Algorithmes récursifs

Une fonction récursive est une fonction qui s'appelle elle-même. C'est un concept puissant de l'informatique : certaines tâches compliquées s'obtiennent à l'aide d'une fonction récursive simple. La récursivité est l'analogue de la récurrence mathématique.

Cours 1 (Récursivité (début))**.**
L'exemple incontournable pour commencer est le calcul des factorielles. On rappelle que

$$n! = 1 \times 2 \times 3 \times 4 \times \cdots \times (n-1) \times n.$$

Par exemple $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

Algorithme par une boucle.

```
def factorielle_classique(n):
    f = 1
    for k in range(2,n+1):
        f = f*k
    return f
```

Algorithme récursif.

```
def factorielle(n):
    if n == 1:      # Cas terminal
        f = 1
    else:           # Cas général
        f = factorielle(n-1)*n
    return f
```

L'algorithme classique met en œuvre une boucle « pour » et une variable f qui vaut tour à tour $f = 1$, puis $f = 1 \times 2$, $f = 1 \times 2 \times 3$,... jusqu'à $f = 1 \times 2 \times 3 \times \cdots \times n = n!$
L'algorithme récursif est différent, regarde bien le code de la fonction `factorielle()` : dans ces lignes de code une instruction fait appel à la fonction `factorielle()` elle-même. L'algorithme est basé sur la relation de récurrence :

$$n! = (n-1)! \times n$$

Donc pour calculer $n!$ il suffit de savoir calculer $(n-1)!$ mais pour calculer $(n-1)!$ il suffit de savoir calculer $(n-2)!$ (car $(n-1)! = (n-2)! \times (n-1)$)... Quand-est-ce que cela se termine ? Lorsque il faut calculer $1!$ alors par définition on sait $1! = 1$.
Pour bien comprendre ce qu'il se passe, il est conseillé d'afficher un message à chaque appel de la fonction. Voici une version modifiée de notre fonction récursive :

```
def factorielle(n):
    if n == 1:      # Cas terminal
        print("Cas terminal. Appel de la fonction avec n =",n)
        f = 1
    else:           # Cas général
```

```
        print("Cas général.  Appel de la fonction avec n =",n)
        f = factorielle(n-1)*n
    return f
```

La commande `factorielle(10)` renvoie la valeur $10! = 3\,628\,800$. Et au passage voici l'affichage à l'écran produit par cette commande :

```
Cas général.  Appel de la fonction avec n = 10
Cas général.  Appel de la fonction avec n = 9
Cas général.  Appel de la fonction avec n = 8
Cas général.  Appel de la fonction avec n = 7
Cas général.  Appel de la fonction avec n = 6
Cas général.  Appel de la fonction avec n = 5
Cas général.  Appel de la fonction avec n = 4
Cas général.  Appel de la fonction avec n = 3
Cas général.  Appel de la fonction avec n = 2
Cas terminal. Appel de la fonction avec n = 1
```

C'est donc un peu comme un compte à rebours, pour calculer 10! on demande le calcul de 9! qui nécessite le calcul de 8!… lorsque l'on arrive au calcul de 1! on renvoie 1, cela débloque la valeur de 2! donc celle de 3!… et à la fin on obtient la valeur de 10!

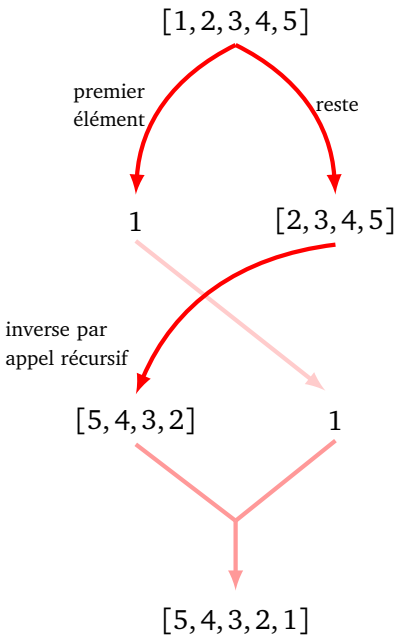
Activité 1 (Pour bien commencer.)**.**

Objectifs : programmer ses premiers algorithmes récursifs.

Recommandation. Comme la récursivité est un concept difficile à appréhender, n'hésite pas à afficher les étapes intermédiaires. Par exemple, tu peux commencer chaque fonction par une instruction du type :

`print("Appel de la fonction avec n =", n)`

1. La somme des carrés des premiers entiers est :
$$S_n = 1^2 + 2^2 + 3^2 + \cdots + (n-1)^2 + n^2.$$
   (a) Programme une fonction `somme_carres_classique(n)` qui calcule cette somme S_n à l'aide d'une boucle.
   (b) En utilisant la formule de récurrence :
$$S_1 = 1 \qquad \text{et} \qquad S_n = S_{n-1} + n^2 \quad \text{pour } n \geqslant 2,$$
   programme une fonction `somme_carres(n)` qui calcule S_n par un algorithme récursif. Le cas terminal correspond à $n = 1$ pour lequel $S_1 = 1$. Le cas général d'un $n \geqslant 2$, correspond au calcul de la somme S_n, écrite sous la forme $S_n = S_{n-1} + n^2$. Donc pour calculer S_{n-1} tu effectues un appel récursif à `somme_carres(n-1)`.
2. Programme une fonction récursive `inverser(liste)` qui inverse l'ordre des éléments d'une liste. Par exemple `inverser([1,2,3,4,5])` renvoie `[5,4,3,2,1]`. Le principe à suivre est le suivant : on extrait le premier élément ; on inverse le reste de la liste ; enfin on rajoute le premier élément à la fin de liste obtenue.

