        ss = ' ' * ((t-d)//2)
        s.append( ss + ds + ss )
    s += [ ' '*(t//2)+'|'+ ' '*(t//2) ] * (n-len(l))
    return s
def affiche_config(n, piquets):
```

```

l1, l2, l3 = map(lambda l: piquet_vers_str(n, l), piquets)
for i in range(1,n+1):
    print(l1[-i]+l2[-i]+l3[-i])
def affiche_solution(n, l):
    piquets = [ list(reversed(range(1,n+1))), [], [] ]
    affiche_config(n, piquets)
    for i, j in l:
        disque = piquets[i-1].pop()
        piquets[j-1].append(disque)
        print()
        affiche_config(n, piquets)

```

[25]: *# si vous avez bien repondu, vous devriez avoir le bon déroulement*
 affiche_solution(3, hanoi(3, 1, 3, 2))

```

#      |      |
###    |      |
##### |      |

|      |      |
###    |      |
##### |      #

|      |      |
|      |      |
#####  ###   #

|      |      |
|      #      |
#####  ###   |

|      |      |
|      #      |
|      ###   #####

|      |      |
|      |      |
#      ###   #####

|      |      |
|      |      ###
#      |      #####

|      |      #
|      |      ###
|      |      #####

```

```
[26]: from civ import *
init()
test(hanoi, 3, 1, 2, 3, [(1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1,
↪2)])
test(hanoi, 3, 2, 1, 3, [(2, 1), (2, 3), (1, 3), (2, 1), (3, 2), (3, 1), (2,
↪1)])
test(hanoi, 4, 3, 2, 1, [(3, 1), (3, 2), (1, 2), (3, 1), (2, 3), (2, 1), (3,
↪1), (3, 2), (1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2)])
valide()
```

OK!: hanoi(3, 1, 2, 3) vaut [(1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2)]

OK!: hanoi(3, 2, 1, 3) vaut [(2, 1), (2, 3), (1, 3), (2, 1), (3, 2), (3, 1), (2, 1)]

OK!: hanoi(4, 3, 2, 1) vaut [(3, 1), (3, 2), (1, 2), (3, 1), (2, 3), (2, 1), (3, 1), (3, 2), (1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2)]

Tous les tests sont valides