Info 3 : Récursivité
Proposition de correction

Contents

1 Qu'est-ce que la récursivité ? 1
- 1.1 Un exemple classique... 1
- 1.2 Pour aller plus loin (mais au programme !) - Un peu de vocabulaire 2
- 1.3 Définition et caractéristiques 4

2 Un exercice visuel : flocons de Von Koch 4

3 Un exercice classique... 6

4 S'exercer à la récursion 9
- 4.1 L'essentiel 9
- 4.2 Retour sur les Tours de Hanoi 10
- 4.3 Pour aller plus loin... 11

1 Qu'est-ce que la récursivité ?

1.1 Un exemple classique...

Intéressons-nous à la fonction factorielle, définie ainsi :

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

Il est possible d'écrire une fonction "basique" fact(n) retournant la valeur de $n!$

Q1 : Écrire cette fonction et la tester.

[6]:
```
def fact(n):
    f=1
    for i in range(1,n+1):
        f=i*f
    return f
```

Essayons :

[7]:
```
print("Factorielle de 5 : ",fact(5),". Factorielle de 0 : ",fact(0),".␣
↪Factorielle de 1 : ",fact(1))
```

```
Factorielle de 5 :  120 . Factorielle de 0 :  1 . Factorielle de 1 :  1
```

Cependant, il y a une façon bien plus élégante d'écrire cette fonction, en utilisant la **récursivité**. Celle-ci repose sur le principe d'utiliser, dans la fonction qu'on définit, cette **même** fonction. C'est ce qu'on fait *naturellement* en écrivant la définition mathématique $n! = n \times (n-1)!$: $n!$ est définie via $(n-1)!$.

Ecrivons alors la version *plus élégante* de la fonction précédente, la fonction fact_el(n) :

[8]:
```
def fact_el(n):
    if n==0 :
        return 1
    else :
        return fact_el(n-1)*n
```

Q2 : Effectuer le calcul de 3! grâce à cette fonction, en utilisant le debogeur intégré à *Thonny*. Pourquoi plusieurs fenêtres s'ouvrent ? Combien (au maximum) de calculs sont *commencés* ? Quelle notion vue récemment peut être utile pour expliquer cette exécution ?

[9]:
```
print("Factorielle de 5 : ",fact_el(5),". Factorielle de 0 : ",fact_el(0),".␣
↪Factorielle de 1 : ",fact_el(1))
```

```
Factorielle de 5 :  120 . Factorielle de 0 :  1 . Factorielle de 1 :  1
```

Plusieurs *calculs* s'ouvrent dans plusieurs fenêtres : l'exécution commence par *l'ouverture* du calcul de 3!. Pour ce calcul, elle a besoin de la valeur de 2! : une nouvelle exécution *s'ouvre* pour obtenir la valeur de 2!, alors que la précédente (celle pour obtenir 3!) n'est pas encore finie, donc encore *ouverte*. Et ces *ouvertures d'exécutions* se poursuivent, jusqu'à arriver au calcul de 0!. Cette fois-ci, le code de la fonction 'fect_el{ ne nécessite plus *l'ouverture* d'un nouveau calcul, car le résultat est explicite (il s'agit de 1) : on a *terminé* d'ouvrir des exécutions de calculs. Mais le calculs de 3! n'est pas fini ! Avec la valeur de 0!, le calcul *encore ouvert* de 1! va pouvoir se *terminer* (ou *fermer*); puis celui de 2!, et ainsi de suite.

On se rend donc compte que le **premier** calcul à se *terminer* (ou *finir*) est le **dernier* a avoir été *ouvert* : on pourrait nommer ces exécutions *ouvertes* une** pile d'exécution**, obéissant au principe *LIFO* (*Last in First Out*).

1.2 Pour aller plus loin (mais au programme !) - Un peu de vocabulaire ....

Pile d'éxécution : A chaque appel d'une fonction, python stocke cette information dans une pile, ainsi que la valeur de toutes les variables associées. Sur le principe d'une pile, lorsque la fonction est effectuée ou terminée, ces informations sont retirées du sommet de la pile. Dans le cas d'une fonction récursive faisant de (trop) multiples appels à elle-même, sans jamais "terminer" ces fonctions, on se retrouve avec un nombre d'informations stockées qui augmente exponentiellement : on dit que la pile "*déborde*". C'est comme écrire une expression mathématique en ouvrant un nombre bien trop important de parenthèses (ici inutiles), sans jamais les fermer :$(1+(2+(3+\dots)\dots))$. On est alors vite "débordé", et on a beacoup de mal à savoir s'il y a bien le bon nombre de parenthèses fermantes ! On se trouve dans la situation bien connue des programmateurs : celle du "*Stack overflow*", tellement célèbre quelle a donné son nom à un célèbre site d'entraide entre développeurs !

A notre petit niveau, on peut essayer de voir la réaciton de python à l'appel de la fonction fact_el pour un grand nombre :

[10]:
```
fact_el(100000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[10], line 1
----> 1 fact_el(100000)

Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

    [… skipping similar frames: fact_el at line 5 (2973 times)]

Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

Cell In[8], line 1, in fact_el(n)
----> 1 def fact_el(n):
      2     if n==0 :
      3         return 1

RecursionError: maximum recursion depth exceeded
```

Dans Python est implémenté une protection, fixant un nombre maximum d'appels récursifs (soit une taille maximale à la pile), que l'on peut modifier à l'aide de la fonction setrecursionlimit(n).

Terminaison : Une fonction définie par récursivité doit avoir une **condition d'arrêt** valide, comme lorsque l'on utilise une condition *while*. Par exemple, on peut essayer d'effectuer fact_el(5.3) : on se retrouve dans une boucle infini !

[13]:
```
print("Factorielle de 5 : ",fact_el(5.3))
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[13], line 1
----> 1 print("Factorielle de 5 : ",fact_el(5.3))
```

```
Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

    [… skipping similar frames: fact_el at line 5 (2974 times)]

Cell In[8], line 5, in fact_el(n)
      3     return 1
      4 else :
----> 5     return fact_el(n-1)*n

RecursionError: maximum recursion depth exceeded
```

Heureusement, la limite du nombre de récursion montre ici son utilité, en empêchant au calcul de se poursuivre indéfiniment.