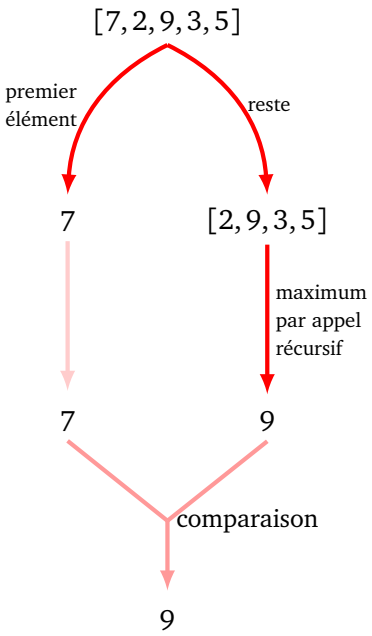


Voici l'algorithme en détails :

Algorithme.

- — Entête : `inverser(liste)`
  — Entrée : une `liste` $[x_0, x_1, \ldots, x_{n-1}]$ de longueur n.
  — Sortie : la liste inversée $[x_{n-1}, x_{n-2}, \ldots, x_1, x_0]$.
  — Action : fonction récursive.
- *Cas terminal.* Si la longueur n de la liste est 1 (ou 0) : renvoyer la liste sans modification (il n'y a rien à inverser).
- *Cas général.*
  — On note x_0 le premier élément de `liste`.
  — On note `fin_liste` le reste de la liste (la liste sans x_0).
  — On effectue un appel récursif `inverser(fin_liste)` qui renvoie une liste `fin_liste_inverse`.
  — On ajoute à ce résultat l'élément x_0 en queue de liste.
  — On renvoie cette liste.

3. Programme une fonction récursive `maximum(liste)` qui renvoie le maximum d'une liste. Par exemple `maximum([7,2,9,3,5])` renvoie 9.



Utilise le principe suivant :

- *Cas terminal.* Si la liste ne contient qu'un seul élément, alors le maximum est cet élément.
- *Cas général.*
  - On note x_0 le premier élément de la liste.
  - On calcule la maximum du reste de la liste par un appel récursif. On note M' ce maximum.
  - On compare x_0 et M' : si $x_0 > M'$ alors on pose $M = x_0$, sinon on pose $M = M'$.
  - On renvoie M.

4. Programme une fonction récursive `binaire(n)` qui pour un entier n donné, renvoie son écriture binaire sous la forme d'une chaîne de caractères. Par exemple `binaire(23)` renvoie `'10111'`.

   Voici le principe :

   - *Cas terminaux.* Si $n = 0$ renvoyer `'0'`, si $n = 1$ renvoyer `'1'`.
   - *Cas général.*
     - Calculer l'écriture binaire de $n//2$ par un appel récursif.
     - Si n est pair rajouter `'0'` à la fin de la chaîne renvoyée.
     - Sinon rajouter `'1'`.
     - Renvoyer la chaîne obtenue.

Cours 2 (Récursivité (suite))**.**

Voyons un autre exemple de fonction récursive. On souhaite déterminer si un mot est un palindrome ou pas, c'est-à-dire s'il peut se lire dans les deux sens comme **RADAR** ou **ELLE**.

Voici comment on décide de procéder :

- *Cas terminal numéro 1.* Si le mot contient zéro ou une lettre, c'est un palindrome !
- *Cas terminal numéro 2.* Si la première lettre et la dernière lettre sont différentes, alors le mot n'est pas un palindrome (peu importe les lettres du milieu).
- *Cas général.* Dans le cas général on sait que le mot contient au moins deux lettres (sinon c'est le cas terminal 1) et que la première et dernière lettre sont identiques (sinon c'est le cas terminal 2). Notre mot est donc un palindrome si et seulement les lettres « du milieu » forment un palindrome.

Voici trois exemples avec la réponse à la question « Est-ce que le mot donné est un palindrome ? » :

RADAR → **ADA** → **D** donc « Vrai »

SERPES → **ERPE** → **RP** donc « Faux »
ELLE → **LL** → " " donc « Vrai »

Voici la fonction correspondante :

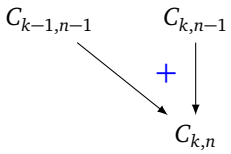
```
def est_palindrome(mot):
    n = len(mot)

    # Cas terminal 1
    if n <= 1:
        return True

    # Cas terminal 2
    if mot[0] != mot[n-1]:
        return False

    # Cas général
    mot_milieu = mot[1:n-1]
    ok_palind = est_palindrome(mot_milieu)
    return ok_palind
```

Activité 2 (Fibonacci, Pascal & Cie)**.**

Objectifs : étudier des cas de récursivité plus compliqués.

1. **Fibonacci.**
   La suite de Fibonacci est définie par une formule de récurrence qui dépend des deux termes précédents.
   $$F_0 = 0 \quad \text{et} \quad F_1 = 1 \qquad \text{puis} \quad F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geqslant 2.$$
   Vérifie que les premiers termes sont :
   $$F_0 = 0 \quad F_1 = 1 \quad F_2 = 1 \quad F_3 = 2 \quad F_4 = 3 \quad F_5 = 5 \quad F_6 = 8 \quad \dots$$
   Programme une fonction récursive `fibonacci(n)` qui renvoie F_n.
   - Les cas terminaux sont pour $n = 0$ et $n = 1$.
   - Pour le cas général, il faut faire deux appels récursifs : un appel de la fonction au rang $n-1$ qui renvoie F_{n-1} et un appel au rang $n-2$ qui renvoie F_{n-2}. Ensuite tu renvoies la somme de ces deux nombres.

   Commentaires. Cet algorithme n'est pas efficace car il est très lent pour $n \geqslant 30$. Essaie de comprendre pourquoi. Pour répondre à cette question tu peux afficher « Tiens, je calcule encore F_2 ! » à chaque appel de la fonction pour lequel $n = 2$.
2. **Coefficients du binôme.**
   Les ***coefficients du binôme de Newton*** $C_{k,n}$ sont définis pour $0 \leqslant k \leqslant n$. Ils se calculent par une formule de récurrence :
   $$C_{k,n} = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n \\ C_{k-1,n-1} + C_{k,n-1} & \text{sinon.} \end{cases}$$
   Un coefficient s'obtient donc comme la somme de deux autres. Ce coefficient $C_{k,n}$ est habituellement noté $\binom{n}{k}$ et lu « k parmi n ».



