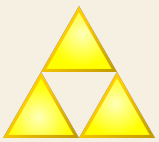
Récursivité

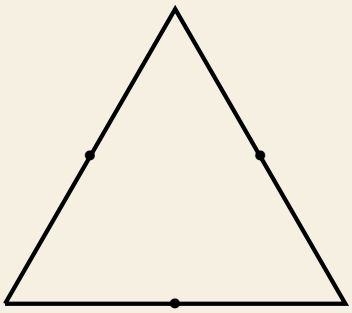
Exercice 1 : La Triforce

La légende raconte que celui qui touchera la Triforce réunie pourra réaliser tous ses désirs.

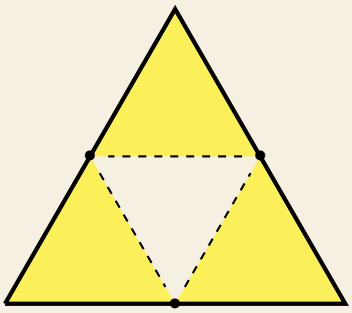




Ce symbole présent dans la licence **Zelda** permet de rendre à Hyrule son apparence d'origine et vaincre ainsi définitivement Ganon. Chaque triangle représente les qualités élémentaires des guerriers : la force, la sagesse et le courage.

Pour construire la Triforce, on découpe un triangle équilatéral en quatre triangles en reliant les milieux de chacun des côtés comme ci-dessous. On retire ensuite le triangle central.



Triforce initiale

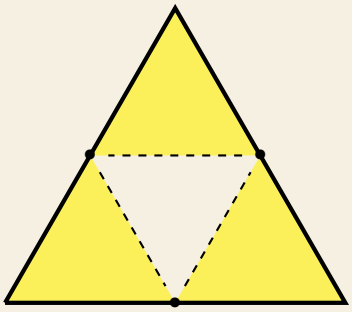


Triforce après une découpe

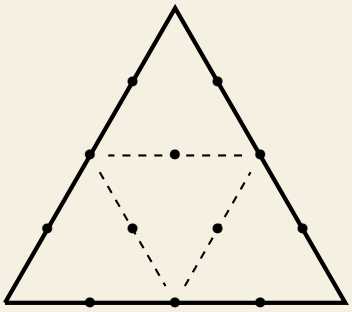
1. On décide de faire évoluer la Triforce grâce à la règle suivante :

« **Transformer chaque triangle de la Triforce en Triforce.** »

Représenter cette évolution ci-dessous :



Triforce après une découpe



Triforce après deux découpes

2. On s'intéresse à présent à l'aire de la Triforce obtenue après un certain nombre de découpes. On note $u(n)$ cette aire où n est le nombre de découpes. Par exemple :

- $u(0)$ correspond à l'aire du triangle équilatéral initial ;
- $u(1)$ correspond à l'aire de la Triforce de Zelda ;
- $u(2)$ correspond à l'aire de la nouvelle Triforce ;
- etc.

D.W.

a. On suppose que le triangle équilatéral initial est de côté 1.
Déterminer la valeur de $u(0)$.

b. Expliquer pourquoi on a les relations

$$u(1) = \frac{3}{4}u(0) \qquad \text{et} \qquad u(2) = \frac{3}{4}u(1)$$

3. On souhaite implanter en Python la fonction u définie ci-dessus.
Compléter le code proposé :

```
def u(n:int) -> float:
    """ Renvoie l'aire de la Triforce après n découpes """
    if n == ... :
        res = ...
    else:
        res = ...
    return res
```

Cours

On appelle **fonction récursive** toute fonction qui fait appel à elle-même dans son implantation.
Une telle fonction est constituée de **deux blocs** :

- ❑ **Le cas de base** : c'est une valeur particulière propre à la fonction pour lequel **il ne doit pas** y avoir d'appel récursif ;
- ❑ **Les autres cas** : ce sont les autres valeurs pour lesquelles la fonction réalise un appel récursif.