On peut ainsi démontrer qu'un algorithme récursif se termine ou non, par récurrence, en cherchant un variable entière strictement décroissante et minorée. Dans le cas de la fonction fact_el, il s'agit simplement de la variable n, strictement décroissante à chaque nouvel appel de la fonction, et minorée (dans le cas où n est entier !) par 0 : le code `n==0` est la condition de terminaison. **Une condition de terminaison doit toujours exister dans une fonction récursive**.

Complexité : Pour la fonction fact_el(n), l'unique opération pour chaque valeur de n est d'appeler cette même fonction, pour la valeur (n-1), jusqu'à arriver à la valeur n=0. Pour obtenir $n!$, on fait donc n opérations de coût $\mathcal{O}(1)$, et donc au total cette fonction a pour complexité $\mathcal{O}(n)$, soit la même complexité que pour la version non récursive.

1.3 Définition et caractéristiques

Une fonction **récursives** est une fonction qui fait appel à **elle-même** .

Remarques :

- Une fonction récursive possède généralement un ou des cas à traiter *à part*, de façon non récursive, qui permettent généralement la terminaison de la fonction. Par exemple, pour la fonction fact_el(n), il s'agit de la portion du code if n==0 :.
- La récursivité est semblable au principe de *récurence* en mathématique.

Avantage de la récursivité : L'écriture de la fonction est généralement bien plus simple, intuitive et compréhensible. De plus, elle se rapproche de l'écriture mathématique, comme dans le cas de la fonction factorielle.

Inconvénient de la récursivité : Cette simplicité d'écriture peut se traduire par une plus grande complexité pour la machine, et donc une exécution plus lente.

Remarque : le terme complexité fait ici référence à divers notion : temps de calculs, empreinte mémoire, etc.

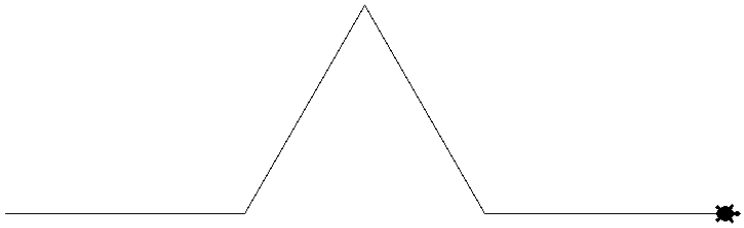
2 Un exercice visuel : flocons de Von Koch

Dans cet exercice, nous allons présenter un code pour générer des tracer de courbes de Von Koch, des "flocons". Cela nécessite une bibliothèque nommée *turtle*. Commençons par une partie du code nécessaire pour tracer ces flocons : la fonction `côté`.

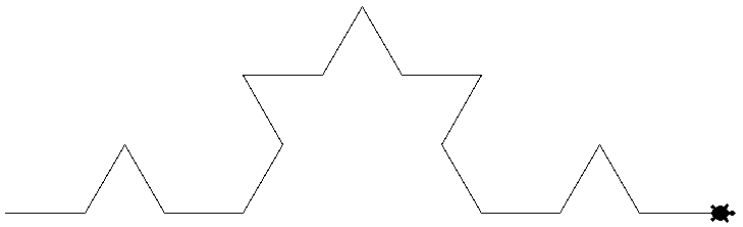
Q3 : Valider le code correspondant du fichier *I3_Rec.py* dans *Thonny*, puis déterminer quels sont les rôles des arguments `taille_fig` et `recursivite`(lors des tests, ne pas dépasser `recursivite=4`).

Pour le paramètre de réursivité à 0 :

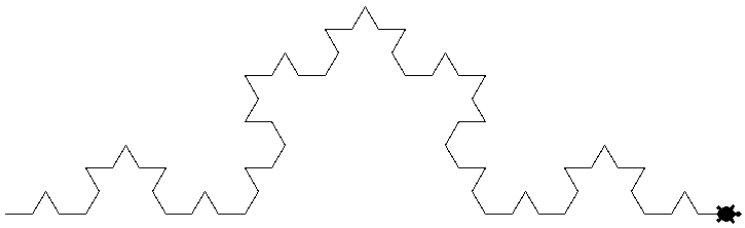
Pour le paramètre de réursivité à 1 :



Pour le paramètre de réursivité à 2 :

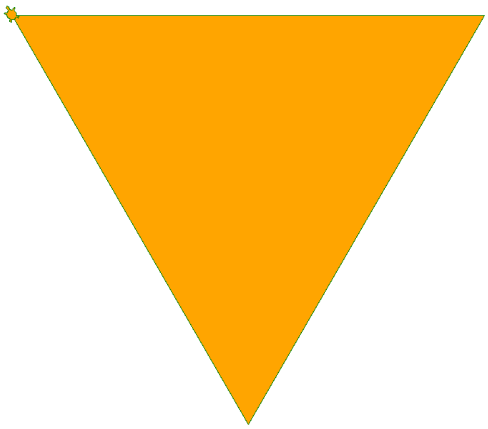


Pour le paramètre de réursivité à 3 :

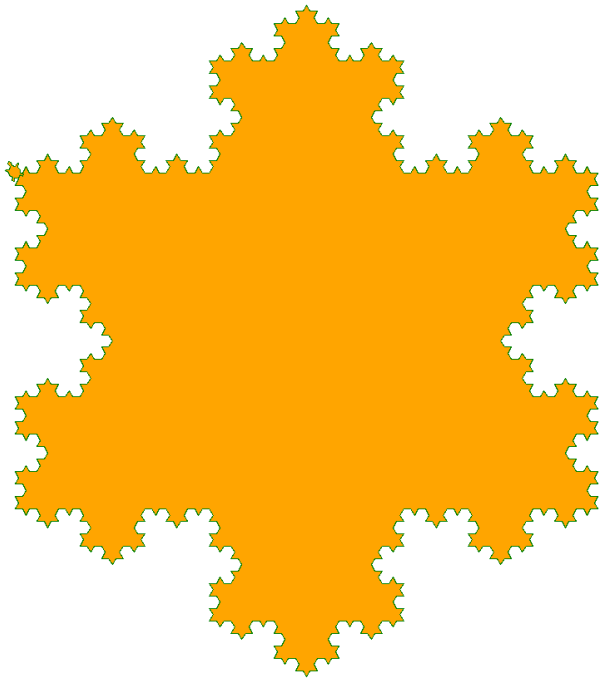


Q4 : Même question que précédemment, avec le code correspondant et les variables `taille_fig` et `recursivite`.