Ce sont aussi les coefficients qui apparaissent dans le développement de $(a+b)^n$. Par exemple :

$$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$$

et on a

$$C_{0,4} = 1 \qquad C_{1,4} = 4 \qquad C_{2,4} = 6 \qquad C_{3,4} = 4 \qquad C_{4,4} = 1$$

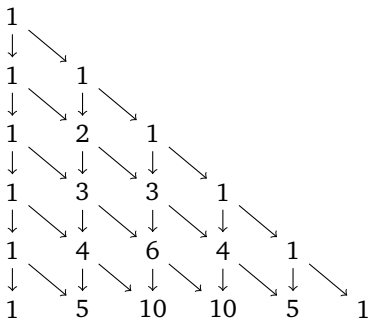
Programme une fonction récursive `binome(k,n)` qui renvoie $C_{k,n}$.

- Les cas terminaux sont pour $k = 0$ et $k = n$.
- Pour le cas général, il faut faire deux appels récursifs : un appel pour $C_{k-1,n-1}$ et un autre pour $C_{k,n-1}$.

3. **Triangle de Pascal.**

   Utilise ta fonction précédente pour afficher le ***triangle de Pascal*** : la ligne numéro n est composée des coefficients $C_{k,n}$ pour $k = 0, 1, \ldots, n$ (la numérotation n des lignes commence avec $n = 0$).

   Sur la figure de gauche le principe du calcul du triangle de Pascal : le triangle se remplit ligne par ligne, chaque nombre étant la somme issue des deux flèches qui y arrivent. Sur la figure de droite la sortie à l'écran attendue.




```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

   Indications. Voici comment afficher une chaîne de caractères à l'écran sans passer à la ligne suivante : `print(chaine,end="")`.

4. **Triangle de Pascal des termes impairs.** Modifie ta fonction précédente de façon à afficher un `"X"` à la place d'un terme $C_{k,n}$ impair et une espace pour un terme pair. Quelle figure géométrique reconnais-tu ?

```
X
XX
X X
XXXX
X   X
XX  XX
X X X X
XXXXXXXX
X       X
XX      XX
X X     X X
```

5. **Somme des chiffres.** Programme une fonction récursive `somme_chiffres(n)` qui calcule la somme des chiffres qui composent l'entier n. Par exemple avec $n = 1\,357\,869$, la fonction renvoie $n' = 39$.

   Pour cela sépare l'entier n en deux parties :
   - le chiffre des unités, obtenu par $n\%10$ (sur l'exemple $n\%10 = 9$),
   - la partie restante de l'écriture décimale de l'entier, obtenue par $n//10$ (sur l'exemple $n//10 = 135\,786$).

6. **Résidu d'un entier.** L'entier $n = 1\,357\,869$ sera divisible par 3, si et seulement si sa somme de chiffres $n' = 39$ est divisible par 3. Comment savoir si n' est divisible par 3 ? On recommence en calculant la somme des chiffres de n', ici on obtient $n'' = 12$. Comment savoir si n'' est divisible par 3 ? La somme des chiffres est $n''' = 3$ qui est bien divisible par 3. Donc n'', n' et n sont divisibles par 3.

   Le *résidu* d'un entier n est l'entier r compris entre 0 et 9 obtenu en itérant le processus : prendre la somme des chiffres et recommencer. Ainsi le résidu de $n = 1\,357\,869$ est $r = 3$.

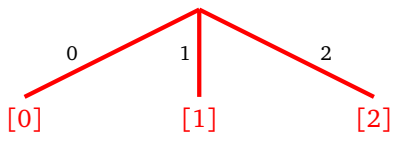
   Programme une fonction récursive `residu_chiffres(n)` qui calcule le résidu r de l'entier n.

   Indications. La fonction `residu_chiffres()` fait d'abord appel à la fonction précédente `somme_chiffres()` puis fait un appel récursif.

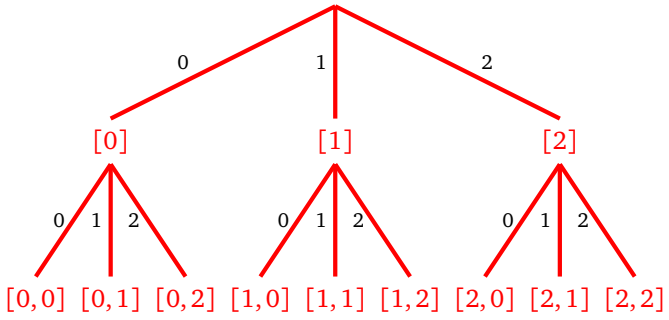
Cours 3 (Parcours d'arbre)**.**

Arbre. On souhaite construire toutes les listes possibles de n éléments formées avec les trois choix $0, 1, 2$.

- Si $n = 1$, il n'y a que trois listes possibles d'un seul élément chacune : $[0]$, $[1]$, $[2]$.
- Si $n = 2$, il n'y a que 9 listes possibles ayant deux éléments : $[0,0]$, $[0,1]$, $[0,2]$, $[1,0]$, $[1,1]$, $[1,2]$, $[2,0]$, $[2,1]$, $[2,2]$.
- On modélise cela sous la forme d'un arbre : les arêtes sont les choix 0, 1 ou 2. Les sommets sont les listes obtenues.

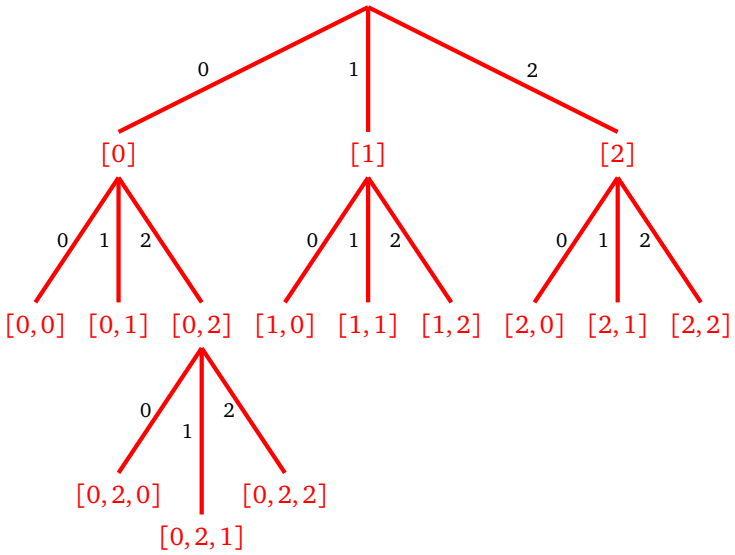


Arbre pour $n = 1$.



Arbre pour $n = 2$.

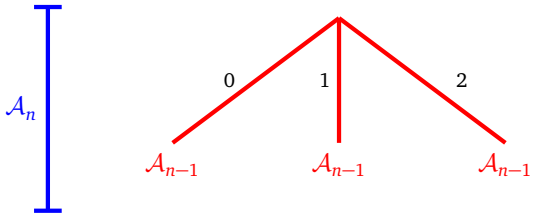
- Pour le cas général, il y a 3^n listes. Si on regarde l'arbre du haut vers le bas, alors on descend d'un sommet en rajoutant 0, 1 ou 2 *à la fin* de la liste.



Une partie de l'arbre pour $n = 3$.

Algorithme récursif.
L'idée récursive pour construire toutes ces listes est la suivante :

- on suppose que l'on a construit toutes les listes possibles de longueur $n-1$,
- on ajoute 0 *en tête* de chacune de ces listes,
- puis on ajoute 1 *en tête* de chacune de ces listes,
- enfin on ajoute 2 *en tête* de chacune de ces listes.
- On obtient donc 3 fois plus de listes qu'au départ.
- En terme d'arbre : pour construire l'arbre $\mathcal{A}_n$, on part de trois copies de l'arbre $\mathcal{A}_{n-1}$ reliées par un même sommet au-dessus avec des arêtes pondérées par 0, 1 et 2.



Construction récursive de l'arbre $\mathcal{A}_n$.

Fonction.
Voici la fonction `parcours(n)` qui renvoie toutes les listes possibles ayant n éléments.