Attention : il se peut que le cas de base soit constitué de plusieurs cas.

Par exemple, pour obtenir la somme d'une liste à l'aide d'une fonction récursive, on pourrait écrire :

```
def somme(L:list) -> int:
    """ Renvoie la somme d'une liste d'entiers """
    if L == []:
        res = 0
    else:
        res = L[0] + somme(L[1:])
    return res
```

Le cas de base est identifié comme étant la condition `L == []`.
L'appel récursif se fait lors de l'instruction `somme(L[1:])`.

Lors de l'instruction `somme([12, 5, 3, 8])`, les appels récursifs se représentent par :

```
-> somme([12, 5, 3, 8])
... -> somme([5, 3, 8])
...... -> somme([3, 8])
......... -> somme([8])
............ -> somme([])
............ <- 0
......... <- 8
...... <- 11
... <- 16
<- 28
```

L'évaluation de l'instruction `somme([12, 5, 3, 8])` **reste en suspend** tant qu'il n'y a pas de valeur de retour pour l'instruction `somme([5, 3, 8])`. Elle même reste en suspend tant que l'instruction `somme([3, 8])` n'a pas été évaluée etc. On parle alors de **pile d'exécution**.

Exercice 2 : Opérations mathématiques

Lorsqu'on effectue une addition, on utilise les tables apprises lors de nos passages en petites classes. Avant ces classes, les enfants ajoutent autant de fois 1 que nécessaire. Par exemple, pour calculer $8+3$, un enfant effectue les calculs suivants :

$$8+3=9+2=10+1=11$$

que l'on peut illustrer avec la technique des doigts :

8 9 10 11

La récursivité intervient dans la démarche au moment où pour faire la somme entre 8 et 3, j'effectue la somme entre 9 et 2. Pour effectuer la somme entre 9 et 2, j'effectue la somme entre 10 et 1. Pour réaliser cette somme, je fais la somme entre 11 et 0. Mais je sais faire cette somme.

1. a. Identifier le cas de base pour réaliser une addition.

b. Compléter le code suivant :

```python
def addition(a:int, b:int) -> int:
    """ Effectue une addition récursivement """
    if b == ... :
        res = ...
    else:
        res = ...
    return res
```

c. Donner les appels récursifs lors de l'instruction `addition(1987, 3)`.

2. De la même façon quand on effectue une multiplication, on ajoute autant de fois que nécessaire le multiplicateur. Par exemple pour effectuer l'opération 8×3, on calcule naïvement :

$$8\times 3=8+8\times 2=16+8\times 1=24+8\times 0=24$$

que l'on peut illustrer avec la technique des doigts :

0 8 16 24

a. Identifier le cas de base pour réaliser une multiplication récursivement.

b. Compléter le code suivant :

```python
def multiplication(a:int, b:int) -> int:
    """ Effectue une multiplication récursivement """
    if b == ... :
        res = ...
    else:
        res = ...
    return res
```

c. Donner les appels récursifs lors de l'instruction `multiplication(12, 4)`

Exercice 3 : Somme des chiffres

On souhaite définir une version récursive qui fournit la somme des chiffres d'un entier passé en paramètre. Avec l'instruction `somme_chiffres(1987)`, on a obtenu les appels récursifs :

```
-> somme_chiffres(1987)
... -> somme_chiffres(198)
...... -> somme_chiffres(19)
......... -> somme_chiffres(1)
......... <- 1
...... <- 10
... <- 18
<- 25
```

1. En calquant l'exemple, donner les appels récursifs lors de l'instruction `somme_chiffres(7102)`

2. a. Quels sont les cas de base de cette fonction récursive ?

b. Proposer un code pour cette fonction.

Exercice 4 : Divisibilité par trois

On rappelle le critère de divisibilité par trois :

> Un nombre est divisible par trois si la somme de ses chiffres est divisible par trois.