Pour le paramètre de réursivité à 0 :



Pour le paramètre de réursivité à 4 :



3 Un exercice classique...

On s'intéresse au problème suivant : le but est de déterminer le nombre u_n de façon de monter un escalier de n marches, sachant qu'on peut monter au choix une ou deux marches d'un seul pas.

Q5 : Donner les valeurs de u_1, u_2, u_3 et u_4.

Correction : $u_1 = 1$, $u_2 = 2$, $u_3 = 3$ etr $u_4 = 5$.

Q6 : Écrire la fonction `un(n)` retournant u_n pour un nombre de marches $n \geq 1$. Le but est évidemment d'écrire une fonction récursive : pour cela, réfléchir sur la façon d'atteindre la nième marche (en particulier, sur quelle marche se trouvait-on auparavant ?).

Correction : La suite se définit ainsi :

$$\begin{cases} u_1 = 1 \text{ et } u_2 = 2 \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

Et la fonction python récursive s'écrit :

[3]:
```
def un(n):
    assert n>= 1
    if n == 1 :
        return 1
    elif n == 2 :
        return 2
    else :
        return un(n-1)+un(n-2)
```

[7]:
```
print("1 marche :",un(1),"/ 2 marches :",un(2),"/ 3 marches :",un(3),"/ 35␣
↪marches :",un(35))
```

```
1 marche : 1 / 2 marches : 2 / 3 marches : 3 / 35 marches : 14930352
```

[8]:
```
print("/ -1 marche :",un(-1))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-8-5572e7380e3f> in <module>
----> 1 print("/ -1 marche :",un(-1))

<ipython-input-3-8b4a1687cf3b> in un(n)
      1 def un(n):
----> 2     assert n>= 1
      3     if n == 1 :
      4         return 1
      5     elif n == 2 :

AssertionError:
```

Cette exercice est en réalité très similaire à une suite mathématique célèbre : la suite de Fibonacci. Cette suite de Fibonacci se définit par :

$$\begin{cases} u_0 = 0 \text{ et } u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

Q7 : Écrire la fonction récursive Fib_rec(n) retournant le nième terme de la suite de Fibonacci :

[18]:
```
def Fib_rec(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else :
        return Fib_rec(n-1)+Fib_rec(n-2)
```

Essayons :

[10]:
```
print("12ieme terme de la suite de Fibonacci : ",Fib_rec(12))
```

```
12ieme terme de la suite de Fibonacci :  144
```

Pour aller plus loin... On peut montrer que la complexité vaut $\mathcal{O}(\varphi^n)$ avec φ nombre d'or, ce qui correspond aussi au nombre d'appel de la fonction à elle-même. Une fonction non récursive sera bien plus efficace :

[17]:
```
def Fib(n):
    F=[0,1]
    for i in range(n):
        F.append(F[-1]+F[-2])
    return F[n]
```

Vérifions l'éxécution :

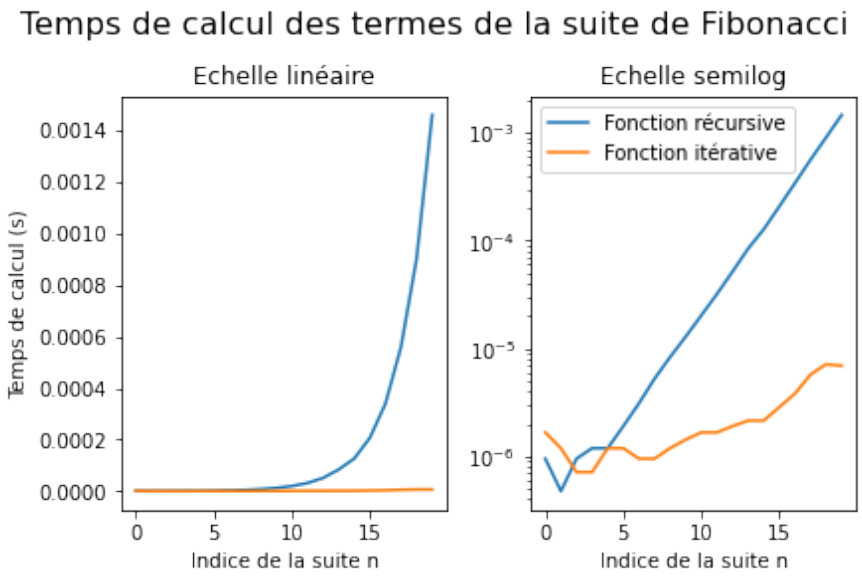
[12]:
```
print("12ieme terme de la suite de Fibonacci, méthode récursive : ",Fib_rec(12))
print("12ieme terme de la suite de Fibonacci, méthode itérative : ",Fib(12))
```

```
12ieme terme de la suite de Fibonacci, méthode récursive :  144
12ieme terme de la suite de Fibonacci, méthode itérative :  144
```

Et en terme de performance :

[19]:
```
N,Rec,It=[],[],[]
for n in range(20):
    N.append(n)
    time1=time()
    r=Fib_rec(n)
    time2=time()
    i=Fib(n)
    time3=time()
    if i==r:
        Rec.append(time2-time1)
        It.append(time3-time2)
    else :
        print('Erreur, pas le même résultat avec les deux fonctions')

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs[0].plot(N,Rec,label='Fonction récursive')
```

```
axs[0].plot(N,It,label='Fonction itérative')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('Indice de la suite n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul des termes de la suite de Fibonacci", fontsize=16)

axs[1].semilogy(N,Rec,label='Fonction récursive')
axs[1].semilogy(N,It,label='Fonction itérative')
axs[1].set_title('Echelle semilog')
axs[1].set_xlabel('Indice de la suite n')
plt.legend()
plt.show()
```



4 S'exercer à la récursion

4.1 L'essentiel

Q8 : Calcul de puissance par récursivité : écrire une fonction `puiss(a,n)` calculant a^n avec n entier naturel, par récursivité.