```
def parcours(n):
    # Cas terminal
    if n == 1:
        return [[0],[1],[2]]    # ou bien n == 0 il faut [[]]

    # Cas général
    sous_liste = parcours(n-1)
    liste_deb_0 = [ [0] + x for x in sous_liste ]
    liste_deb_1 = [ [1] + x for x in sous_liste ]
    liste_deb_2 = [ [2] + x for x in sous_liste ]
```

```
    liste = liste_deb_0 + liste_deb_1 + liste_deb_2
    return liste
```

- Pour $n = 1$, la fonction renvoie la liste $[[0],[1],[2]]$.
- Pour $n = 2$, il y a $3^2 = 9$ éléments : $[[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2]]$.
- Pour $n = 3$, il y a $3^3 = 27$ éléments : $[[0,0,0],[0,0,1],[0,0,2],[0,1,0],[0,1,1],\ldots,[2,2,1],[2,2,2]]$.

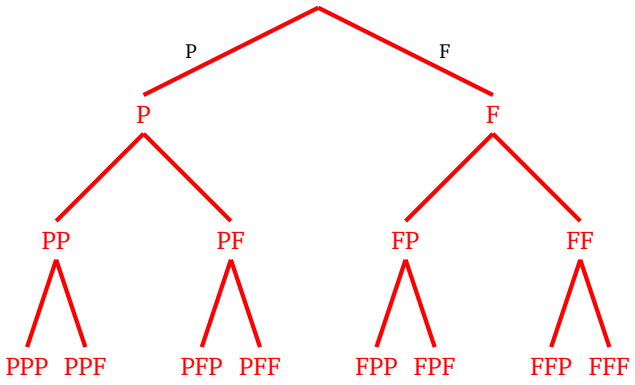
Activité 3 (Parcours d'arbre)**.**

Objectifs : résoudre des problèmes en parcourant des arbres.

1. **Pile ou face.**

   Programme une fonction `pile_ou_face(n)` qui renvoie la liste de tous les tirages possibles à pile ou face avec n lancers. Par exemple :
   - pour $n = 1$ (cas terminal), la fonction renvoie `['P','F']` (soit pile, soit face) ;
   - pour $n = 2$, elle renvoie `['PP','PF','FP','FF']` (premier tirage pile/pile, ...) ;
   - pour $n = 3$: `['PPP','PPF','PFP','PFF','FPP','FPF','FFP','FFF']`.

   Indications. Base-toi sur le modèle de parcours d'arbre du cours ci-dessus avec ici deux choix au lieu de trois.



2. **Réduire des listes emboîtées.**

   On considère une liste qui peut contenir des entiers, ou bien des listes d'entiers, ou bien des listes contenant des entiers et des listes d'entiers...

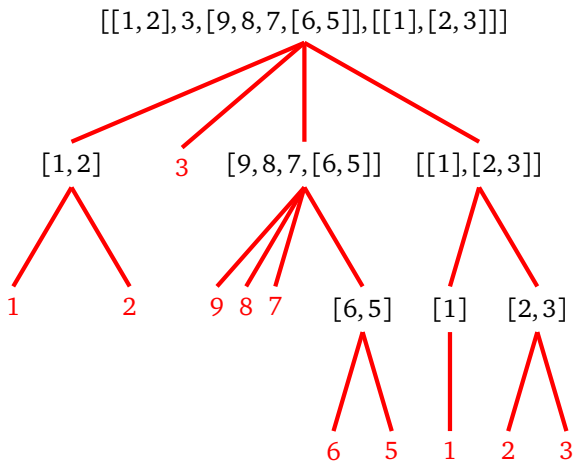
   Par exemple :

   `[ [1,2], 3, [9,8,7,[6,5]], [[1],[2,3]] ]`

   On veut en extraire les éléments atomiques (les entiers) :

   `[1, 2, 3, 9, 8, 7, 6, 5, 1, 2, 3]`

   Voici comment modéliser les imbrications sous forme d'un arbre : les sommets sont soit des entiers, soit des listes. Pour les sommets qui sont des listes ses enfants sont les éléments de cette liste.



Programme une fonction `une_seule_liste(liste)` qui effectue la tâche demandée. C'est assez simple avec un algorithme récursif.

- Définir une liste des éléments extraits qui au départ est une liste vide.
- Pour chaque élément de la liste :
  - soit c'est un entier (cas terminal) et on l'ajoute à la liste des éléments extraits,
  - soit c'est une liste (cas général) et par un appel récursif on en extrait ses éléments. On ajoute ces éléments à la liste des éléments extraits.
- Renvoyer la liste des éléments extraits.

Indications. Pour savoir si un élément est un entier ou bien une liste, tu peux utiliser la fonction `isinstance(element,type)`. Par exemple :

- `isinstance(5,int)` renvoie « Vrai »,
- `isinstance(7,list)` renvoie « Faux ».

Challenge. Essaie de programmer cette fonction sans la récursivité !

Tu trouveras une très belle application de parcours d'arbres dans la fiche « Le compte est bon ».

Activité 4 (Diviser pour régner)**.**

Objectifs : séparer un problème en deux morceaux et traiter chaque morceau de façon récursive.