Par exemple 444 est divisible par trois car la somme de ses chiffres vaut 12.
12 est-il divisible par trois ? La somme des chiffres de 12 vaut 3 qui est divisible par trois.

12 est donc divisible par trois. Donc la somme des chiffres de 444 également.
444 est ainsi divisible par trois.

1. Donner les appels récursifs lors de l'instruction `div_par_trois(7345294684572)`

2. a. Donner les quatre cas de base.

b. Proposer une implantation récursive pour cette fonction.

Exercice 5 : Tri rapide

On considère la fonction `tri_rapide(L:list)-> list` définie par :

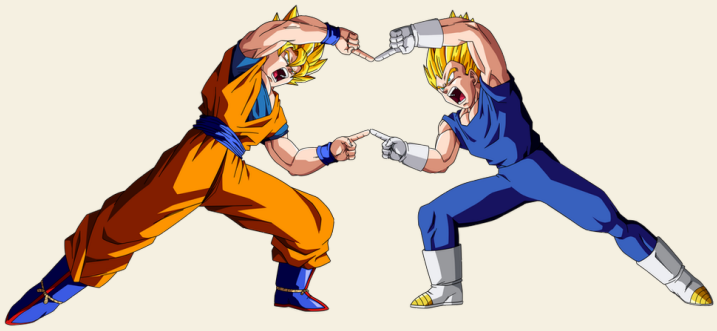
```
def tri_rapide(L:list) -> list:
    n = len(L)
    if n <= 1:
        res = L
    else:
        pivot  = L[0]
        gauche = [x for x in L[1:] if x <= pivot]
        droite = [x for x in L[1:] if pivot < x]
        res = tri_rapide(gauche) + [pivot] + tri_rapide(droite)
    return res
```

Remarque : L'opérateur + entre deux listes effectue une concaténation de ces deux listes.

1. Donner les appels récursifs lors de l'instruction `tri_rapide([18, 13, 24, 10, 15, 34, 31, 45])`.
Indication : on pourra écrire `tr` à la place de `tri_rapide`.

Exercice 6 : Fusion de deux listes

On s'intéresse dans cet exercice à la fusion de deux listes triées dans l'ordre croissant.



Grâce à cette fusion, on cherche à obtenir une liste triée dans l'ordre croissant.

Par exemple, l'instruction `fusion([1, 8, 12, 15], [3, 5, 10, 20, 30])` doit renvoyer la liste

`[1, 3, 5, 8, 10, 12, 15, 20, 30]`

L'algorithme de fusion entre deux listes `L1` et `L2` est donné ci-dessous :

- ❏ si une des deux listes à fusionner est vide, on renvoie l'autre liste ;
- ❏ sinon, on compare le premier élément de `L1` avec le premier élément de `L2` :
  - si `L1[0] <= L2[0]`, on renvoie la liste `[L1[0]]` suivi de la fusion entre le reste de `L1` et `L2` ;
  - sinon, on renvoie la liste `[L2[0]]` suivi de la fusion entre `L1` et le reste de `L2`.

1. Préciser les cas de base de la fonction `fusion(L1:list, L2:list) -> list`.

2. Proposer une implantation récursive.

3. Compléter le code de la fonction `fusion` en version itérative ci-dessous :

```
def fusion_ite(L1:list, L2:list) -> list:
    """ Renvoie une fusion de L1 et L2 """
    L = []
    while L1 != ... and L2 != ... :
        if L1[0] <= L2[0]:
            ...
        else:
            ...
    return L + L1 + L2
```

Indication : on pourra utiliser la méthode `pop` qui renvoie (et retire) un élément d'une liste selon l'indice fourni.

4. On propose la fonction `foo(L:list) -> list` suivante :

```
def foo(L:list) -> list:
    n = len(L)
    if n == 1:
        res = L
    else:
        res = fusion(foo(L[:n//2]), foo(L[n//2:]))
    return res
```

Donner les appels récursifs de l'instruction `foo([14, 2, 23, 7, 45, 18, 20])`