[20]:
```
def puiss(a,n):
    if n>0 :
        return a*puiss(a,n-1)
    else :
        return 1
```

[21]:
```
from math import pi
puiss(pi,6)-pi**6
```

[21]:
```
-1.1368683772161603e-13
```

Q9 : Un palyndrome est un mot qui peut se lire indifféremment de gauche à droite ou de droite à gauche en gardant le même sens. Écrire une fonction récursive qui teste si une chaîne de caractères est un palindrome ou non.

[27]:
```
def palind(mot):
    n = len(mot)
    if n == 0 or n == 1 :
        return True
    elif mot[0]!=mot[-1] :
        return False
    else :
        return palind(mot[1:n-1])
```

[37]:
```
palind("mot"),palind("abba"),palind("kayak")
```

[37]:
```
(False, True, True)
```

Q10 : Écrire deux fonctions récursives mutuelles `pair(N)` et `impair(N)` permettant de savoir si un nombre N entier naturel est pair, et si un nombre N entier naturel est impair.

Remarque : Ce sont des fonctions récursives mutuelles, c'est-à-dire que ***`pair(N)`*** *appelle* ***`impair(N)`****, et inversement.*

[31]:
```
def pair(N):
    if N == 0 :
        return True
    elif N == 1 :
        return False
    else :
        return impair(N-1)

def impair(N):
    if N == 0 :
        return False
    else :
        return pair(N-1)
```

[33]:
```
pair(45),pair(44), impair(0), impair(91)
```

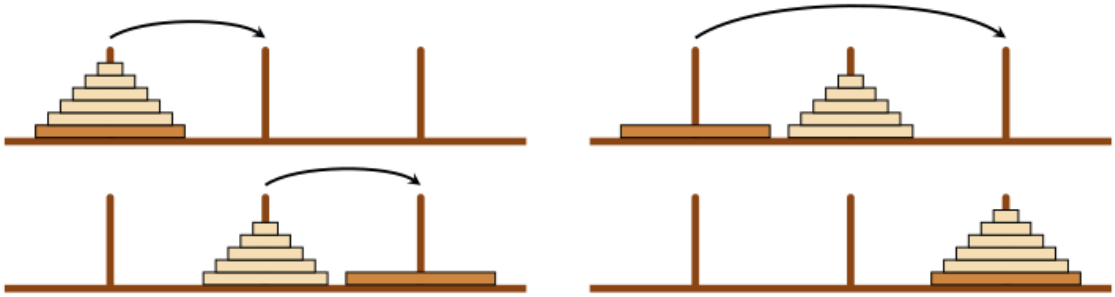
[33]:
```
(False, True, False, True)
```

4.2 Retour sur les Tours de Hanoi

Il n'est pas évident d'écrire une solution itérative pour ce problème, alors qu'une solution récursive est très adaptée. En effet, pour déplacer les n disques de A à C, il "suffit" de :

- déplacer les $n-1$ disques du dessus de A à B ;
- déplacer le nème disques de A à C ;

- déplacer les n − 1 disques de B à C. Reste à savoir comment déplacer n-1 disques… Heureusement, pour déplacer n-1 disques, on peut ré-appliquer la méthode précédente, et ainsi de suite jusqu'à arriver à déplacer 1 disque ! Il s'agit donc bien d'une méthode récursive.



Bien entendu, si n = 0, il n'y rien à faire !

Q11 : Créer une fonction `Hanoi(n)` retournant le nombre d'opérations nécessaires pour résoudre le problème des Tours de Hanoi, avec n disques.

On pourra en profiter pour démontrer par récurrence que ce nombre est $2^n - 1$. Reste à se convaincre que c'est bien le nombre minimum d'opérations…

[25]:
```
def Hanoi(n):
    if n==0:
        return 0
    else :
        # on compte le nombre de déplacement : Hanoi(n-1) pour déplacer les n-1 disques les plus petits sur une autre tour,
        # 1 pour déplacer le plus grand disque et à nouveau
        # Hanoi(n-1) pour déplacer les n-1 disques les plus petits sur le grand disque
        return 2*Hanoi(n-1)+1
```

Et donc, pour résoudre le problème :

[26]:
```
n=6
print("Pour résoudre les Tours de Hanoi avec ",n," disques, il faut ",Hanoi(n)," opérations")
```

```
Pour résoudre les Tours de Hanoi avec  6  disques, il faut  63  opérations
```

4.3 Pour aller plus loin…

Suite l'exercice précédent : nous allons faire en sorte d'afficher les différentes étapes de la résolution. On utilisera les fonctions déjà écrites dans le fichier *I3_Rec.py*, et on rappelle que `f_animation(Tours)` affiche le système des Tours de Hanoi.

Q12 : Compléter la fonction `Hanoi_aff(Tours,n,i,j)` récursive permettant de déplacer les n disques supérieurs de la tour i vers la tour j, tout en affichant les tours à chaque opération. `Tours` est la liste des trois piles représentant les trois tours.