1. **Minimum.**

   Programme une fonction récursive `minimum(liste)` qui renvoie le minimum d'une liste de nombres. Par exemple avec la liste $[7,5,3,9,1,12,13]$ la fonction renvoie 1.

   L'idée est de séparer la liste en une partie gauche et une partie droite.

   - On traite chaque partie séparément : un appel récursif renvoie le minimum de la sous-liste de gauche et un appel récursif renvoie le minimum de la sous-liste de droite.
   - Le minimum de la liste est donc le plus petit de ces deux minimums.
   - Le cas terminal est lorsque la liste est de longueur 1.

2. **Distance de Hamming.**

   La ***distance de Hamming*** entre deux listes de même longueur est le nombre de rangs pour lesquels éléments sont différents. Par exemple les listes $[1,2,3,4,5,6,7]$ et $[1,2,0,4,5,0,7]$ diffèrent à deux endroits, donc la distance de Hamming entre les deux listes vaut 2.

   Programme une fonction récursive `distance_hamming(liste1,liste2)`.

Réfléchis au cas terminal (la longueur de la liste est 1) et à comment calculer la distance de Hamming entre deux listes connaissant la distance entre les demi-listes à gauche et la distance entre les demi-listes à droites.

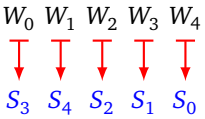
3. **Factorielle (encore !).**
   Soient a et b deux entiers avec $b > 0$. On définit une généralisation de la factorielle :
   $$p(a,b) = a(a+1)(a+2)\cdots(b-2)(b-1)$$
   (il y a $b-a$ facteurs).
   Par exemple $p(10,16) = 10 \times 11 \times 12 \times 13 \times 14 \times 15$.
   On se propose de calculer $p(a,b)$ par la formule :
   $$p(a,b) = p(a,a+k//2)\cdot p(a+k//2,b) \qquad \text{où } k = b-a.$$
   Sur notre exemple cela revient à décomposer le produit en deux sous-produits :
   $$p(10,16) = \big(10 \times 11 \times 12\big) \times \big(13 \times 14 \times 15\big) = p(10,13) \times p(13,16).$$
   Transforme cette formule en un algorithme récursif et en une fonction récursive `produit(a,b)`.
   Utilise ceci pour obtenir une nouvelle méthode de calcul de $n!$

Cours 4 (Dérangements).
Des couples arrivent à un bal masqué, chaque couple est déguisé en une paire de *Wonderwoman*/*Superman*. Lors de la fête les couples sont séparés et les danseurs se mélangent. Au moment du bal chaque *Wonderwoman* danse avec un *Superman*. Quelle est la probabilité qu'aucun de ces couples de danseurs soit un couple initial ?

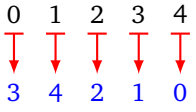
(W_0,S_0) (W_1,S_1) (W_2,S_2) (W_3,S_3) (W_4,S_4)

Avant le bal.

Pendant le bal : un exemple de mélange.



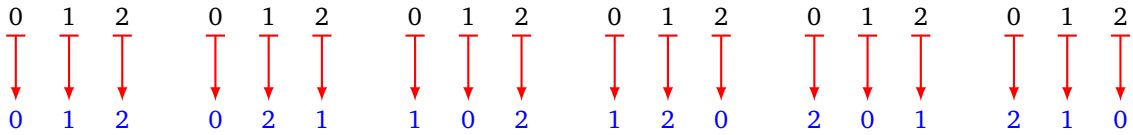
Permutation associée.



Permutation. On numérote les *Wonderwoman* de 0 à $n-1$ et on attribue le même numéro i au *Superman* en couple avec la *Wonderwoman* numéro i.
Pendant le bal les couples se reforment et la *Wonderwoman* numéro i danse avec n'importe lequel des *Superman* du numéro 0 jusqu'au numéro $n-1$ (y compris son légitime numéro i). On note ce couple $i \mapsto j$.

Permutation et dérangement.

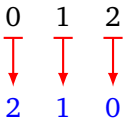