[5]:
```
from matplotlib.patches import Rectangle

# Fonctions utiles

def creer_pile():
    return []

def empiler(P,v):
    P.append(v)

def depiler(P):
    if len(P)==0:
        raise ValueError("Erreur : pile vide")## permet d'afficher un␣
 ↪magnifique message rouge d'alerte dans le shell
    else :
        P.pop()
    return(P)

def taille(P):
    return len(P)

def est_vide(P):
    return len(P)==0

def sommet(P):
    if est_vide(P)==1:
        raise ValueError("Erreur : pile vide")## permet d'afficher un␣
 ↪magnifique message rouge d'alerte dans le shell
    else :
        return P[-1]

def creation_tours(n): # Permet de créer la situation initiale
    A,B,C=creer_pile(),creer_pile(),creer_pile()
    for i in range(n):
        empiler(A,n-i)
    return [A,B,C]

# Pour l'affichage des tours...

def f_init_schema(Pouces):
    fig = plt.figure(1,figsize=(Pouces,Pouces)) # Définition d'une␣
 ↪fenêtrenuméro 1 de taille 20' X 20' dans laquelle l'animation aura lieu
    return fig

def f_affiche_schema(fig,n):
    fig.show() # Affichage du schéma cinématique
```

```
    plt.gca().set_aspect('equal', adjustable='box') # L'option Adjustable␣
 ↪permet d'éviter des redimensionnements intempestifs lors de l'évolution de␣
 ↪la géométrie
    plt.xlim(-n,5*n)
    plt.ylim(0,n+1)

def f_clear_schema():
    plt.clf() # Effacement de la figure en vue de la prochaine

def f_barreaux(n): # Ajout du segment d'une pièce à la figure du tracé de␣
 ↪schéma cinématique
    for i in range(3):
        X1 = 2*n * i
        Y1 = 0
        X2 = X1
        Y2 = n
        plt.plot([X1,X2],[Y1,Y2],color= "b", linewidth = 10 )

def f_rectangle(fig,x,y,l,h): # Ajout d'un étage à fig en précisant ses␣
 ↪coordonnées (x,y), et ses dimensions (l,h)
    currentAxis = plt.gca()
    currentAxis.add_patch(Rectangle((x,y),l,h,facecolor="grey"))

def f_etage(fig,Pile,Etage,Valeur,n): # Détermination des informations de␣
 ↪l'étage Etage de la pile Pile pour l'ajouter à fig
    Largeur = Valeur
    Hauteur = 1
    X_Med = 2*n * Pile - Largeur / 2
    Y_Med = Etage
    f_rectangle(fig,X_Med,Y_Med,Largeur,Hauteur)

def f_etages(fig,Piles,n): # Ajout de tous les étages de chaque tour à fig
    for i in range(3):
        Pile = Piles[i]
        t = len(Pile)
        for j in range(t):
            Etage = j
            Valeur = Pile[j]
            f_etage(fig,i,Etage,Valeur,n)

def f_animation(LP): # Affichage de la figure avec toutes les informations
    fig = f_init_schema(20)
    f_clear_schema()
    n = len(LP[0])+len(LP[1])+len(LP[2])
    f_barreaux(n)
    f_etages(fig,LP,n)
    f_affiche_schema(fig,n)
```

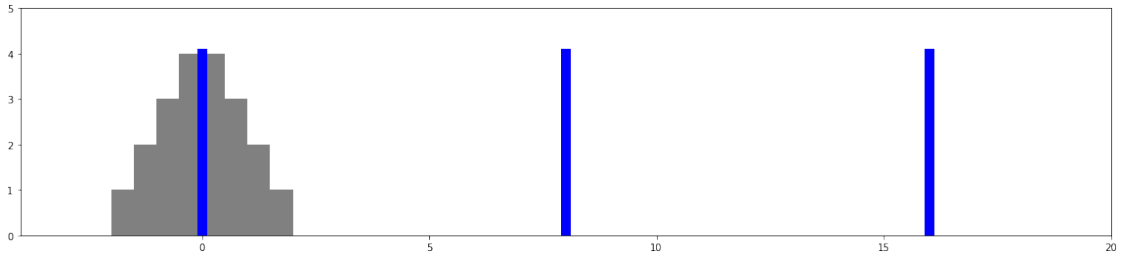
```
    plt.pause(0.5)
```

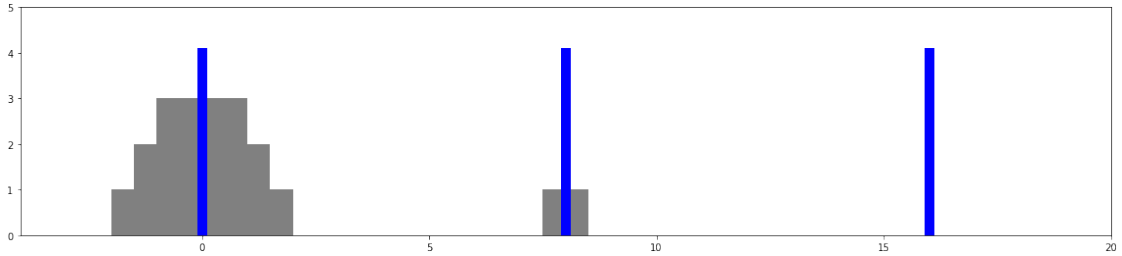
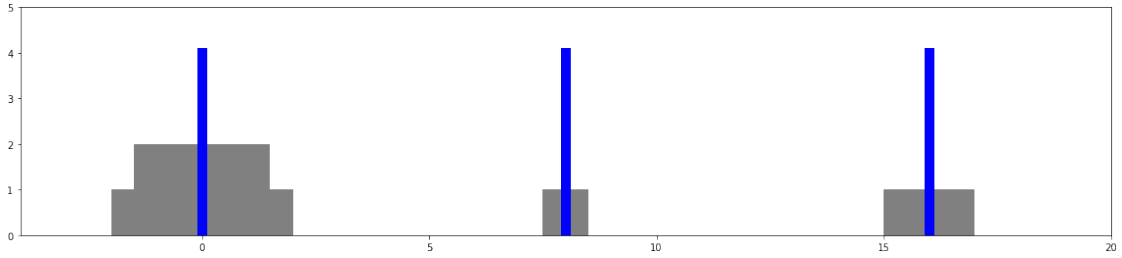
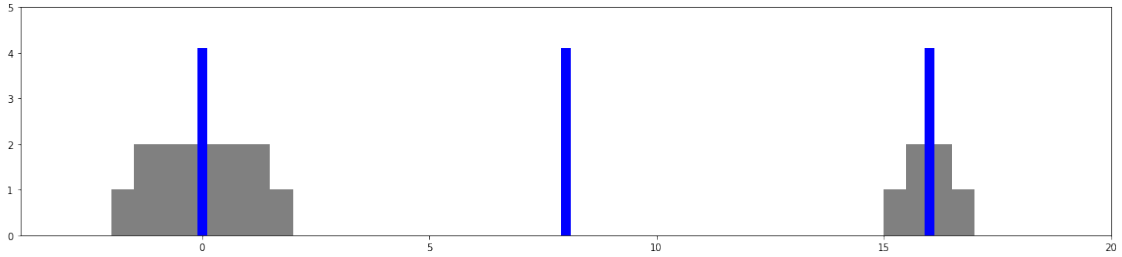
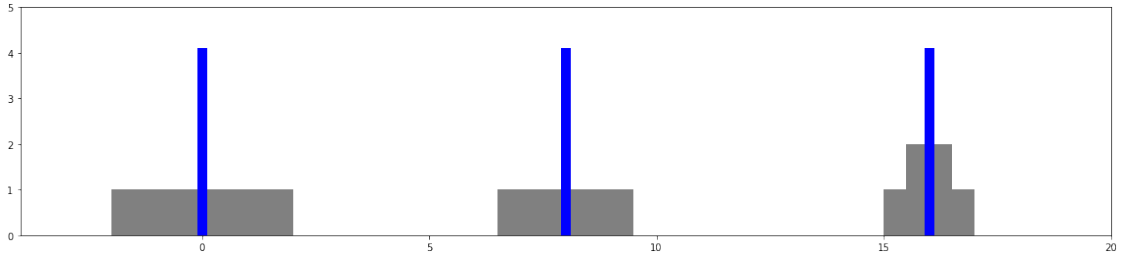
[6]:
```
def Hanoi_aff(Tours,n,i,j):

    if n==1 : # S'il n'y a qu'un disque sur la tour i
        empiler(Tours[j],sommet(Tours[i]))
        depiler(Tours[i])
        f_animation(Tours)
    else :
        # cherchons la tour vide k, qui n'est pas la tour j ni la i :
        k=(j+1)%3 # on fait un essai : on regarde la tour située à droite de la␣
 ↪tour j
        if k==i : # si c'est aussi la tour i ...
            k=(k+1)%3 # on prend la suivante !

        # On déplace les n-1 disques sur la tour k
        Hanoi_aff(Tours,n-1,i,k) #n-1 opérations

        # On déplace le nième disque sur la tour j (désormais sommet de cette␣
 ↪tour)
        Hanoi_aff(Tours,1,i,j) #1 opération

        # On redéplace les n-1 disques sur la tour j
        Hanoi_aff(Tours,n-1,k,j) #n-1 opérations
```