- Une ***permutation*** est une liste d'associations $i \mapsto j$, pour i, j dans $\{0, \ldots, n-1\}$.
- Le nombre de permutations possibles est $n!$
- Un exemple avec $n = 3$: voici les $3! = 6$ permutations possibles :



- Voici deux exemples avec $n = 5$:

- Un ***dérangement*** est une permutation qui vérifie $i \neq j$, pour l'association $i \mapsto j$ et ce quel que soit $i = 0, \ldots, n-1$.
- Voici la liste des dérangements pour $n = 3$:

- Les autres permutations pour $n = 3$ ne sont pas des dérangements, par exemple la permutation suivante vérifie $1 \mapsto 1$ donc n'est pas un dérangement :



- Pour les deux permutations d'ordre $n = 5$ définies en exemple ci-dessus, l'une est un dérangement et l'autre pas. Trouve qui est qui.

Modélisation. On modélise une permutation par la liste des images :

Une permutation d'ordre n sera donc représentée par une liste dans laquelle les entiers de 0 à $n-1$ apparaissent chacun une fois et une seule. Voici un exemple de permutation d'ordre 5 et sa représentation par une liste :

Activité 5 (Dérangements)**.**

Objectifs : calculer le nombre de dérangements par différentes méthodes.

Rappel du problème : Des couples arrivent à un bal masqué, chaque couple est déguisé en une paire de Wonderwoman/Superman*. Lors de la fête les couples sont séparés et les danseurs se mélangent. Au moment du bal chaque* Wonderwoman *danse avec un* Superman*. Quelle est la probabilité qu'aucun de ces couples de danseurs ne soit un couple initial ?*

Le ***nombre de dérangements*** d_n est défini par :

$$d_1 = 1 \qquad \text{et} \qquad d_n = nd_{n-1} + (-1)^n \quad \text{pour } n \geqslant 2.$$

1. Programme une fonction `derangement_classique(n)` qui renvoie d_n en utilisant une boucle.
   Indications. $(-1)^n = +1$ si n est pair et -1 sinon.
2. Programme une fonction récursive `derangement(n)` qui renvoie aussi d_n.
3. La probabilité qu'aucun des couples initiaux ne soit reformé est donnée par :
   $$p_n = \frac{d_n}{n!}.$$
   - Calcule cette probabilité pour de petites valeurs de n.
   - Compare cette probabilité avec $1/e$ (où $e = \exp(1) = 2.718\ldots$).
   - Est-ce que la convergence est rapide (quand $n \to +\infty$) ?
   - Conclure : « Il y a environ ..... % de chance qu'aucun couple initial ne soit reformé. »
4. **Dérangement ?**
   On se donne une permutation sous la forme d'une liste d'entiers de 0 à $n-1$. Programme une fonction `est_derangement(permutation)` qui teste si la permutation donnée est (ou pas) un dérangement.
   Dérangement. On rappelle que pour un dérangement on n'a jamais $i \mapsto i$.
   Exemple. La permutation codée par $[2,0,3,1]$ est un dérangement, la fonction renvoie « Vrai ». Par contre la permutation codée par $[3,1,2,0]$ n'est pas un dérangement, car $2 \mapsto 2$, la fonction renvoie « Faux ».
5. **Toutes les permutations.**
   Programme une fonction `toutes_permutations(n)` qui renvoie la liste de toutes les permutations de longueur n.
   Par exemple pour $n = 3$, voici la liste de toutes les permutations possibles :
   ```
   [[2, 1, 0], [1, 2, 0], [1, 0, 2], [2, 0, 1], [0, 2, 1], [0, 1, 2]]
   ```
   Pour cela l'algorithme est basé sur un principe récursif : par exemple si on connaît toutes les permutations à trois éléments (voir juste au-dessus), alors on obtient les permutations à 4 éléments en insérant la valeur 3 à toutes les positions possibles de toutes les permutations à 3 éléments possibles :
   - notre première permutation à trois éléments $[2,1,0]$, donne par insertion de 3 les permutations à quatre éléments $[3,2,1,0]$, $[2,3,1,0]$, $[2,1,3,0]$ et $[2,1,0,3]$;
   - ensuite avec $[1,2,0]$ on obtient $[3,1,2,0]$, $[1,3,2,0]$, $[1,2,3,0]$, $[1,2,0,3]$;
   - on continue avec les autres permutations pour obtenir en tout $4! = 24$ permutations d'ordre 4.

Algorithme.

- — Entête : `toutes_permutations(n)`
  — Entrée : un entier n.
  — Sortie : la liste des $n!$ permutations d'ordre n.
  — Action : fonction récursive.
- *Cas terminal.* Si $n = 1$ alors renvoyer la liste `[ [0] ]` (qui contient l'unique permutation à un seul élément).
- *Cas général.*
  — On effectue un appel récursif `toutes_permutations(n-1)` qui renvoie une liste `old_liste` de permutations d'ordre $n-1$.
  — Une `new_liste` est initialisée à la liste vide.
  — Pour chaque permutation de `old_liste` et pour chaque i allant de 0 à $n-1$, on insère le nouvel élément $n-1$ au rang i. On ajoute cette nouvelle permutation d'ordre n à `new_liste`.
  — On renvoie `new_liste`.

6. **Tous les dérangements.**
   Programme une fonction `tous_derangements(n)` qui à partir de la liste de toutes les permutations d'ordre n ne renvoie que les dérangements. Vérifie sur les premières valeurs de n que ce nombre de dérangements vaut bien d_n.

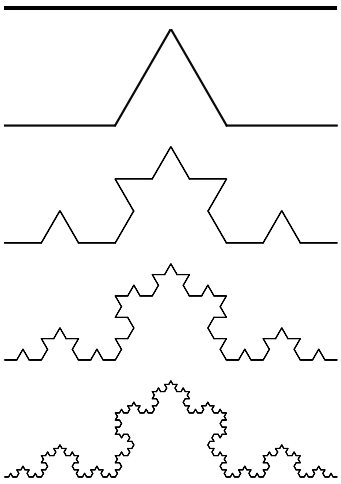
Cours 5 (Tortue `Python`)**.**
Voici un bref rappel des principales fonctions du module `turtle` afin de diriger la tortue de `Python` :

- `forward(100)/backward(100)` avancer/reculer de 100 pixels,
- `left(90)/right(90)` tourner à gauche/droite de 90 degrés,
- `goto(x,y)` aller à la position (x, y),
- `x,y = position()` récupérer les coordonnées courantes de la tortue,
- `setheading(angle)` s'orienter dans la direction donnée,
- `up()/down()` lever/abaisser le stylo,
- `width(3)`, `color('red')` style du tracé,
- `showturtle()/hideturtle()` affiche/cache le pointeur,
- `speed('fastest')` pour aller plus vite,
- `exitonclick()` à placer à la fin.

Activité 6 (Tortue récursive)**.**

Objectifs : tracer des fractales à l'aide de la tortue et des algorithmes récursifs.

1. **Le flocon de Koch.**
   Le flocon de Koch est une fractale définie par un processus récursif. À chaque étape, chaque segment est remplacé par 4 nouveaux segments plus petits formant une dent. Voici les étapes en partant d'un segment horizontal.



Définis une fonction récursive `koch(l,n)` qui trace le flocon de Koch d'ordre n ; ℓ est un paramètre de longueur.

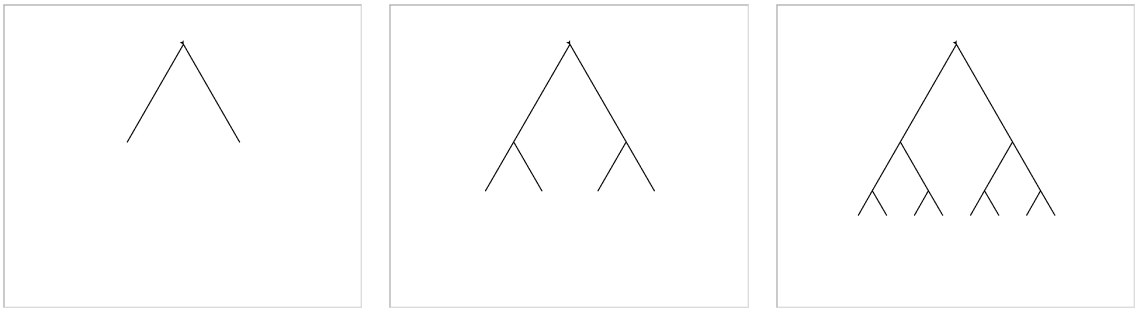
Le principe du tracé est le suivant :

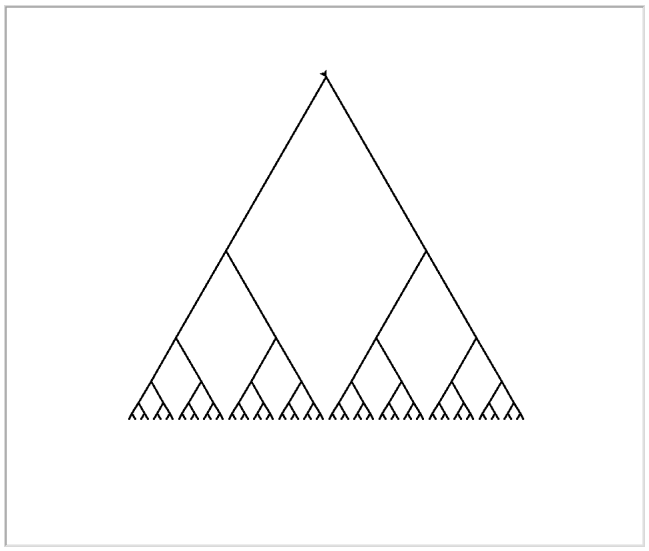