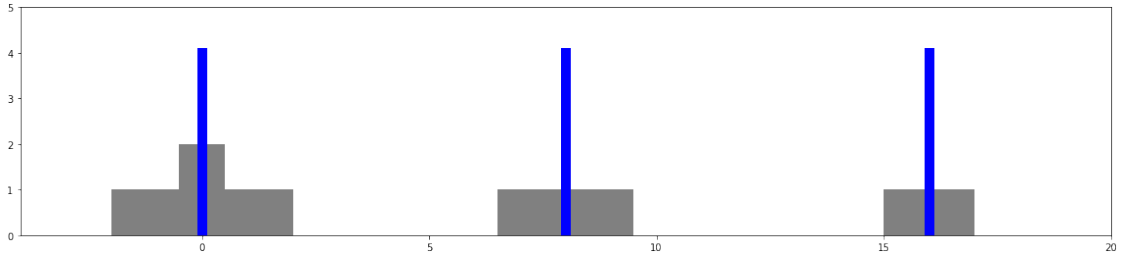
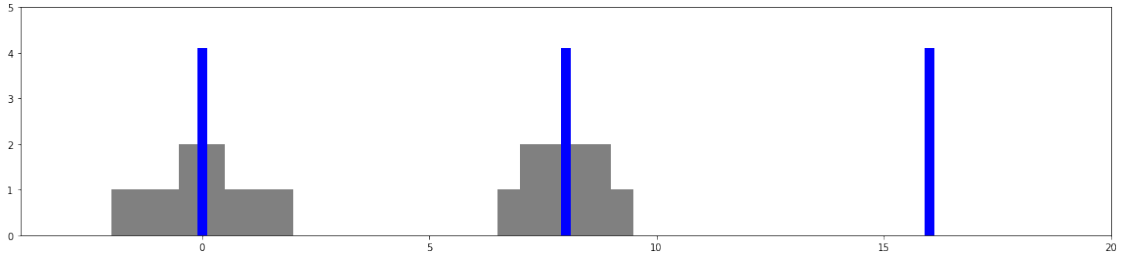
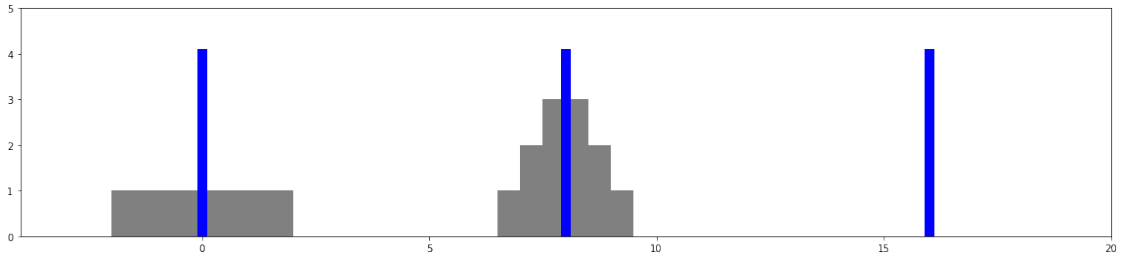
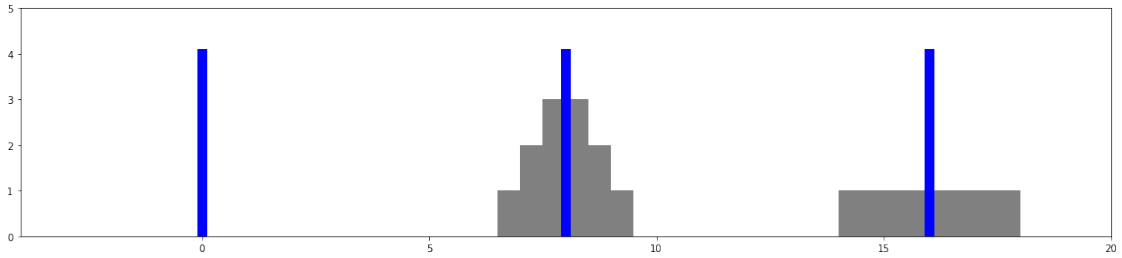
Vérifions le bon fonctionnement :