- *Cas terminal.* Si $n = 0$, tracer un segment de longueur ℓ.
- *Cas général.*
  - — Tracer le flocon de Koch d'ordre $n-1$, associé à la longueur $\ell/3$.
  - — Tourner un peu vers la gauche.
  - — Tracer le flocon de Koch d'ordre $n-1$, associé à la longueur $\ell/3$.
  - — Tourner vers la droite.
  - — Tracer le flocon de Koch d'ordre $n-1$, associé à la longueur $\ell/3$.
  - — Tourner un peu vers la gauche.
  - — Tracer le flocon de Koch d'ordre $n-1$, associé à la longueur $\ell/3$.

2. **Arbre binaire.**

   Adapte la fonction précédente en une fonction `arbre(l,n)` pour dessiner des arbres dont la profondeur dépend d'un paramètre n (ℓ est un paramètre de longueur).

   Voici les dessins pour $n = 1$, $n = 2$, $n = 3$ et $n = 6$.

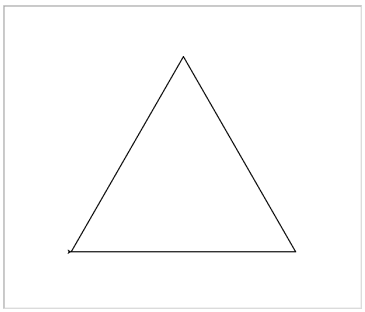
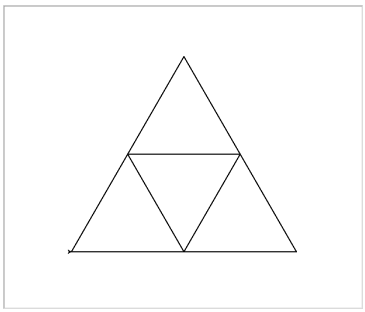
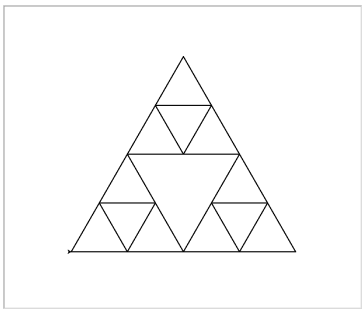
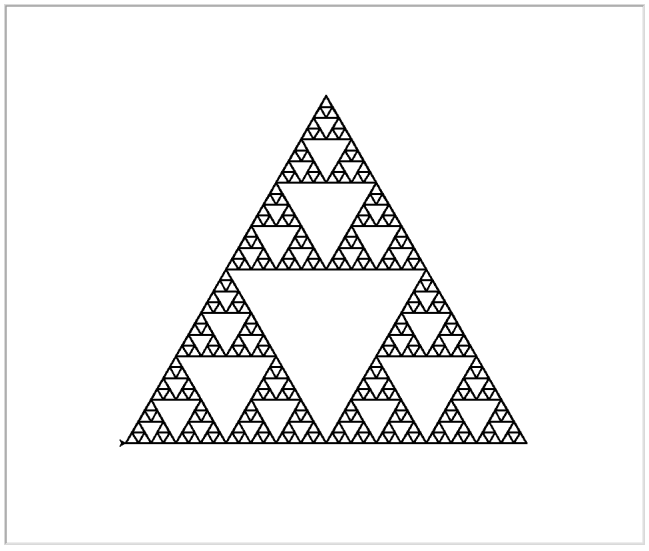




3. **Triangle de Sierpinski.**

   Trace les différentes étapes qui conduisent au triangle de Sierpinski par une fonction récursive `triangle(l,n)` (ℓ est un paramètre de longueur, n est un paramètre de profondeur).

   Voici les dessins pour $n = 1$, $n = 2$, $n = 3$ et $n = 6$.

   Le principe récursif est le suivant :

   - Si $n = 0$ ne rien faire.
   - Sinon répéter trois fois :
     — faire un appel récursif `triangle(l/2,n-1)`,
     — avancer de ℓ pas,
     — tourner de 120 degrés.

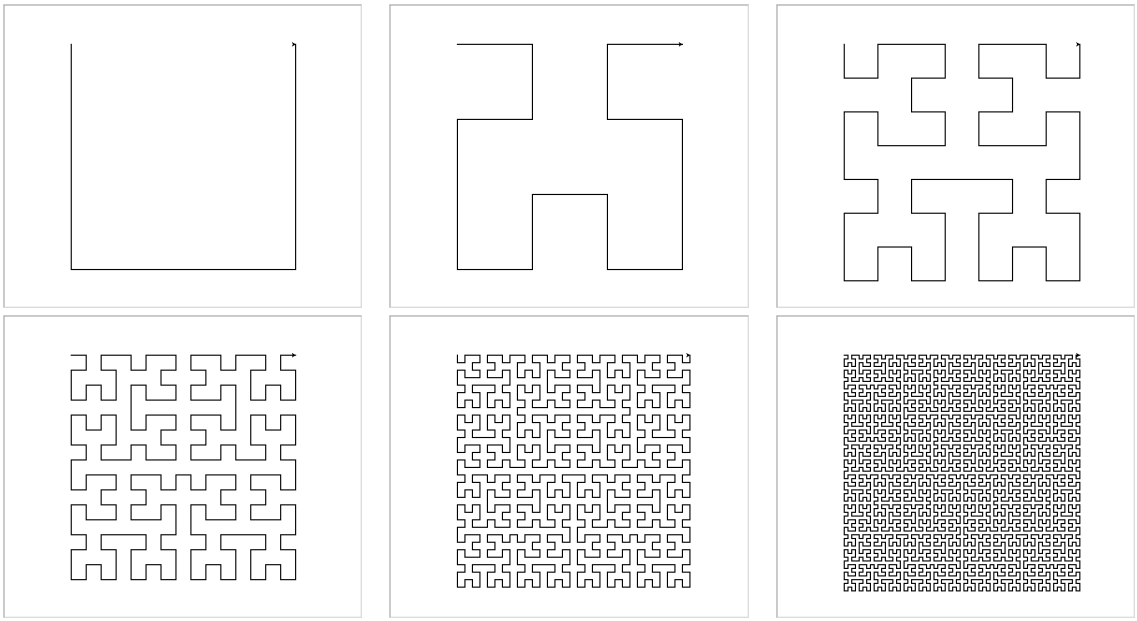
4. **Courbe de Hilbert.**

   Trace les premiers pas de la courbe de Hilbert à l'aide d'une fonction récursive `hilbert(angle,n)`.

   Le tracé récursif se fait selon le principe expliqué ci-dessous, ℓ est une longueur fixée à l'avance, θ est l'angle qui vaut ±90 degrés, n est l'ordre du tracé.

   - Si $n = 0$ ne rien faire.
   - Sinon :
     — tourner à gauche de $-\theta$,
     — faire un appel récursif avec comme paramètres $-\theta$ et l'ordre $n-1$,
     — avancer de la longueur ℓ,
     — tourner à gauche de $+\theta$,
     — faire un appel récursif avec comme paramètres $+\theta$ et l'ordre $n-1$,
     — avancer de la longueur ℓ,
     — faire un appel récursif avec comme paramètres $+\theta$ et l'ordre $n-1$,
     — tourner à gauche de $+\theta$,
     — avancer de la longueur ℓ,
     — faire un appel récursif avec comme paramètres $-\theta$ et l'ordre $n-1$,
     — tourner à gauche de $-\theta$.

   Voici les dessins pour l'angle initial valant $\theta = +90$ degré et des ordres n allant de 1 à 6.



5. **Fractale aléatoire.**

   Programme une fonction récursive `fractale_cercle(l,n)` qui contient une part d'aléatoire. La fonction dessine un quart de cercle, puis décide au hasard (une chance sur deux par exemple) si elle trace un plus petit cercle par un appel récursif, ensuite elle continue avec le tracé d'un quart du cercle initial et décide alors de tracer éventuellement un plus petit cercle…

   Voici des dessins pour $n = 1$, $n = 2$, $n = 3$ et $n = 4$. Bien sûr d'une fois sur l'autre le dessin change au hasard.