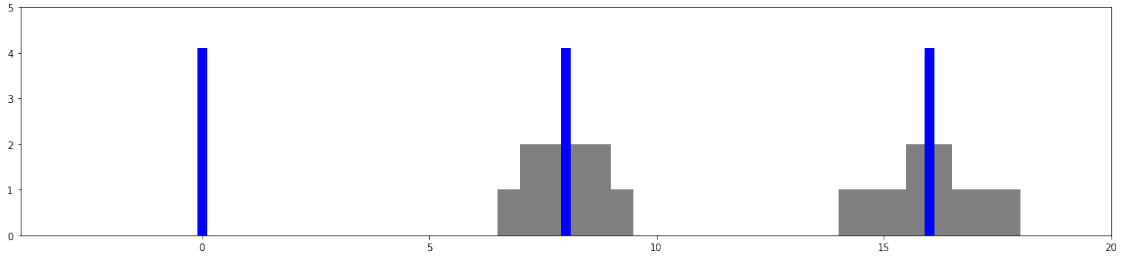
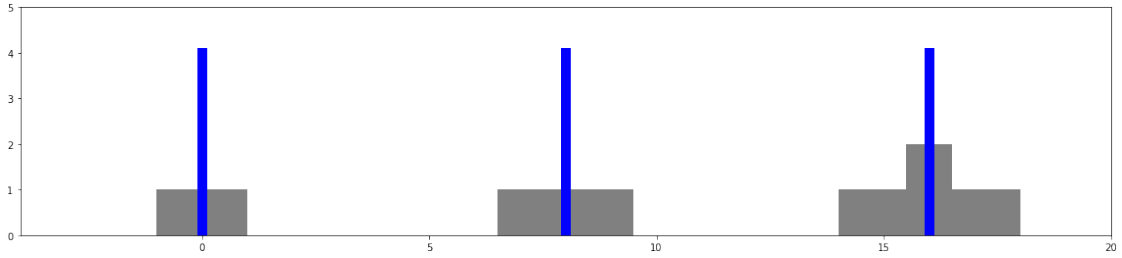
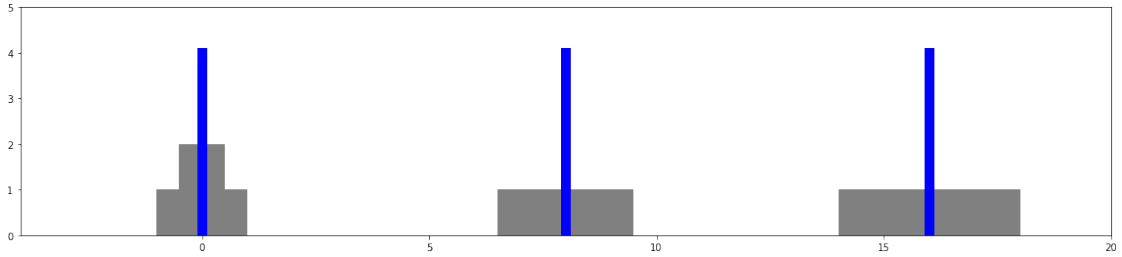
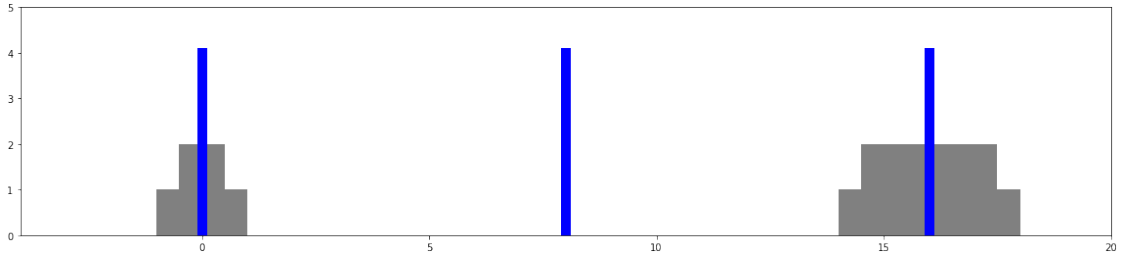
[7]:
```
n=4
T=creation_tours(n)
f_animation(T)
Hanoi_aff(T,n,0,2)
```

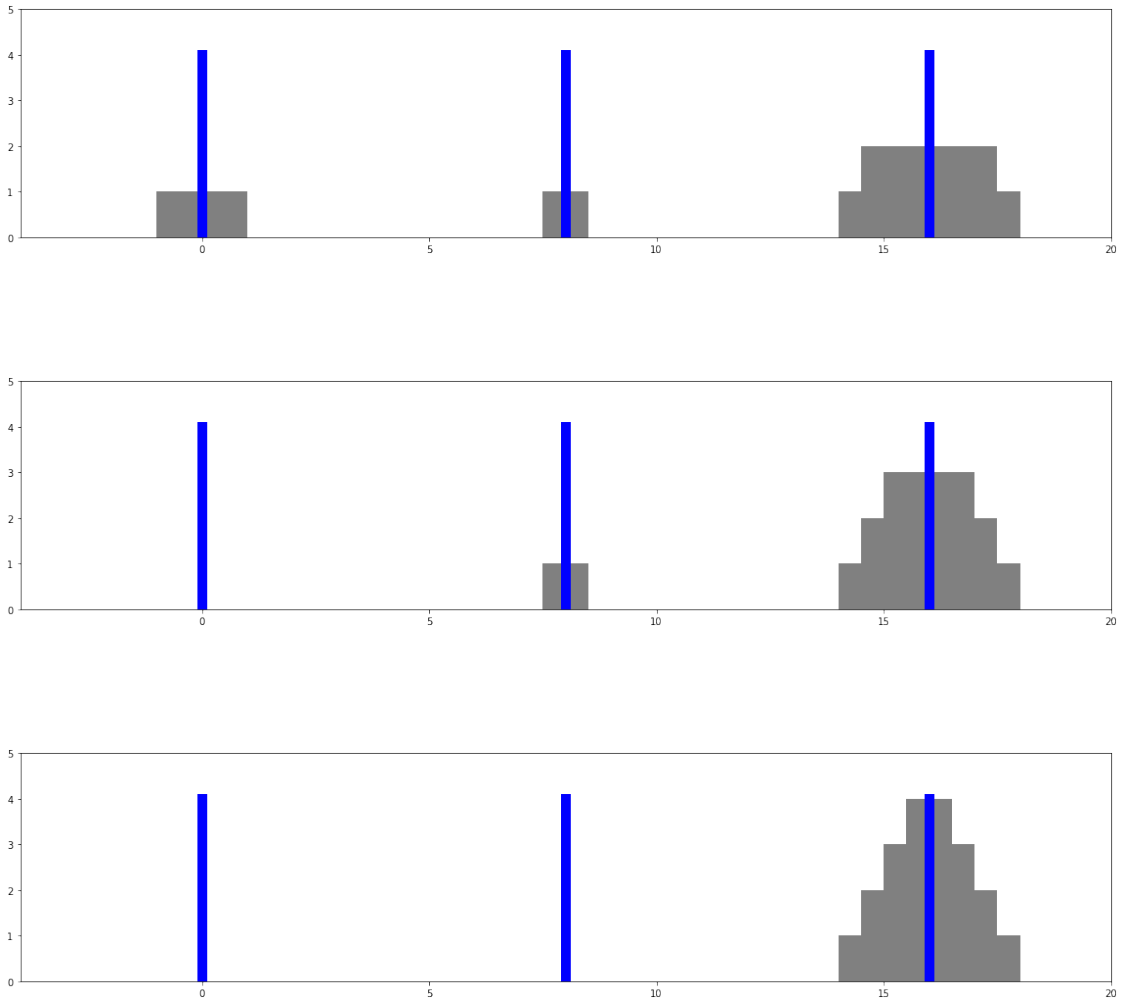
```
c:\users\cpgetpc-profs\miniconda3\lib\site-packages\matplotlib\figure.py:418:
UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the
figure
  "matplotlib is currently using a non-GUI backend, "
```





Q13 : Proposer une fonction permettant de répondre à la question suivante : Combien de personnes peut-on enrichir si, disposant de beaucoup d'argent, on décide de le distribuer de la manière suivante : 1€ à la première personne qu'on croise, 2€ à la seconde, 3€ à la troisième, etc ?

[1]:
```
def distrib(arg):
    if arg==0: # terminaison
        return 0
    n=distrib(arg-1) #nb de personnes qu'on peut enrichir avec 1€ de moins
    arg_d=n*(n+1)/2 # argent déjà donné
    if n+1==arg-arg_d: # si on peut donner de l'argent à une nouvelle personne
        return n+1
    else :
        return n

def distrib_b(arg):
    if arg == 1:
```

```
        return 1,0
    N,reste = distrib_b(arg-1)
    if reste >= N :
        N+=1
        reste = 0
    else :
        reste += 1
    return N,reste
```

Vérifions :

[5]:
```
arg=20
print("On peut enrichir ",distrib(arg)," personnes avec ",arg,"€.")
print("On peut enrichir ",distrib_b(arg)[0]," personnes avec ",arg,"€, et il␣
 ↪reste ",distrib_b(arg)[1]," €.")
arg=21
print("On peut enrichir ",distrib(arg)," personnes avec ",arg,"€.")
print("On peut enrichir ",distrib_b(arg)[0]," personnes avec ",arg,"€, et il␣
 ↪reste ",distrib_b(arg)[1]," €.")
```

```
On peut enrichir  5  personnes avec  20 €.
On peut enrichir  5  personnes avec  20 €, et il reste  5  €.
On peut enrichir  6  personnes avec  21 €.
On peut enrichir  6  personnes avec  21 €, et il reste  0  €.
```

Q14: Ecrire une fonction, qui prend comme argument une liste représentant les pièces d'argent qu'on possède, et une somme qu'on doit payer. Cette fonction retournera `True` si on peut faire l'appoint, et `False` sinon. On pourra utiliser la fonction `L.sort()` permettant de trier une liste L du plus petit au plus grand élément.

[69]:
```
def appoint(somme, L):
    L.sort()
    if somme == 0:
        return True # on peut toujours payer 0 euro
    elif L == [] or L[0]>somme:
        return False # on ne peut rien payer sans argent (sauf 0 euro) et si on␣
 ↪ne possède que des pièces de valeur supérieure à la somme
    else:
        if appoint(somme-L[0],L[1:]): # L'appoint est possible en utilisant la␣
 ↪plus petite pièce
            return True
        else :
            return appoint(somme,L[1:]) # On regarde si l'appoint est possible␣
 ↪sans utiliser cette plus petite pièce
```

Vérifions :

[70]:
```
S=33
L=[10,4,5,2,1,2,3,4,2,6]
```

```
Lb=[8,4,8,2,2,2,4,4,2,6]
print("Appoint possible ? ",appoint(S,L))
print("Appoint possible ? ",appoint(S,Lb))
```

```
Appoint possible ?  True
Appoint possible ?  False
```

Q 15 :

Le PGCD est le plus grand commun diviseur. Par exemple, on a :

- PGCD(2,3) = 1 car 1 est le plus grand nombre qui divise (au sens de la division entière) à la fois 2 et 3.
- PGCD(15,10) = 5 car 5 est le plus grand nombre qui divise (au sens de la division entière) à la fois 15 et 10.

On note comme en Python $a\%b$ le reste de la division euclidienne de l'entier a par l'entier non nul b (on le lit aussi a modulo b). L'algorithme d'Euclide permet le calcul du PGCD de deux entiers naturels en utilisant les propriétés suivantes :

$$\forall a \in \mathbb{N}, PGCD(a,0) = a \text{ et } \forall b \in \mathbb{N}^*, PGCD(a,b) = PGCD(b, a\%b)$$

Formaliser l'algorithme et le programmer en Python (donner une version récursive).

[41]:
```
def PGCD(a,b):
    if b == 0 :
        return a
    else:
        return PGCD(b,a%b)
```

[42]:
```
PGCD(2,3),PGCD(10,15)
```

[42]: (1, 5)